FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Codebase modernisation for car sharing Android application
Student:	Branislav Bilý
Supervisor:	Ing. Václav Jirovský, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

The task is to find obsolete parts of the current car-sharing student project application and to modernise said obsolete parts using modern tools following developer guides, such as using programming language Kotlin, MVVM architecture and Compose. Focus on the area, where modernisation will have the most impact - identify these areas together with the project's student team. Follow these steps:

1. Familiarise yourself with the project and used technologies
2. Analyse the codebase and research possible improvements
3. Refactor obsolete code to meet modern standards
4. Thoroughly test and document new implementation
5. Compare previous and current implementation and justify refactoring

---

Electronically approved by Ing. Michal Valenta, Ph.D. on 16 February 2023 in Prague.

Bachelor's thesis

CODEBASE MODERNISATION FOR CAR SHARING ANDROID APPLICATION

Branislav Bilý

Faculty of Information Technology
BI-WSI – Web and Software Engineering
Supervisor: Ing. Václav Jirovský, Ph.D.
May 11, 2023

Contents

List of Figures

List of code listings

I would like to thank my supervisor Ing. Václav Jirovský, Ph.D. for his guidance and experience. I would also like to thank my family for supporting me throughout my studies, especially my brother. Lastly, my gratitude goes to my friends.

Declaration

I declare that I have prepared the submitted thesis independently and that I have listed all the information sources used in accordance with the Methodological Guideline on the Ethical Preparation of University Theses. I acknowledge that my thesis is subject to the rights and obligations arising from Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a license agreement for the use of this thesis as a school work pursuant to Section 60 (1) of the Copyright Act for a definite period of time until the expiry of the protection under the Agreement. The use of the submitted work is governed by the Cooperation Agreement concluded in connection with the cooperation between the Czech Technical University in Prague and ŠKODA AUTO a.s. and ŠKODA AUTO DigiLab s.r.o. on the research project "CarSharing for university students", published in the Register of Contracts at https://smlouvy.gov.cz/smlouva/5973503. I am bound by a Non-Disclosure Agreement that I will not disclose to any third party any confidential information that I have obtained during my work on the Project.

In Prague on May 11, 2023 .....

Abstract

This bachelor thesis is focused on refactoring the code of an Android application, that was written using older technologies and architectures. The main part of the thesis is to show the evolution of Android development, how it changed over a couple of years, and where it is headed. The second part shows one method, how to take an older application and gradually refactor code, that will be easily expandable and testable.

Keywords Android mobile application, refactor, application architecture, Jetpack Compose, technical debt

Abstrakt

Tato bakalářská práce se zabývá refaktoringem Android aplikace, která byla napsána s využitím starších technologií a architektur. Hlavní část se zabývá evolucí Android vývoje, jak se změnil v průběhu několika let a kam směřuje. Druhá část obsahuje jeden způsob, jak postupně refaktorovat kód starší aplikace, aby byl jednoduše rozšiřitelný a testovatelný.

Klíčová slova Android mobilní aplikace, refaktorování, architektura aplikace, Jetpack Compose, technický dluh

9

Introduction

Smartphones have become an inseparable part of our lives. From listening to music to sharing cars, smartphones, and mobile applications that run on them are becoming more and more complex to cater to users who want to do more and more things from the comfort of their palms. Behind every application, there are developers who try to bring their products to the market as fast as possible without any bugs or hidden problems.

Software development is a difficult and ever-changing field. What was once considered best practice a few years back is now an outdated technique that no one uses. Even Java, one of the most popular programming languages, is not immune to advancements in software development. Therefore, the problem of rewriting applications is not something new, but it is something necessary. The most drastic way is to rebuild the whole application from the ground up, using new technologies and frameworks. This is the approach that Google chose for Android with Compose, into which we will look at in a later chapter. Another way is to refactor parts of the application one by one, based on where the new technologies will have the greatest impact. This is the approach I chose for a car-sharing application.

The goal of the thesis is to identify a part of an Android application that is most suitable for refactoring and to refactor that said part using the newest technologies and architectures that are recommended by Google. Refactoring will involve rewriting Java code in Kotlin, using a new declarative way to create layouts using Compose, and changing the architectural pattern from MVC to MVVM and later to MVI, all while following OOP principles and maintaining functionality. By the end of the refactoring process, the technical debt of the application should be lower, and the code should be easier to read, expand, and test.

Chapter 1
Refactoring

This chapter focuses on what refactoring is, when and how to refactor, and what can happen when we neglect our code. It will also define a couple of terms that are important to understand when we talk about refactoring.

1.1 Technical debt

In any software system, there is a certain essential complexity required for the software to perform its function. However, most software systems also contain unnecessary complexity that make them harder to understand without providing any benefits. Technical debt is a metaphor that treats this unnecessary complexity as financial debt. Interest payments are the extra resources that will be needed to be spent on the system to add new features [1]. Technical debt accumulates when the development team is either forced or free to choose an easy but limiting solution.

Just like financial debt, technical debt is sometimes necessary to bring a product to market. Just like monetary debt, technical debt also accumulates interest. This can make it expensive to repay the debt when it is left alone for a long time.

Unrealistic release dates are the biggest culprit for technical debt. Managers are often not software engineers, so they do not understand the challenges that come with software development. Programmers are also notoriously bad at estimating the time required to implement new features. Combining these two factors often results in cutting corners during development to expedite feature delivery. This adds unnecessary complexity that will need to be reduced in the future. After a few of these rushed deliveries, it can become extremely time-consuming and difficult to add new features. This is exactly what happened with the Oracle database, where after years of tight deadlines, the addition of one line of code could potentially break thousands of different tests [2].

Technical debt can also increase when the development team suffers from **insufficient quality control**. Every team should have documented coding standards that need to be followed and enforced. These standards can include how to version code, write classes and methods, and what design patterns should be used. To enforce these rules, senior developers need to check the code of their peers and point out any problems. When a team does not have these standards in place, every developer will solve problems their way, which makes the code harder to read and maintain. The practice of reviewing the code of peers is called *code review*, where senior developers spend time reviewing the code of others instead of writing code themselves. It is a type of insurance policy where, by paying with the time of senior developers, the development team gets assurance that less experienced developers do not introduce unnecessary complexity or hidden problems.

With the rise of cloud trends, parallel programming or increased emphasis on security, **in-**

sufficient expertise is another problem that can create technical debt. Teams may adopt tools and technologies without fully understanding their benefits and drawbacks. Sometimes these tools can be even unnecessary and can needlessly increase the complexity of software.

The complexity of software is also increased when there is **no documentation**. This not only makes it harder for new developers to understand and write new code, but it also impacts senior developers on the project [3]. The ratio of time spent reading versus writing code is well over 10 to 1 [4], so by making it easier to read, we are making it easier to write.

1.2 Refactoring

▶ **Definition 1.1** (Refactoring as a noun)**.** *"A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour" [5].*

Refactoring refers to *any* change made to the software with the intention of making it easier and more cost-effective to modify. This can include rewriting code in a different programming language, completely overhauling the system's architecture, or simply switching to a new library. However, the key requirement is that the software's observable behaviour *must not be altered*. After refactoring, the software should continue to function in the same way as it did before the changes were made. It is not permissible to add new features or alter any existing functionality during the refactoring process.

▶ **Definition 1.2** (Refactoring as a verb)**.** *"To restructure software by applying a series of refactorings without changing its observable behaviour" [5].*

So by refactoring we are trying to manage the technical debt of the software system.

1.3 When to refactor

There are several motivations when to refactor. The main ones include:

- It is difficult to add new functionality
- There is code area that is consistently buggy
- Encountering code smells
- New better technology can be used

As we can see, refactoring can be often motivated by technical debt [6].

It is difficult to add new functionality

When we need to modify parts of the code that are unrelated to new features, we often resort to adding fixes, exceptions, and workarounds to make it functional. This is a clear indication that we should refactor the code first before implementing any new features.

There is code area that is consistently buggy

Frequent occurrences of bugs in a specific section of code can suggest that the code was either poorly designed or written. Furthermore, it's probable that the code was not adequately tested.

Encountering code smells

▶ **Definition 1.3** (Code smell)**.** *"A surface indication that usually corresponds to a deeper problem in the system" [7].*

Just as smelling smoke in our home is a sign that something is wrong, code smell is a sign of a bad code. Code smells include:

- Duplicated code
- Nested if statements
- Global states and variables
- Long methods and large classes
- Unreachable code

The great thing about code smells is that they are easy to spot, even for inexperienced developers, and are fairly easy to correct. With today's powerful IDEs, some types of code smells, such as duplicated or unreachable code, can be corrected with the press of a button.

It is important to keep in mind that not all code smells are created equal. For example, some long methods are long because they need to be. Therefore, before refactoring code mindlessly, we need to ask ourselves simple questions to determine whether a particular code smell indicates a problem that needs to be addressed or not [7].

New better technology can be used

This does not mean that we always have to use the latest technologies and libraries, as this can ironically lead to worse code quality. However, there may be trends in the way a product is developed that developers should be prepared to follow. For example, Java was the default programming language for Android development since the introduction of the Android platform in 2008 [8]. However, in just two years, Kotlin became the first-class language for Android apps in May 2017 [9] and the preferred language in May 2019 [10]. Therefore, ever since May 2019, every Android application written in Java is accumulating technical debt. The process of rewriting Java into Kotlin is explained in section 5.2.

1.4 How to refactor

To prepare for the refactoring process, it's essential to have a good understanding of version control systems. This is because we can't risk altering the code's behaviour, and we must have the ability to revert to a previous version where the code worked correctly. In this thesis, Git is utilized as the version control system, but other systems can also be used.

Once we have the version control system in place, we can proceed with the refactoring process, which can be broken down into three distinct stages. The first stage is **refactoring**, which involves identifying and eliminating code smells, following new design patterns, or making any other necessary improvements to the code. Once the changes have been made, we move on to the second stage, which involves **running tests** to ensure that the behaviour of the code has not been affected. If any issues are found, we return to the first stage and repeat the refactoring process. If the tests pass, we can move on to the final stage, which is **committing** the changes. This involves saving the current state of the application using Git, after which we can continue with the refactoring process.

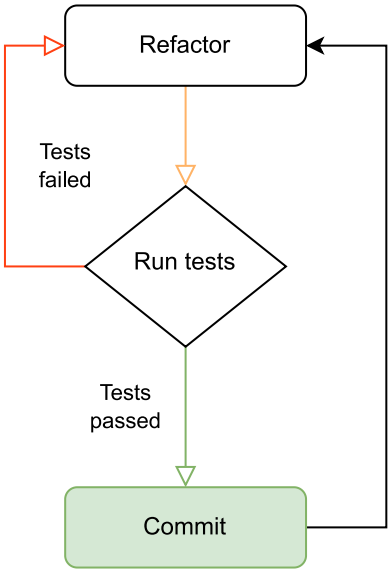

Figure 1.1 Flowchart showing 3 stages of refactoring

1.5 Neglecting refactoring

I will start this section by drawing a comparison between software and a building with glass windows. There exists a famous theory titled *"Broken Windows"* by James Q. Wilson and George L. Kelling [11], which demonstrated that when one broken window is left unrepaired, it signals that no one cares, and subsequently, breaking more windows costs nothing. This analogy is particularly relevant to software development. When there is already a poorly designed part of an application (a broken window), it becomes easier to justify writing poorly designed code (breaking more windows). Unfortunately, this process can often occur subconsciously, which is why development teams and management need to proactively consider the importance of refactoring.

1.6 When not to refactor

In this chapter, I have highlighted the importance of refactoring in software development. However, it is important to note that there are situations where refactoring may not be the best solution. It is always a trade-off between the benefits of refactoring and the time and effort required to implement it. For example, if the feature to be added is small and the required refactoring is extensive, it may be more efficient to just implement the feature without refactoring. Additionally, sometimes it may be more feasible to rewrite the application from scratch. Deciding when to refactor or when to implement new features without refactoring requires experience and professional judgement. Ultimately, the goal of refactoring is to improve product quality and speed up development, but it is important to consider the costs and benefits of each approach before making a decision [1].

Chapter 2
Android Platform Specifics

This chapter is focused on the basics of Android development and gives a brief overview of used technologies, tools, and architectures.

2.1 Android Platform

Android Platform was launched in 2007 by the Open Handset Alliance, an alliance of many companies that included Google, HTC, Motorola, and others [12]. It had turbulent development, especially thanks to the rapid increase of touchscreen popularity that started with the release of the first iPhone when the Android team was still focusing on developing Android for phones with physical keyboards [13]. This lead to a great deal of ramifications to this day, since the entire platform carries a lot of technical debt. In spite of that, Android is the leader in the market share of all smartphones, with 71.63% market share in Q1 of 2023 [14].

2.1.1 Android SDK

The Android SDK (software development kit) is a collection of tools used to develop Android applications. It includes necessary libraries, a debugger, an emulator, and much more. Whenever a new version of the Android operating system is released, a new version of the SDK is also released to support the latest features. Since the SDK is closely tied to the Android version, there are multiple versions of this collection available.

This creates a dilemma of choosing the minimum SDK version to support in an Android app. By choosing an older SDK version, developers can ensure that their app will be compatible with a wider range of devices, but they may also need to make an additional effort to implement certain features or workarounds for limitations of older versions. On the other hand, choosing a newer SDK version can give developers access to the latest features and tools, but may limit the devices on which the app can be installed, as shown in 2.1 figure. Ultimately, the decision of which SDK version to support will depend on the specific needs of the app and its target audience.

2.1.2 Android Studio

Android Studio is the official integrated development environment (IDE) for Android app development. It is based on IntelliJ IDEA by JetBrains, which makes it a powerful tool for development. In addition to the features included in IntelliJ, such as code completion, renaming, and code simplification, Android Studio includes tools specifically for Android development. These

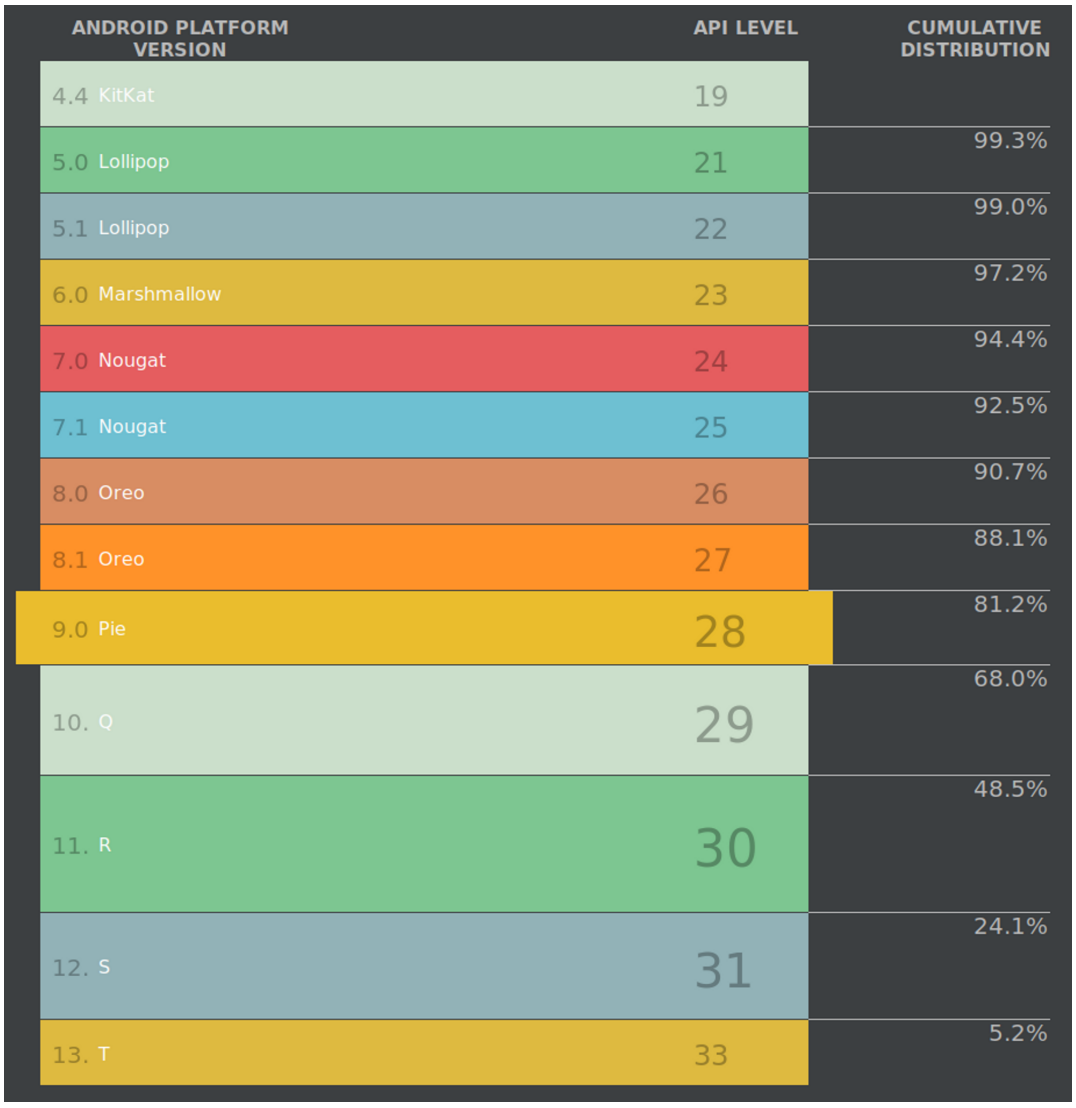

Figure 2.1 Different Android versions, their API versions and cumulative distributions [15]

tools include an emulator for emulating Android devices, Logcat for logging system messages, the Layout Inspector, which allows developers to inspect the current layout on the device in detail, and much more.

2.1.3 Activity and Fragment

An Activity in Android is a "*single, focused thing that a user can perform*" within an application [16]. It is the main building block for creating user experiences. For example, an app may have separate Activities for user login, password resetting, and registration. If a user is currently in the login Activity and wants to reset their password, the app can simply switch to the appropriate Activity.

"Fragment is a piece of application's user interface that is placed inside an Activity" [17]. To continue with the Activity example, we would have one main Activity whose sole purpose would be to host Fragments. Instead of changing Activities to reset a password or register, we could just change the Fragment that the Activity is hosting. This was the way most applications were built before 2.1.8 Compose.

2.1.4 Views

View is a basic building block for creating user interface components. From this View, every UI component called *Widget* is built from. We can organize these *Widgets* into *ViewGroups*, which are containers that house others Views or ViewGroups [18]. The most notable ViewGroup is *ConstraintLayout*.

ConstraintLayout allows us to create responsive layouts directly inside Android Studio by drag-and-dropping components, instead of editing any XML (which is the language used for specifying Views). It works by adding two constraints, vertical and horizontal, and by adjusting the constraint bias, we can move components within the ConstraintLayout. Since we are working with biases and not pixels, we can be sure that our layout will look the same on almost all screen sizes [19].

2.1.5 Resources

When developing apps, we need to save additional files and static content such as images, layout definitions, and text displayed to the user. It is recommended to store these files separately to maintain them independently. We can provide alternate resources based on device configuration by grouping them in specially-named resource directories. At runtime, Android uses the correct resource directory, allowing us to customize the user experience for the device [20]. For example, we can use translated text in the app based on the device's preferred language.

2.1.6 Emulator

Emulator emulates Android devices, allowing us to quickly and efficiently test our application on various devices and Android API levels, without the need to own any physical device. This emulated Android device can specify the location of the device, simulate phone calls, messages, rotation, and much more.

In most cases, the emulator is the best option for our testing needs. Alternatively, we can install the application on a real Android device.

2.1.7 LiveData

LiveData is a special type of data class that is observable and lifecycle-aware. Whenever the value of the LiveData object changes, it notifies all its observers. The lifecycle awareness guarantees that only the observers that are in an active lifecycle state are notified of the change. This ensures that the UI always has the latest data and avoids the need for manual lifecycle handling. LiveData objects also avoid memory leaks, as they can clean up after themselves when their lifecycle owner is destroyed [21].

2.1.8 Jetpack Compose

Google defines Jetpack Compose (later only Compose) as: "*Android's recommended modern toolkit for building native UI. It simplifies and accelerates UI development on Android. Quickly bring your app to life with less code, powerful tools, and intuitive Kotlin APIs*" [22].

UI in Compose is created by calling *Composable* functions that create UI elements called Composables. These Composables create trees, based on the order they were called.

Composable functions

Composable functions are a fundamental building block in Compose. They are a new type of function in Kotlin that allows us to describe our UI in code. These functions can call other Composable functions, and we can change the behaviour of our UI by passing parameters to these functions. Instead of building UI using XML tags, we create UI by nesting function calls. In the code listing 2.1, we can see an example of a Composable function that calls the built-in Composable function Button.

■ **Code listing 2.1** Example of a Composable function composing a disabled button with text

```
@Composable
fun MyButton() {
    Button(
        onClick = { /* Function called when button is clicked */ },
        enabled = false,
    ) {
        Text(text = "Click me")
    }
}
```

Why was Compose created

This is a completely different way of building UI. So why the drastic change and why now? After a couple of tries to refactor the underlying technology of Android Views, Google decided to write a new toolkit for building UI from scratch. The main two factors were:

Decoupling (separating) UI toolkit from the Android platform. As was mentioned in the 2.1.1 Android SDK subsection, SDK is closely coupled with the Android version, meaning that it can only be used to create Android UI and cannot be updated without updating the Android version, which limits the speed and frequency of updates. With Compose, we can use the same UI toolkit to write Android, iOS (operating system on iPhones and iPads), desktop, and web apps [23]. This significantly reduces the amount of code required to make a multi-platform application.

Declarative model for the creation of the UI. There is an industry-wide shift towards this model. Fundamental change is that *Widgets* are stateless and do not expose any setters or

getters. When we want to update a Widget, we simple call the same function that created the Widget with new parameters. This approach was first pioneered by web community with frameworks like React with great success.

It is safe to say that most UIs across all platforms will be developed using a similar approach as Compose [24].

2.2 Kotlin

"Kotlin is a modern statically typed programming language used by over 60% of professional Android developers that helps boost productivity, developer satisfaction, and code safety" [25].

Kotlin can be compiled into bytecode and executed on the Java Virtual Machine (JVM), which means it can run on any system that supports Java. This feature makes it simple to incorporate Kotlin into projects developed using Java, such as Android applications. Furthermore, Android Studio offers an almost seamless transition process, as it can automatically convert Java classes to Kotlin classes.

2.3 Gradle

Gradle is an open-source build automation tool. It handles external dependencies, defining build configuration, building applications, running tests, and much more. Gradle is:

Highly customizable – Gradle is modeled in a way that is customizable and extensible in the most fundamental ways.

Fast – Gradle completes tasks quickly by reusing outputs from previous executions, processing only inputs that changed, and executing tasks in parallel.

Powerful – Gradle is the official build tool for Android, and comes with support for many popular languages and technologies [26].

2.4 Lifecycle

"As a user navigates through, out of, and back to our app, the application goes through different stages of Lifecycle". Lifecycle has its own callbacks, allowing us to declare specific behaviour for each stage. Activity, Fragment, Composables, or ViewModels (check 3.5.2 MVVM pattern), all have their Lifecycles [27].

2.4.1 Lifecycle of an Activity

The Activity class provides multiple callbacks to declare how the Activity will behave when the user leaves and re-enters the Activity. For example, when the user changes the screen orientation, Android re-draws the entire Activity, losing all the progress the user made. Similarly, when the user receives a phone call, progress can be lost. We can save the progress by overriding *onStop()* callback, which is called when the Activity is no longer visible to the user. Then, we can load the saved progress overriding *onRestart()* callback [27]. As can be seen in figure 2.2, the Lifecycle of an Activity is complicated but very important. Proper usage of an Activity's lifecycle is essential for Android apps to function properly.

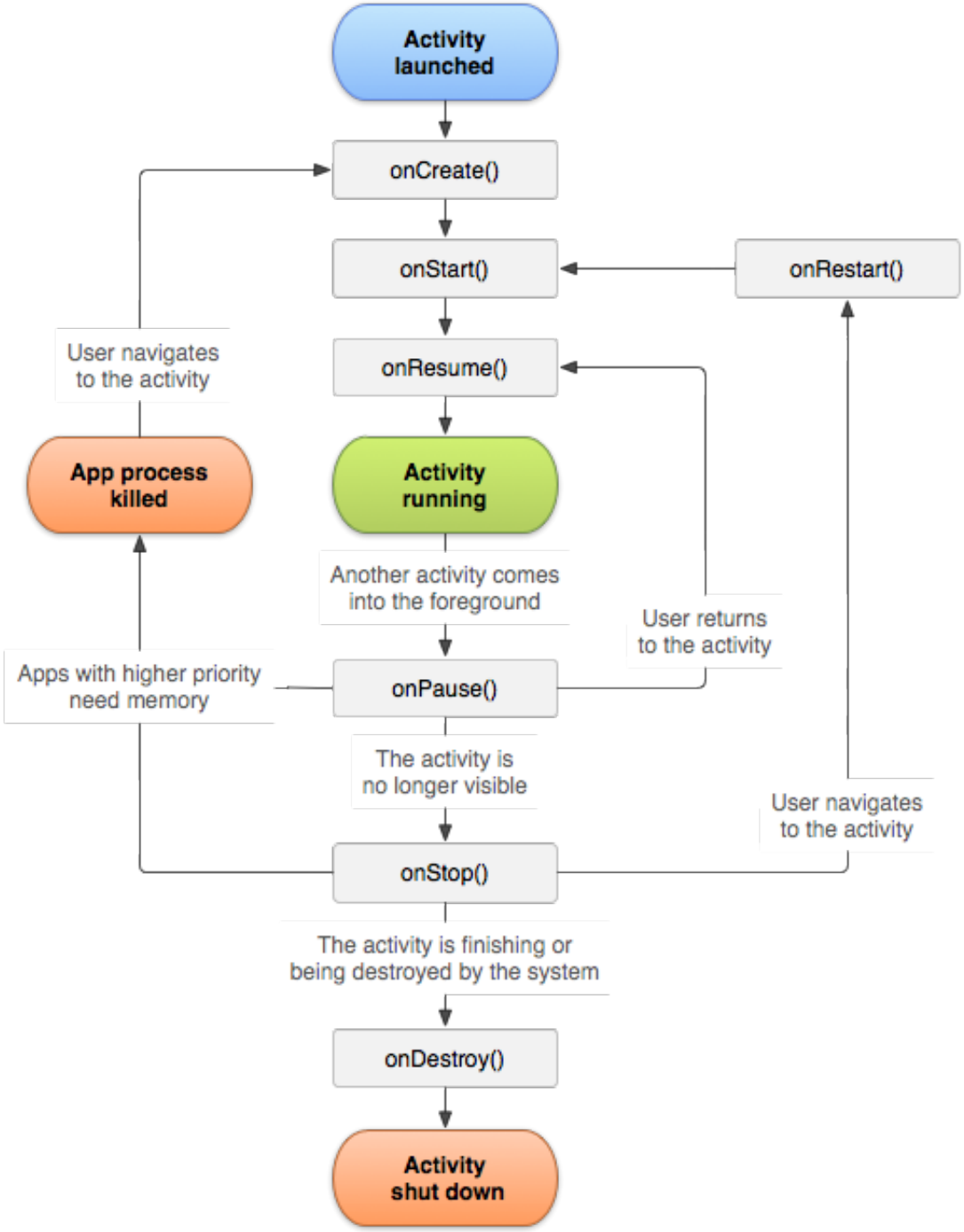

Figure 2.2 Overview of Activity Lifecycle [27]

2.4.2 Lifecycle of a Composable

The Lifecycle of a Composable is handled in a very different way than the lifecycle of an Activity. When Compose runs Composable functions for the first time, it keeps track of the order in which they were called. When the state of the app changes, Compose schedules an *recomposition*, which means re-executing only the Composable functions that were affected by the new state. This recomposition can happen 0 or more times, depending on the changes in the app's state [28]. In contrast to the Activity lifecycle, the lifecycle of a Composable is much simpler, and it is not something that developers need to worry about in most cases. Figure 2.3 illustrates the difference between the two lifecycles and highlights the simplicity of the Composable lifecycle.

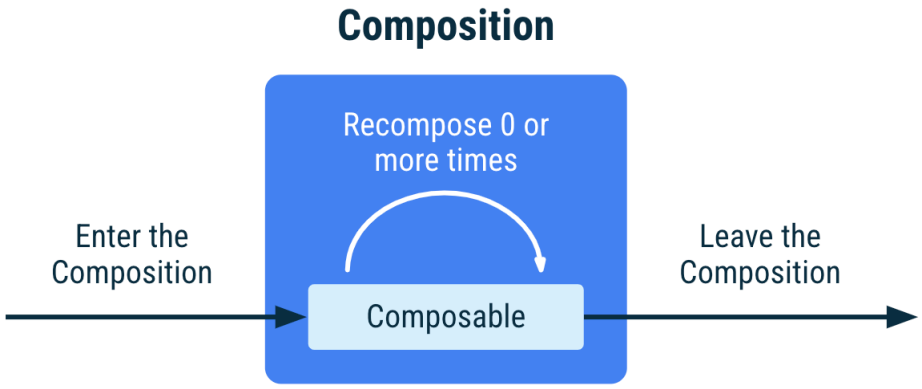

Figure 2.3 Lifecycle of a Composable [28]

2.5 App architecture

As apps become more complex, it is crucial to develop them in a way that is easy to scale and test. App architecture provides a framework for defining boundaries between different parts of the app and specifying their respective responsibilities. To achieve these goals, we should design our app architecture according to specific principles [29].

2.5.1 Separation of concerns

"The SoC principle identifies the parts of an application with a specific purpose and encapsulates these parts in closed units. These units only communicate with each other using specified interfaces. Thanks to this principle, the software - which would have otherwise been overly complicated - is divided up into manageable components" [30].

A great example of not following this principle are UI-based classes, such as *Activity* of *Fragment*. These classes often not only contain logic that handles UI but also business logic or data models, causing them to easily grow into hundreds if not thousands of lines of code. As a result, it becomes difficult to test and scale them. This type of class is often referred to as a *God class* and should be avoided whenever possible [29].

2.5.2 Driving UI from data models

Data models represent the data of our app. We should drive our UI from data models and preferably store them in persistent models, such as a local database. These models should be

independent of the UI and all other components in our app. This means that they are not tied to the Android application lifecycle but will nonetheless be destroyed when the OS decides to remove the app's process from the memory [29].

2.5.3 Single source of truth

"When a new data type is defined in your app, you should assign a Single Source of Truth (SSOT) to it. The SSOT is the owner of that data, and only the SSOT can modify or mutate it. To achieve this, the SSOT exposes the data using an immutable type, and to modify the data, the SSOT exposes functions or receive events that other types can call" [29]. Code listing 2.2 shows a simple counter that follows this principle. The private variable is mutable, and only the SSOTCounter class can change the value of the counter. The SSOTCounter class exposes the value of the counter through an immutable variable that cannot be changed outside of the SSOTCounter class.

Code listing 2.2 Example of simple counter that follows SSOT principle

```
class SSOTCounter {
    private val _counter = mutableStateOf(0)
    val counter: State<Int> = _counter

    fun increaseCounter() {
        /*  */
    }
}
```

2.5.4 Unidirectional Data Flow

Unidirectional Data Flow is one pattern often used along side the Single source of truth. *"In Android, state or data usually flow from the higher-scoped types of the hierarchy to the lower-scoped ones. Events are usually triggered from the lower-scoped types until they reach the SSOT for the corresponding data type"* [29].

Chapter 3
Design

This chapter focuses on design patterns, frameworks and technologies I used when refactoring.

3.1 Using Kotlin

This section will show many benefits of using Kotlin over Java.

Data class

In Java, it is common to use a DTO (data transfer object) class whose sole purpose is to transfer data. Even with just a few parameters, this class can become quite large. To solve this issue, the *Project Lombok* Java library provides a solution. With just a few simple annotations, Lombok can generate constructors, getters, setters, and more during compilation. This makes our code cleaner, and we can easily understand what the class is doing based on the annotations [31].

Kotlin solves this problem by providing a built-in class marked with *data*. This class automatically includes all members that are useful for a DTO class [32].

Examples of all three types of DTO classes are included in the attachment *files/DTOClasses*.

Null safety

In Kotlin, a reference to a value that can be null must be explicitly marked with *?* at the end. Conversely, a reference that is not marked with *?*, is guaranteed to not be null. However, this does not mean that Kotlin is immune to NullPointerException (later NPE). There are two main causes of NPE in Kotlin: explicitly throwing NPE or using *!!* (often called *bang bang*) non-null assertion operator. This operator converts any value to a non-null type and throws an exception if the value is null [33]. Using the non-null operator is often considered a code smell as it undermines Kotlin's null safety features.

While Kotlin does help developers think about null values and where they are possible, it is not a silver bullet that can completely prevent NPEs [34].

Extension functions

Kotlin provides ability to extend a class or an interface with new functions without need for inheritance. This allows us to extend functionality of *Views* or classes like *String* [35].

Default and named arguments

In Kotlin, it's possible to omit function parameters when they have default values. This default value can be set by appending = to the type. If a function has default values and we want to use them, we can simply exclude those arguments altogether.

Additionally, it's possible to name one or more arguments when calling a function. This is helpful when a function has many arguments and it's difficult to associate a value with an argument. It also allows us to freely change the order of arguments [36].

Combining default and named arguments can greatly improve the readability of Composable functions and reduce the amount of code required to build them. This can be observed in the 3.1 code listing, where the TextWithDefault function is called twice - once using the default value, and a second time using a named argument.

Code listing 3.1 Example of a Composable function with default and named arguments

```
@Composable
fun TextWithDefault(
    text: String = "Great text",
) { /* .... */}
@Composable
fun ComposableFunction() {
    TextWithDefault()
    TextWithDefault(text = "Not a great text")
}
```

Sealed class and when expression

A sealed class is a special type of class in Kotlin that provides more control over inheritance. By putting the *sealed* modifier before its name, all direct subclasses of a sealed class must be declared in the same package and are known at compile time.

The main advantage of using sealed classes is in the use of the *when* expression, which is a conditional expression with multiple branches. If all cases are covered in the statement, an else clause is not required. However, if the *when* statement is not exhaustive and all possibilities are not covered, Kotlin will not compile the code. For example, if a new Error type is added to the code listing 3.2 but is not included in the *when* statement, the code will not compile [37].

Code listing 3.2 Example of using sealed class

```
sealed class Error {
    class FileNotFound: Error()
    class NoPermission: Error()
    class NoInternet: Error()
}
/* ... */
fun handle(error: Error) {
    when(error) {
        is Error.FileNotFound -> print("FileNotFound")
        is Error.NoInternet -> print("NoInternet")
        is Error.NoPermission -> print("NoPermission")
    }
}
```

We could compare sealed classes to *Enum*. The set of values of an Enum is also known at compile time, but each Enum constant can have only a single instance, unlike subclasses of a sealed class, which can each have multiple instances [38].

Kotlin Multiplatform

Kotlin Multiplatform is a technology that can simplify cross-platform project development by reducing the time spent on writing and maintaining the same code for multiple platforms. One example is the ability to share business logic and connectivity code between Android and iOS applications [39].

3.2 Dependency injection

Dependency injection (DI) is a software design pattern that decouples a class from its dependencies. Instead of a class creating its own dependencies in constructors or other places where they are needed, dependencies are injected into the class, usually in the constructor. It is preferable to inject these dependencies as interfaces, which abstract the dependency and allow for easy component changes [40].

Koin

Koin is a Kotlin dependency injection framework with simple API. Koin extends the Android platform to provide specific features for Android development, such as injecting ViewModels [41].

3.3 High cohesion, Low coupling

High cohesion is a software design principle that aims to ensure that each class and method in a system has a single, well-defined responsibility. A class or method with high cohesion is focused on a specific task or set of related tasks, and its behaviour is closely aligned with that task or tasks. High cohesion is often associated with the *Single Responsibility Principle* (SRP), which states that a class or module should have only one responsibility.

By designing classes and methods with high cohesion, we can create more modular and reusable code. High cohesion helps us to reduce code duplication and to isolate changes to a specific part of the system, making it easier to maintain and test. When we follow the principle of high cohesion, our code becomes more focused, easier to understand, and less prone to errors.

Low coupling refers to the degree of dependency between classes in a software system. When classes are designed to be as independent from each other as possible, they can be modified without causing ripple effects in other modules. This is a significant advantage of low coupling in software design. On the other hand, high cohesion implies that every class and method should have a clear and specific purpose, resulting in better separation of logic into multiple files and more reusable code [42]. Achieving high cohesion and low coupling is possible with tools such as Koin, a dependency injection library, as well as UseCases and Repositories.

3.3.1 UseCase

A UseCase is a class with a singular purpose, which is to perform a specific task or operation. In particular, it is designed to manipulate data that is passed through it. It is important that a UseCase be stateless and free from mutable data. Additionally, it is recommended that a new instance is created each time UseCase is injected as a dependency.

For instance, consider a UseCase for extracting values from a JSON object. To implement this, we define an interface and an associated implementation. When we need to extract values from a JSON object, we can inject the implementation as an interface. This abstraction of an implementation not only allows us to easily add new implementations for extracting values from JSON but also to test classes that use this UseCase.

3.3.2 Repository

The repository acts as a SSOT of data for our application. It coordinates data from different data sources by resolving conflicts between them and exposing only the correct data. These sources can include local files, local databases, or a network connection. Similarly to UseCase, Repository is abstracted using an interface, which hides the source of data from the rest of the application [43].

3.4 Building UI with Compose

Building UI with Compose is completely different from building UI using XML and Views, as was the norm before. This is great news for someone who is trying to get into Android development, as even senior developers are only starting to learn how to use Compose to the fullest potential.

3.4.1 Creating layout

To create layouts in Compose, we have access to three different standard layout elements: *Column*, used for placing items vertically on the screen, *Row* for placing items horizontally, and *Box* for placing elements on top of each other. Combining these three layout elements is all we need to create layouts. The visual representation of these elements can be seen in Figure 3.1.

To set positions for children within Column, we can set *verticalArrangement* and *horizontalAlignment* arguments. For Row, it is *horizontalArrangement* and *verticalAlignment*. For Box, we have access to *contentAlignment* parameter [44].

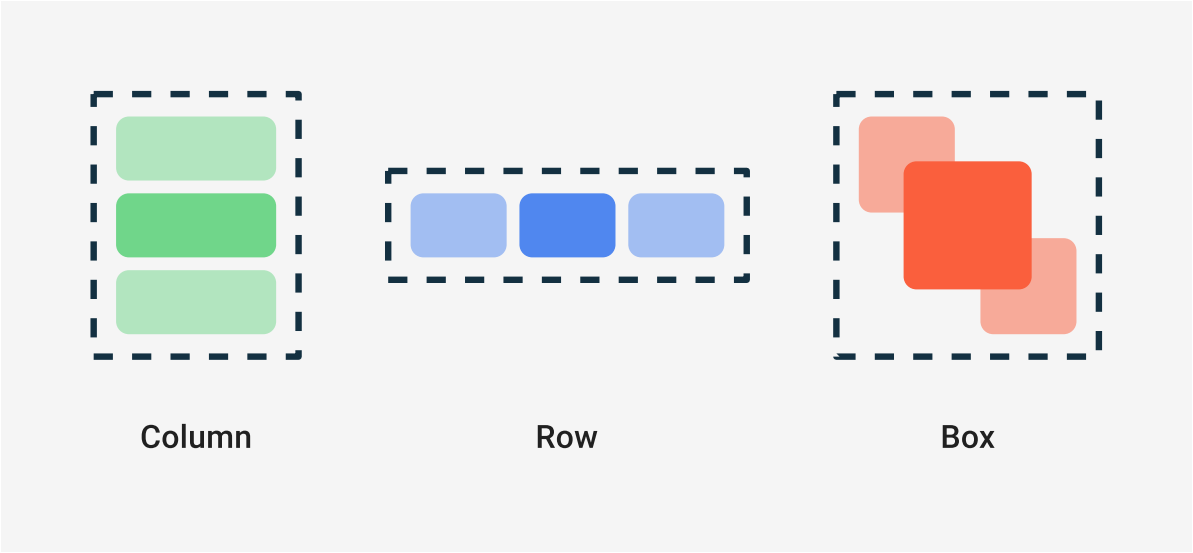

Figure 3.1 Graphical visualisation of standard layout elements [44]

3.4.2 Creating custom Composables

One of the primary goals of Compose is to make creating custom Views as simple as creating a new function. This is exactly how we create custom Composables in Compose: by creating a new Composable function. Since the entire Compose UI toolkit is open source, we can take a look at how developers at Google created each built-in Composable. If we want to create

something slightly different from the built-in Composable, we can copy the function and change a few parameters.

Before Compose, creating a custom View was something that only brave and senior developers would do. Now in Compose, creating a custom Composable is the second step in the Hello World tutorial [24].

3.4.3 Configuring layout and Composables

When we want to ensure that a layout meets our design requirements, we must use *modifiers* to decorate and enhance our Composables. These modifiers are essential for configuring our layout. When we add a modifier to a Composable, we create a chain of modifiers that are processed sequentially. Therefore, it is important to order the modifiers correctly to achieve the desired effect [44]. An example of a modifier in a Column can be seen in 3.3 code listing.

■ **Code listing 3.3** Example of using Modifier class to fill max possible width, have horizontal padding and test tag

```
Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(horizontal = 16.dp)
            .testTag("MyColumn"),
    )
```

3.5 App architecture patterns

2.5 App architecture section introduced best practices for creating high-quality applications. But it did not explain how to implement and use these practises together. For that we need to introduce architecture patterns. There are many different patterns, each with unique advantages and disadvantages.

3.5.1 MVC

The Model-View-Controller (MVC) is an architectural pattern that separates the code into three distinct components. The Model component is responsible for managing the data, including any business logic and communication with the data source (network or database). The View component is responsible for the user interface, usually an Activity or Fragment with ViewGroups. The View displays data from the Model and provides interaction with the user. The Controller component handles the communication between the View and the Model. It processes user input and manages the data accordingly [45].

One advantage of this pattern is that it is easy and fast to implement. However, a significant disadvantage is that the View is dependent on both the Controller and the Model. Any changes to the View require updates to several classes. Moreover, it often results in a bloated Controller with hundreds of lines of code, making it difficult to test and scale [46].

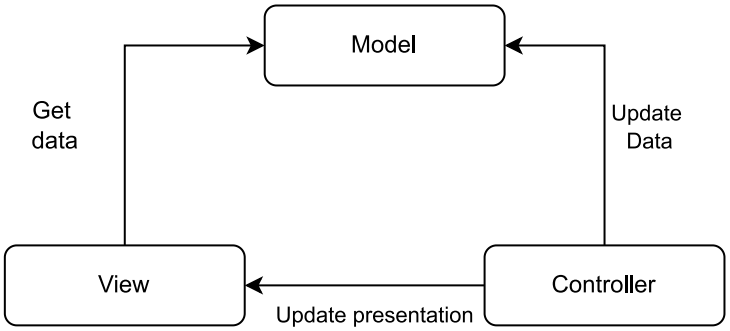

Figure 3.2 Flowchart illustrating MVC pattern [45]

3.5.2 MVVM

Model-View-ViewModel (MVVM) is an architectural pattern that separates code into 3 components. The Model and View components are similar to those in MVC, with the main difference being the ViewModel. Unlike the Controller in MVC, the ViewModel does not hold any reference to the View. Instead, the ViewModel updates the Model based on user interaction and exposes data streams containing data from the Model, which are then observed by the View.

The advantages of this pattern are that there are no references between the 3 components, making it easier to develop them in parallel and to scale the application. Since the ViewModel does not depend on the View or Model, it is easy to test. However, this pattern can be more complex than MVC and requires a deeper understanding of the system and the observable pattern. Additionally, it may be overkill for smaller projects [47].

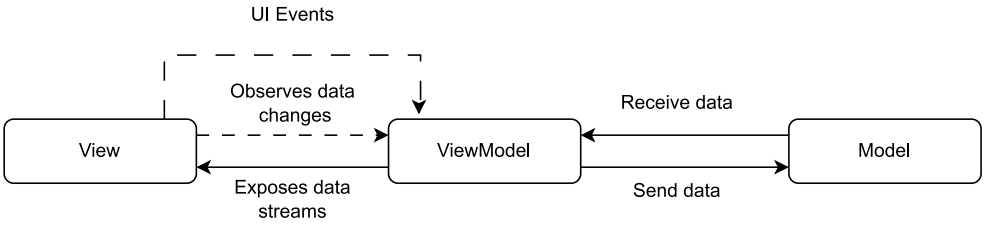

Figure 3.3 Flowchart illustrating MVVM pattern [48]

3.5.3 MVI

Model-View-Intent (MVI) is a new architectural pattern that is growing in popularity ever since it was introduced in Android. Unlike the MVC or MVVM patterns, MVI works on a unidirectional and circular flow. The model represents data but also the state of the UI as an immutable class. This ensures that the state can be changed only in one place, following SSOT principle. The intent represents an intention to perform an action. The advantages of this pattern are easy implementation of the SSOT principle, easy debugging thanks to the unidirectional data flow, having an immutable representation of the UI state, the ability to test all layers independently, and simple implementation in Compose. The disadvantage of this pattern is that it requires a lot of boilerplate code (code used in multiple places without much change) [49], [50], [51].

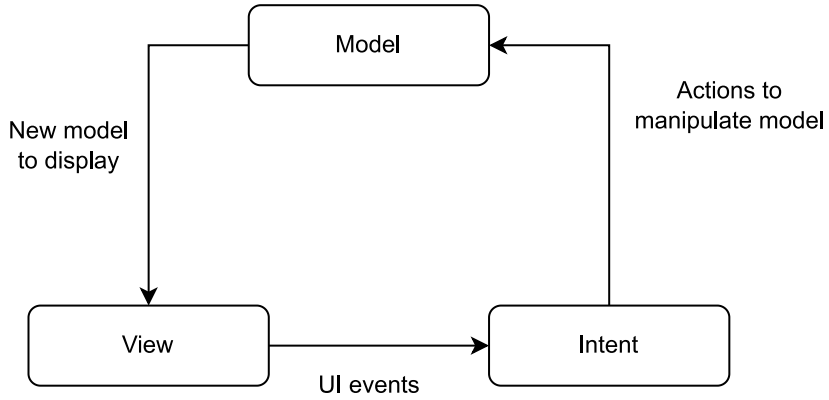

Figure 3.4 Flowchart illustrating MVI pattern [50]

Chapter 4
Analysis of the application

This chapter is focused on the analysis of the application.

4.1 Application parts

This section will provide a description of all the primary sections of the application with which the user interacts. Each part will be briefly discussed and evaluated to determine whether it is suitable for refactoring.

Map

The Map is the central component of the application and the primary interface with which users interact. It displays the locations of available cars, parking zones, and the user's location on the map. Users can filter the available cars and click on a specific car to view information about it. From there, they can reserve the car or access more detailed information.

Some Android Views are too complex to be converted to Compose. For instance, the *MapView* is one such View, and it is the primary view used in this section of the application. Moreover, this part of the application is quite intricate, containing many features and functionalities.

Vehicles

This section displays a list of all available vehicles in alphabetical order based on the name of the vehicle. Users can filter the cars based on the car model, transmission, vehicle type, or range, similar to the Map section. When a user clicks on a car, they are taken to the Map section, where they can see the details of the vehicle.

This part of the application is relatively small and is not expected to undergo many changes in terms of functionality.

Login

The Login section of the application allows the user to enter their email and password to log in. Both email and password fields provide real-time feedback to the user and display an error message if the input is invalid. The user can only click the *Log in* button if both TextFields are valid. If the user is not registered, they can click the *Register* button, which redirects them to the Uniqway web page for registration (this functionality was disabled during the writing process of this thesis). In case the user forgets their password, they can click the *Forgotten password*

button, which opens a new screen with an email TextField and a button to reset the password. Similar to the email and password fields, the reset password TextField also provides real-time feedback to the user and displays an error message if the input is invalid. Unlike the *Log in* button, the *Reset password* button is always enabled.

The Login section may be small, but it is a crucial component of the application since every user interacts with it at least once. In the future, the functionality of this part is likely to expand as registering on the web page is not ideal, and it would be preferable to have the option to register within the application itself.

Information

The Information section of the application offers users various types of useful information. The *Emergency situations* section provides a list of emergencies and guidance on how to handle them. *Infoline* offers a list of ways users can contact support. The *Vehicle equipment* section displays all the necessary equipment for a ride. The *Information* section lists all the social channels users can use to interact with the Uniqway community. *Contact points* contain all the necessary contact points for registration and other information. Finally, the *About* section displays the application version and a link to the Google Play store.

Choosing part for refactoring

As noted in the 1.2 *Refactoring* section, refactoring is a technique employed to reduce technical debt and facilitate smoother future development. Of all the aforementioned sections of the application, the *Login* section is the most likely to be expanded in the near future. As a result, it is preferable to refactor it. This chapter will now closely examine the Login section of the application.

4.2 Login section

Login section contains two different screens, *Login* and *Reset password*. The following sections will describe exactly their behaviour and how it was implemented.

Login

The login screen primarily validates both the email and password TextField and displays an error message for each TextField if the input is invalid. The *Log in* button is only enabled when there are no errors. For each email and password TextField, an instance of the *InputValidator* class is created. This class listens to changes made to the TextField. To determine whether the text is valid, a regex string is passed in the constructor during initialization of the InputValidator and this regex is later used to validate user input. If the text after a change is invalid, the InputValidator creates an error message and calls a callback that alerts the *LoginActivity* controller that the valid status has changed. In LoginActivity, this callback enables or disables the Log in button. As we can see, the ”InputValidator” class has many responsibilities and low cohesion. Its responsibilities should be separated into separate modules.

After the user presses the *Log in* button, the LoginActivity creates a request to the backend server and handles the response accordingly. If the log in attempt is successful, the Activity creates a log entry about the successful log in and then closes. If the log in attempt is not successful, the error message is passed into the *ErrorUtil* class, which creates a dialog showing the user what error has occurred based on the error message. If there was a failure during the request, the user probably has no internet connection, and a corresponding error dialog is displayed.

There is also a feature for users who wish to reset their password. When a user navigates to the ResetPassword screen, the value that was entered in the email TextField is automatically transferred to the ResetPassword TextField, making the process faster and more convenient.

If the user pressed *Forgotten password* button, he is transferred to the *Forgotten password* screen.

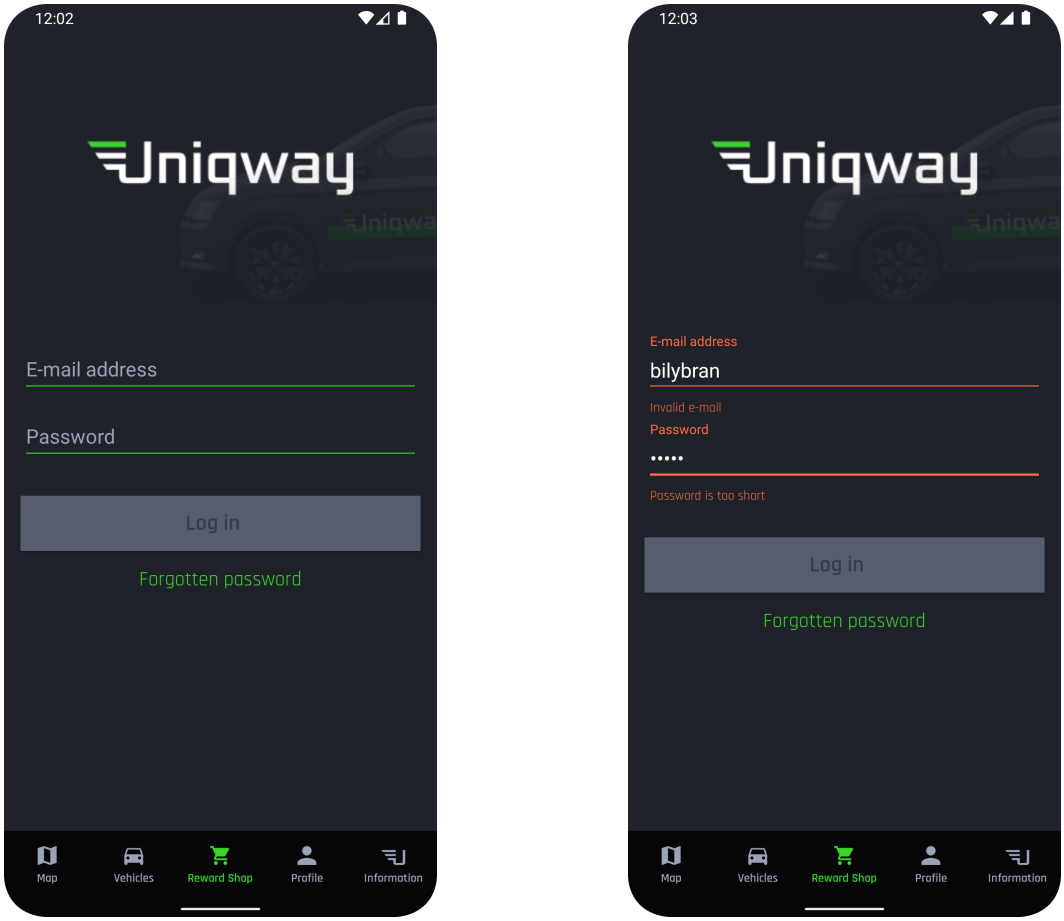

Figure 4.1 Login screen with no error messages and with both email and password error

Forgotten password

On this screen, the email TextField also displays an error message when the text input is not valid, but this error does not affect the ”Reset password” button, which is always enabled. To determine if the input is correct, the *ResetPasswordActivity* uses the same *InputValidator* class as in the Login screen.

After the user presses the *Reset password* button, the first step is to check if the email value is correct. If it is not, an error dialog is displayed. If the email is valid, the ResetPasswordActivity creates a request to the backend server. If there is a failure during the request, it is likely due to the user having no internet connection, and a corresponding error dialog is displayed. Otherwise, the user is shown a dialog indicating that their request has been processed. If the email provided is registered, they will receive an email with further instructions.

Lastly, Forgotten password screen contains an arrow back in the top left corner, allowing the user to navigate back into the Login screen.

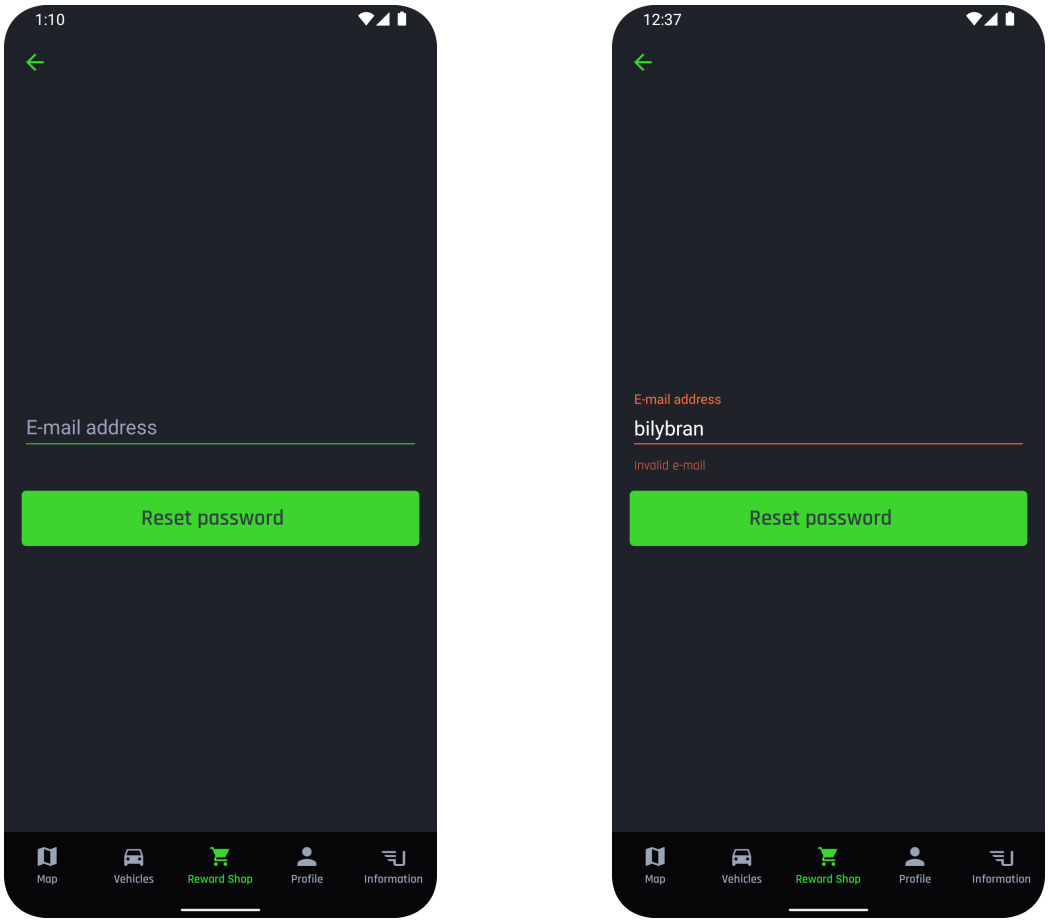

Figure 4.2 Password reset screen with no error message and then with email error

Chapter 5
Implementation

This chapter focuses on how to refactor an application from Android Views to Compose, all while following the principles described in previous chapters.

5.1 Strategies of refactoring

5.1.1 Java class to Kotlin

Migrating an app written in Java to Kotlin is a lengthy process that can take several years. For large projects with thousands of classes, it is unrealistic to rewrite everything while simultaneously implementing new features. Therefore, it is often recommended to migrate incrementally.

Kotlin classes are expected to have fewer lines of code than their Java counterparts. Meta, for example, reported a reduction of 11% in the number of lines of code after refactoring over 10 million lines of Kotlin code [52]. Popular HTTP client OkHttp, on the other hand, only reported a 7% reduction in lines of code [53].

When to migrate Java class to Kotlin

Given the size of the car sharing app project, I decided to write all new classes in Kotlin. Additionally, whenever I had to modify or read a Java class, I converted it to Kotlin. This approach ensured that frequently visited or modified classes are written in Kotlin [54].

How to migrate Java class to Kotlin

It is important to note that while the built-in tool for converting Java code to Kotlin in Android Studio can save time, it is not foolproof and may lead to issues in the resulting Kotlin code. Therefore, it is recommended to review the converted code and make necessary manual changes to ensure that the code is correct and readable. One common issue that may arise from the conversion tool is the usage of the non-null assertion operator, which can lead to potential NPEs at runtime. It is important to avoid using this operator and instead handle null values appropriately to ensure the stability of the application [55], [52].

5.1.2 Migrating from Android Views to Compose

To migrate an entire application to Compose in one go, just like migrating from Java to Kotlin, is an unrealistic approach. Therefore, Google recommends a similar approach for Compose migration as with Java to Kotlin. First, we should start by implementing new features with

Compose, and then gradually replace existing features one screen at a time. Luckily, we can mix Android Views and Compose, allowing us to migrate to Compose one UI component at a time [56].

5.1.3 Migrating to a new app architecture

The Login and Password Reset Activities in the application use the MVC architecture. The transition from MVC to the MVI architecture, which is often recommended with Compose, is quite significant. Therefore, I decided to first migrate both Activities to Kotlin and the MVVM architecture. This additional step allowed me to run tests sooner to ensure that the behaviour of both Activities remained the same. Furthermore, since testing in Compose is slightly different, this step was also done without using Compose. After completing this migration, I started using Compose and modified the architecture to be a *hybrid* of MVVM and MVI, taking advantage of both approaches. ViewModel and its Lifecycle from MVVM and immutable state class from MVI.

5.1.4 Separating concerns

Refactoring is a great opportunity to separate concerns in the app into separate units with a single purpose. A great example would be the *InputValidator* class, which handled listening to changes in the TextField, validating input, and changing the error state. During the migration from MVC to MVVM, this class can be nicely separated into standalone units, greatly improving the flexibility of our code.

5.2 Adding Kotlin to Java Android app

This section will show all steps required to add Kotlin support to Java Android app and how to convert Java into Kotlin.

Adding support for Kotlin

To add support for Kotlin, we will need to update both project and app *build.gradle* files. What to add is shown in *files/JavaToKotlin* folder.

Converting Java class to Kotlin

Once Kotlin support has been added, we can begin converting Java classes to Kotlin. To do this, we open the Java class and use the built-in converter. However, because the converter does not fully understand the code, we will likely need to perform some manual optimizations on the resulting Kotlin class.

Optimizing Kotlin class

The most common optimization required is how to handle null values. The converter creates a nullable property that can never be null but cannot be initialized where it is defined, often with View references. This can be solved by adding the *lateinit* modifier to the variable. Sometimes we know from the system's knowledge that the variable cannot be null, so we can remove nullability.

Since the converter does not understand the code, it creates a non-null assertion operator (!!) wherever an NPE was possible, which is not ideal and is even considered a code smell. Using the safe-call operator (?.) alongside the elvis operator (?:) is much more preferred.

5.3 From MVC to MVVM app architecture

This section will show all steps required to add ViewModel to an Activity. Code after each important step is included in *files/MVC-MVVM* folder.

5.3.1 Making sure ViewModel works

First, we have to make sure that everything is set up correctly. No functionality will be transferred to ViewModel yet.

Adding required dependencies and settings – First, we will have to modify app level *build.gradle* file. We have to enable *dataBinding* and apply Kotlin annotation processor.

Create ViewModel – We create a new ViewModel class extending class *ViewModel* with one String property, that will be used to verify that everything works correctly.

Adding ViewModel variable to layout – Next step is to add avariable to our layout, that will hold reference to our ViewModel and create a simple TextView that will show the String we defined in our ViewModel in the previous step.

Binding ViewModel to variable in layout – This step is most important and can sometimes be forgotten. We have to bind our ViewModel instance to the ViewModel variable in the layout. This can be simply achieved using an extension function on our Activity and calling it in *onCreate* function.

5.3.2 Migrating functionality to ViewModel

Now that we are assured ViewModel is working correctly, we can start to migrate functionality from Activity to ViewModel.

Handling user interaction

Now we want to handle user interactions, such as pressing a button or inputting text. These interactions are handled differently.

To handle **text input**, we can use a property of type *MutableLiveData*, which is an observable property that can be changed. In the previous section, we used a concept called *one-way binding*, which means that the binding goes only one way, from the observable to the View. However, since we want to update our variable based on user input, we need to use *two-way binding*. This binding goes both ways, so the observable or the View can modify the value, and the new value will be propagated to the other.

We can handle **actions** by simply calling a function from ViewModel.

Handling events

To handle events, we can expose LiveData representing our event. For each event, we will have one sealed class and its descendants will represent all possible outcomes of this event. In the Activity, we can observe the LiveData representing our event and use *when* statement to handle all possible outcomes.

Using business logic

We aim to utilize business logic in the ViewModel instead of the Activity. Assuming we followed the Separation of Concerns principle, we have already established access to UseCases and Repositories with our data and business logic. By utilizing any DI library, we can inject them into our ViewModel and subsequently inject our ViewModel into the Activity. For the purposes of this thesis, we will assume that DI is operational. This approach allows us to use business logic and reveal its result using sealed classes.

Handling screen state

To handle the screen state, we can use a special type of LiveData called *MediatorLiveData*. This type of LiveData allows us to observe other LiveData variables and change the screen state based on their values. We can then use the MediatorLiveData value to update the screen state.

By following these steps, we can gradually remove the logic from the Activity and migrate it to the ViewModel.

5.4 Migrating from Views to Compose

This section will show all the steps required to migrate one Activity from Views to Compose.

5.4.1 Making sure Compose works

As with ViewModel, we will first make sure that Compose is working correctly. No Views will be migrated to Compose yet.

Adding required dependencies

The easiest way to add all the required dependencies is to let Android Studio handle it. To create a new empty Compose Activity, go to *File -> New -> Compose -> Empty Compose Activity* and follow the prompts. Android Studio will automatically add all the required dependencies. Additionally, we will need to enable viewBinding in the application's build.gradle file.

Adding Compose to our layout

When working with Compose, we can easily add a *ComposeView* into our layout and specify its ID. This is possible because Compose was designed with view interoperability in mind from the start. As a result, we no longer need to bind ViewModel variables to the layout. Instead, we can remove the setupDataBinding method and create a ViewBinding object. We can then use this object to find our ComposeView and set its content to a Composable.

Creating first Composable

We create a new Composable by creating an empty Kotlin file and adding a new Composable function. We can add Text Composable to make sure that everything is working.

5.4.2 Migrating functionality to Compose

To begin, we create a data class in our ViewModel that represents the state of our UI. This example includes two variables: the text entered by the user and a boolean value indicating whether the Button is enabled. We expose this state through an immutable variable. We then

use this state variable to modify Composables and recompose them whenever their arguments change.

Handling user interaction

Every Composable that a user can interact with provides a callback function that is called every time the user interacts with it. We can simply call a function from our ViewModel in these callbacks.

Handling screen state

Handling screen state in Compose is simpler than in traditional Android Views. Since Composables are just Kotlin functions, we can use conditions or loops to show or hide different parts of the screen, depending on the state of the UI. This allows us to create more dynamic and responsive UI with less boilerplate code.

Separating ViewModel from Composables

Composable functions in Compose allow us to create reusable UI components that can be combined to build a complete UI. The *Preview* feature in Compose allows developers to visualize how these components will look and behave on different devices and configurations without having to build and deploy the app on a device or emulator.

When creating a Composable function that uses a ViewModel, it's recommended to create a second Composable function that doesn't depend on the ViewModel but only contains the necessary variables and callbacks. This enables us to create previews of the Composable without simulating the ViewModel. To create a preview, we can annotate the Composable function with the *Preview* annotation and then use the *PreviewParameter* annotation to specify the input parameters to use in the preview. This approach can significantly speed up the development process by allowing us to iterate and test our UI components quickly without having to perform a complete build and deployment cycle.

5.5 Login screen functionality implementation

This section will focus on how each functionality in the Login screen was implemented using Compose and the MVI app architecture.

5.5.1 TextField error

Every time the value of a TextField changes, the corresponding callback is called in the ViewModel. This callback updates the view state with the new TextField value and checks if the new value is valid. After this, our view state is updated with new values depending on the user input, which we can use to change our screen. Code Listing 5.1 shows how this could be implemented. *EmailTextInput* is a custom Composable that, when the *error* Boolean value is true, shows the *errorText* error message. In the attachment *Implementation/EmailTextFieldError.txt*, there is a simplification of the previous implementation.

Importantly, this callback does not directly change any components but rather updates the view state with new values. Every Composable function that was changed with the new values in the view state undergoes a recomposition and updates itself accordingly, as mentioned in the 2.4.2 section on the Lifecycle of a Composable.

Code listing 5.1 Code snippet of how showing error in a TextField could be implemented

```
//Function in ViewModel
fun emailChanged(newValue: String) {
    val email = newValue
    viewState.update { it.copy(email = email) }
    if (validateEmail(email)) {
        viewState.update { it.copy(emailError = false) }
    } else {
        viewState.update { it.copy(emailError = true) }
    }
}

...

//Composable function
EmailTextInput(
    labelText = stringResource(id = R.string.hint_email),
    error = viewState.value.emailError,
    errorText = stringResource(id = R.string.invalid_mail),
    onValueChanged = viewModel::emailChanged,
)
```

5.5.2 Enabling Log in button

The Login button depends on the values of the TextFields, so after any one of them is updated, we have to check if the login button should be enabled. We can use the callback shown in the previous subsection. Our *Log in* button will then enable or disable itself depending on the view state. Code Listing 5.2 shows how this could be implemented. The previous implementation can be found in the attachment *Implementation/LogInButtonEnabled.txt*.

Code listing 5.2 Code snippet of how showing enabling Button could be implemented

```
//Function in ViewModel
fun checkLoginEnabled() {
    val email = viewState.email
    val password = viewState.password
    //Button is disabled when there is an error or email/password is empty
    if (viewState.emailError || viewState.passwordError ||
        email.isEmpty() || password.isEmpty()
    ) {
        viewState.update { it.copy(buttonEnabled = false) }
    } else {
        viewState.update { it.copy(buttonEnabled = true) }
    }
}

...

//Composable function
UniqwayLoginButton(
    onLoginClicked = viewModel::onLogin,
    buttonEnabled = viewState.buttonEnabled,
)
```

5.5.3 Showing Progress bar

The view state will have one Boolean value indicating whether the progress bar should be visible or not. We can change this value to true before every asynchronous call and to false after every response. In Compose, we can check if the value is true and then show the progress bar. Code Listing 5.3 shows an example of a possible implementation. The previous implementation of the progress bar before refactoring is shown in the attachment *Implementation/ProgressBar.txt*.

Code listing 5.3 Code snippet of how Progress bar could be implemented

```
//Function in ViewModel
fun login() {
    val email = _viewState.value.email
    val password = _viewState.value.password
    viewState.update { it.copy(isLoading = true) }
    val loginCall = restApi.logIn(email, password)
    //Asynchronous call trying to log in the user
    loginCall.enqueue(object : Callback<LoggedUser> {
        override fun onResponse(call: Call<User>,response: Response<User>) {
            viewState.update { it.copy(isLoading = false) }
            // Handle response
        }

        override fun onFailure(call: Call<LoggedUser?>, t: Throwable) {
            viewState.update { it.copy(isLoading = false) }
            // Handle failure
        }
    })
}

...

//Inside a Composable function
Box {
    ViewLayout() {
        ...
    }
    if (viewState.isLoading) {
        ProgressBar()
    }
}
```

5.5.4 Showing Dialog

Similar to the progress bar, we can show a dialog to the user depending on the view state. By creating a class that has all the values required to show the correct dialog, we can save this value in the view state as a *nullable* value. When the value is *null*, no dialog should be shown. But when the value is not null, it is used to show the correct dialog. Code Listing 5.4 shows an example of a possible implementation. The DialogResource data class has only two values, an identification for the text in the title and the body of the dialog. By passing only the identification, we can use Android resources mentioned in section 2.1.5 to show translated text to the user. Unfortunately, the previous implementation was quite complex and could not be simplified for the purposes of this thesis.

Code listing 5.4 Code snippet showing how Dialog could be implemented

```
//Functions in ViewModel
private fun handleLoginUnsuccessful(response: Response<User>) {
    response.errorBody()?.let {
        val errorDialogResource = responseBodyToDialogResource.parse(it)
        showDialogWithError(errorDialogResource)
    }
}

private fun showDialogWithError(dialogResource: DialogResource?) {
    viewState.update { it.copy(dialogError = dialogResource) }
}

fun dismissDialog() {
    viewState.update { it.copy(dialogError = null) }
}

...

data class DialogResource(
    val headerRes: Int,
    val messageRes: Int
)

...

//In Composable function
val dialogError = viewState.value.dialogError
if (dialogError != null) {
    ErrorDialog(
        onDialogClicked = viewModel::dismissDialog,
        titleRes = dialogError.headerRes,
        textRes = dialogError.messageRes,
    )
}
```

5.6 Creating documentation

To create documentation for the code that I wrote, I decided to use *KDoc* comments for each public method and then generate documentation using the documentation engine *Dokka*. This engine understands KDoc and Javadoc comments and generates documentation in many formats, including HTML [57]. The documentation is included in the attachment in folder *dokka*.

Chapter 6
Testing

This chapter focuses on different types of tests in Android and various techniques for testing.

6.1 Importance of testing

Testing our app is an integral part of the development process. It helps to validate that the application works as intended, reduces the risk of bugs in the released product, and reduces stress during release. Testing also offers the advantage of *early failure detection* [58]. The earlier a bug is caught, the more debugging information is available about its origin. Testing our app also makes refactoring faster and safer, allowing developers to focus on optimizing code and minimizing technical debt without worrying about regressions.

Edsger W. Dijkstra, renowned for his contributions to the shortest path algorithm known as Dijkstra's algorithm, once said, *"Program testing can be used to show the presence of bugs, but never to show their absence!"* [59] This statement can be interpreted in various ways, such as the idea that achieving high test code coverage (the percentage of lines of code executed at least once during testing) does not necessarily guarantee a bug-free application.

6.2 Different types of tests

Developing software is a complex undertaking since applications must run on various systems under different loads, making testing them also challenging. As a result, there are various types of tests, depending on the aspects of the application being tested. This thesis will concentrate on *functional* tests and categorize them based on their *degree of isolation* [60].

6.2.1 Unit tests

The foundation of every project should be *unit* tests. Unit tests ensure that each unit of our codebase works as intended. These tests have the narrowest scope of all tests and should largely outnumber any other type of test since they are very fast [61]. While they increase confidence in a single class, they do not verify interactions between multiple classes.

6.2.2 Integration tests

Integration tests focus on testing the interaction between classes or subsystems, ensuring that they work together as intended. These tests often use *test doubles*, to isolate the subsystem being

tested from external dependencies. While integration tests are generally faster than end-to-end tests, they require more setup and may be more complex to write than unit tests.

Test doubles

During testing, it is important to test every element in isolation. However, in some cases, the test subject may depend on other components to function properly. In such cases, it is a common practice to create a test double to provide the necessary behaviour or data. Test doubles are objects that mimic the behaviour of real components in our app, but are created specifically for testing purposes. They offer several advantages, such as making tests faster and simpler, as well as providing a way to test components in isolation [62], [63].

There are several different kinds of tests doubles. These kinds have conflicting definitions depending on the source.

Table 6.1 Different kinds of test doubles [63]

Fake	A test double that has a "working" implementation of the class, but it is implemented in a way that makes it good for tests but unsuitable for production. For example an in-memory database.
Mock	A test double that behaves how we program it to behave and that has expectations about its interactions. Mocks will fail tests if their interactions don't match the requirements that you define. Mocks are usually created with a mocking framework to achieve all this.
Stub	A test double that behaves how we program it to behave but doesn't have expectations about its interactions. Usually created with a mocking framework. Fakes are preferred over stubs for simplicity.
Dummy	A test double that is passed around but not used, such as if we just need to provide it as a parameter. For example, an empty function passed as a click callback.
Spy	A wrapper over a real object which also keeps track of some additional information, similar to mocks. They are usually avoided for adding complexity. Fakes or mocks are therefore preferred over spies.

In this project, Mock testing double was used. I added tests using Fake and Dummy testing doubles.

6.2.3 End-to-end tests

End-to-end tests, also known as Espresso tests in Android (named after the Espresso testing framework), simulate user interactions with Views. These tests can range from testing the behaviour of a single component to using a large navigation test that covers an entire user flow. Because these tests run on a real or emulated device, they take a long time to complete [64].

There are two ways to create end-to-end tests. The first way is through record-and-playback tools where the testing framework records user interactions and generates a test script based on those interactions. Although this method is fast, it can create brittle tests that break easily with even small changes to the system. The second method is to manually write the end-to-end tests, which can be time-consuming. However, this approach results in more reliable tests that are less likely to break due to changes in the system [65].

Flakiness

Flaky tests can pass or fail despite making no changes to code or test itself [66]. The culprit is the asynchronous nature of mobile applications. Sometimes, asynchronous calls like loading data or even showing infinite animations can make our test fail [64].

6.3 Test pyramid and ice-cream cone

Both the test pyramid and ice-cream cone are metaphors that provide us with an idea of how many tests we should have at each level of isolation. The test pyramid is a visual representation of how our applications should be tested, although, from a modern point of view, the original naming of the layers is overly simplistic. With Compose, for example, we can test UI with Unit tests, breaking this metaphor. Nonetheless, the test pyramid illustrates two important lessons: to write tests with different scopes and that the more high-level we get, the fewer tests we should have.

If we do not follow these principles, we can end up with what is called the ice-cream cone. This is the test pyramid flipped on its head, with a few unit tests on the bottom, a few service tests, a lot of UI tests, and even more manual UI tests at the top [61].

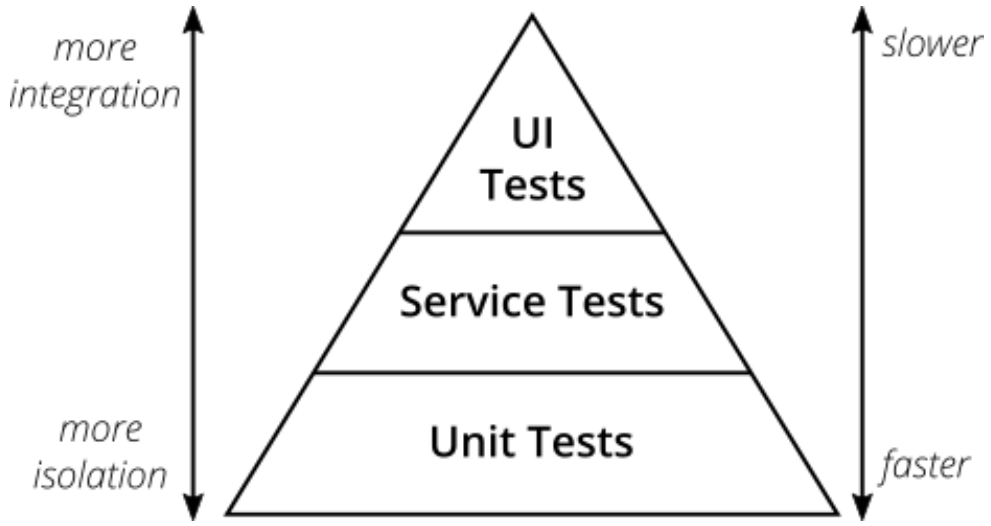

Figure 6.1 The test pyramid [67]

Test ratio

To follow the test pyramid metaphor, the test ratio indicates how many tests should be present in each layer. The recommended guideline for Android applications is $7 : 2 : 1$, meaning that out of 10 tests, 7 should be unit tests, 2 integration tests, and one end-to-end test. However, it is essential to keep in mind that the exact ratio may vary depending on the application and its requirements. Donn Felker suggested a $6 : 4$ ratio (6 unit and integrated tests and 4 end-to-end), which may work better for some applications, particularly those with complex user flows and interactions.

It's also important to note that while end-to-end tests are essential for verifying the behaviour of the app from a user's perspective, they can be slower and more difficult to maintain than unit and integration tests. Therefore, finding the right balance between the different types of

tests is key to ensuring high-quality software that meets both user needs and development team requirements. [68].

6.4 Black box vs. White box testing

Tests could be also separated into two boxes, depending on whether we know, how the test subject is implemented or not [69].

6.4.1 White box testing

When we test a subject with a known implementation, we can write tests to cover all possible outcomes. However, this approach can make the tests *brittle*, meaning that even minor changes to the implementation can break the tests [69].

6.4.2 Black box testing

When testing a subject without considering its implementation, we are testing against the interface. In this approach, we create tests by grouping inputs into equivalent classes that expect the same return value and then testing the edge cases between these equivalent classes. This makes the tests more robust, as changes to the implementation will have no effect on the tests [69].

6.5 Android tests running environment

For most Android developers, the most important aspect to consider in testing is where the tests are executed. *Instrumented* tests run on an Android device, either a physical device or an emulator. The app is built and installed alongside a test app that injects commands and reads the state. Instrumented tests usually involve UI testing, launching an app, and then interacting with it. While unit tests and integration tests can also run on the device, it is less common. The downside of instrumented tests is that they can be slow compared to *local tests*. Local tests execute on our development machine or a server, so they are also called host-side tests. These tests are usually small and fast, running on the JVM and isolating the subject under test from the rest of the app [60].

6.6 UI tests in Compose

UI tests are crucial for verifying the behaviour of our Compose code. They not only include end-to-end tests that simulate user interactions with our app but also Unit tests of our Composables. When working with a hybrid app that contains both Compose components and traditional view hierarchies, we can use both the Compose testing API and the Espresso API in the same test to ensure complete coverage of our app's behaviour.

Creating end-to-end tests in Compose differs from testing a View-based UI, as Compose offers a new set of testing APIs for finding elements, verifying their state, and performing user actions. In Compose UI testing, *semantics* are used to interact with the UI hierarchy. Semantics provide meaning to a given UI element and are generated into a *semantics tree* alongside the UI hierarchy. The semantics tree gives meaning to the UI hierarchy and is mostly used for accessibility, although UI tests can also take advantage of the information exposed by semantics, which includes content descriptions or text.

The testing API in Compose provides three ways to interact with the UI hierarchy. First, *finders* are used to select one or multiple elements (known as nodes in the semantics tree) based on their text, content description, testing tag, and more. After selecting an element, *assertions*

are used to verify that the element exists and has specific attributes. Finally, when interacting with the UI is required, *actions* are used to inject simulated user events on the elements, such as clicks, text input, or other gestures [70].

Finders

The *ComposeTestRule* class provides the *onNode* and *onAllNodes* functions to select one or multiple nodes respectively, but there are also a couple of convenience functions called *Finders* that are used within these functions to specify how to search for nodes in the Semantics tree. The most common searches are provided as convenience functions, such as onNodeWithText, onNodeWithContentDescription, and so on [70].

Assertions

We can call the *assert* function on the node returned by the finder with one or multiple matchers. In addition, there are also several convenience functions available for the most common assertions, such as assertExists, assertIsDisplayed, assertTextEquals, and more [70].

Hierarchical matchers

Hierarchical matchers can go up or down the semantics tree and perform simple matching. Functions include matching parent, siblings, ancestors and descendants [70].

6.7 Tests added to the application

Every UseCase that was added was tested using integration tests, except for one UseCase where providing a testing double for the API response proved to be too difficult. Both the Email and Password Validators were tested using unit tests, utilizing a black box method (testing edge cases of equivalent classes). Compose components that contain behaviour were tested using unit tests as well. Two end-to-end tests were created to test the functionality of sending an email from the Login to ResetPassword TextFields. Lastly, the end-to-end tests for both the Login and ResetPassword screens were first rewritten from Espresso to the Compose testing API. Then, several new tests were added to test the initial state of the screens or test different combinations of TextField values.

6.8 Testing scenarios

This section will focus on testing scenarios of tests included in the *files/Testing* folder.

Unit test email validator

1. Validator returns true on correct email
2. Validator returns true on correct email with different domain
3. Validator returns false when trying to validate null value
4. Validator returns false when email is missing username
5. Validator returns false when email is missing domain

Compose button Unit test

1. First create Button that is enabled and when it is clicked, it changes a boolean value
2. First check if the Button exists
3. Check if it has correct text
4. Then check if is it displayed
5. Click the button and check, if the *onLogin* function was called

Error to dialog Integration test

1. Before all tests start, initialize UseCase variable with test double dependency
2. Initialize Error variable with unknown code
3. Check, if UseCase returned null
4. Change value of variable to another unknown code
5. Check again, if UseCase returned null

Login end-to-end test

1. Input into e-mail address field incorrect e-mail
2. Assert that error text field exists and has correct error message
3. Input into password field incorrect password
4. Assert that error two text fields exist and are showing correct error messages
5. Input into e-mail address field correct email
6. Assert that error text field exists and has correct error message
7. Input into password field correct password
8. Assert that no error text fields exist

Conclusion

The goal of this thesis was to modernize the codebase of an Android application. This included rewriting Java code in Kotlin, changing the app architecture, following recommended design patterns, and using Compose instead of Views.

Initially, support for Kotlin was added to the project. However, this led to a problem with all classes that used the aforementioned *Project Lombok*. These classes had to be rewritten in Kotlin before any other changes could be made to the project. Afterwards, both the Login and Reset Password Activities were rewritten to follow the MVVM architectural pattern and then rewritten from Java to Kotlin. During this change of architectural pattern, all data and business logic were separated from the UI, following the design patterns explained in chapters 2 (Android platform specifics) and 3 (Design), and later injected into the ViewModel using dependency injection. Once the Login and Reset Password Activities were using the MVVM pattern and were written in Kotlin, the change to Compose and MVI pattern was much faster and simpler. Step by step, the Views were removed from the Activity and transferred into Compose, allowing for manual testing of the functionality. After every View was transferred to Compose, I began to rewrite the previous end-to-end tests using Compose testing API and even added some additional tests that I found were missing. I also used unit tests to test Composables, Validators (classes used for validating email and password), and even UseCases, which contain business logic. To ensure that the UseCases were working correctly together, I also added integration tests using fake testing doubles.

Although the number of lines of code in the project only decreased by 1% after transitioning to Kotlin, this does not necessarily indicate that the transition was unnecessary. The project underwent significant changes in its architecture, with business logic and data being separated into their own modules and the adoption of new design patterns. These changes led to the creation of 22 additional files, which contributes to the minimal decrease in the number of lines of code. However, in the DTO classes, there was a significant reduction of 55% in the number of lines of code from Java to Kotlin. Therefore, the number of lines of code may not be the best metric to evaluate the effectiveness of the Kotlin transition. In terms of metrics that are hard to measure, both Login and ResetPassword screens are simpler, since they do not contain any business logic. The MVI pattern and Compose make it easier to debug the screen state and add new functionality. By using ViewModels, DI, and UseCases, changes made to one module won't affect other modules. This makes it faster and more robust to make changes. Even if a new bug is introduced, there are tests at every level of isolation, increasing the probability that the bug won't go unnoticed. Lastly, every public method and class is documented, making it easier to read the code.

There is still a lot of work to be done on the application, as 83% of the code is still written in Java. While it is unrealistic to bring this number down to zero, there are parts of the application that could be refactored into Kotlin and Compose. For instance, the *Information* section could be

converted to Compose, and once all the business logic is converted to Kotlin, it could potentially be used in an iOS application using *Kotlin Multiplatform*.

Bibliography

1. FOWLER, Martin. *Technical debt* [online]. 2019. Available also from: https://martinfowler.com/bliki/TechnicalDebt.html. Accessed on 2023-03-25.
2. *Oracle Database* [online]. 2018. Available also from: https://news.ycombinator.com/item?id=18442941. Accessed on 2023-03-25.
3. *6 technical debt examples and how to solve them* [online]. 2022. Available also from: https://www.techtarget.com/searchitoperations/tip/6-technical-debt-examples-and-how-to-solve-them. Accessed on 2023-03-25.
4. MARTIN, Robert Cecil. Clean Code. In: *Clean code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009, pp. 14–14. ISBN 978-0132350884.
5. FOWLER, Martin. *DefinitionOfRefactoring* [online]. 2004. Available also from: https://martinfowler.com/bliki/DefinitionOfRefactoring.html. Accessed on 2023-03-25.
6. MATTA, Peter. *Refactoring and object-oriented design patterns* [online]. 2022. Available also from: https://docs.google.com/presentation/d/1KOHLt2OHRmxUx0TQ4k41-dVhaLGZ8uTClWT67iDdXyk/edit#slide=id.g615494dbd9_0_0. Accessed on 2023-03-26.
7. FOWLER, Martin. *CodeSmell* [online]. 2006. Available also from: https://martinfowler.com/bliki/CodeSmell.html. Accessed on 2023-03-25.
8. CHEBBI, Ajay. *Choosing the best programming language for mobile app development* [online]. 2021. Available also from: https://developer.ibm.com/articles/choosing-the-best-programming-language-for-mobile-app-development/. Accessed on 2023-03-25.
9. LARDINOIS, Frederic. *Google makes Kotlin a first-class language for writing Android apps* [online]. 2017. Available also from: https://techcrunch.com/2017/05/17/google-makes-kotlin-a-first-class-language-for-writing-android-apps/. Accessed on 2023-03-25.
10. LARDINOIS, Frederic. *Kotlin is now Google's preferred language for Android app development* [online]. 2019. Available also from: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/. Accessed on 2023-03-25.
11. WILSON, James Q.; KELLING, George L. *Broken windows* [online]. 1982. Available also from: https://www.theatlantic.com/magazine/archive/1982/03/broken-windows/304465/. Accessed on 2023-03-26.
12. ROUSE, Margaret. *What is Android platform?* [Online]. 2011. Available also from: https://www.techopedia.com/definition/4219/android-platform. Accessed on 2023-03-27.

13. IONESCU, Daniel. *Original Android Prototype Revealed During Google, Oracle Trial* [online]. 2017. Available also from: https://www.pcworld.com/article/464050/original_android_prototype_revealed_during_google_oracle_trial.html. Accessed on 2023-03-27.
14. STATCOUNTER. *Mobile Operating System Market Share Worldwide* [online]. 2023. Available also from: https://gs.statcounter.com/os-market-share/mobile/worldwide/#quarterly-202301-202301-bar. Accessed on 2023-03-27.
15. PHAM, Dat; ALEXA, Marek. *Recommended minimum SDK version for Android projects* [online]. 2023. Available also from: https://www.megumethod.com/blog/recommended-minimum-sdk-version-for-android-projects. Accessed on 2023-03-27.
16. *Activity* [online]. 2023. Available also from: https://developer.android.com/reference/android/app/Activity. Accessed on 2023-03-27.
17. *Fragment* [online]. 2023. Available also from: https://developer.android.com/reference/android/app/Fragment. Accessed on 2023-03-27.
18. *View* [online]. 2023. Available also from: https://developer.android.com/reference/android/view/View. Accessed on 2023-03-27.
19. *Build a responsive UI with ConstraintLayout* [online]. 2023. Available also from: https://developer.android.com/develop/ui/views/layout/constraint-layout. Accessed on 2023-03-27.
20. *App resources overview* [online]. 2022. Available also from: https://developer.android.com/guide/topics/resources/providing-resources. Accessed on 2023-03-27.
21. *LiveData* [online]. [N.d.]. Available also from: https://developer.android.com/reference/android/arch/lifecycle/LiveData. Accessed on 2023-04-13.
22. *Jetpack Compose UI App Development Kit* [online]. [N.d.]. Available also from: https://developer.android.com/jetpack/compose. Accessed on 2023-03-27.
23. *Compose Multiplatform UI Framework* [online]. 2023. Available also from: https://www.jetbrains.com/lp/compose-multiplatform/. Accessed on 2023-04-13.
24. FELKER, Donn; GOPAL, Kaushik; RICHARDSON, Leland. *171: Jetpack compose with Leland Richardson* [online]. 2019. Available also from: https://open.spotify.com/episode/2OGsY6iuKx6ZL5EKLxJyVn. Accessed on 2023-03-27.
25. *Kotlin and Android* [online]. [N.d.]. Available also from: https://developer.android.com/kotlin. Accessed on 2023-03-27.
26. *Gradle User Manual* [online]. 2022. Available also from: https://docs.gradle.org/current/userguide/userguide.html. Accessed on 2023-03-29.
27. *The activity lifecycle* [online]. 2023. Available also from: https://developer.android.com/guide/components/activities/activity-lifecycle. Accessed on 2023-03-29.
28. *Lifecycle of composables* [online]. 2023. Available also from: https://developer.android.com/jetpack/compose/lifecycle. Accessed on 2023-03-29.
29. *Guide to app architecture* [online]. 2023. Available also from: https://developer.android.com/topic/architecture. Accessed on 2023-03-28.
30. *Separation of Concerns* [online]. 2023. Available also from: https://help.sap.com/doc/abapdocu_753_index_htm/7.53/en-US/abenseperation_concerns_guidl.htm. Accessed on 2023-03-28.
31. *Project Lombok* [online]. 2023. Available also from: https://projectlombok.org. Accessed on 2023-03-30.
32. *Data classes* [online]. 2023. Available also from: https://kotlinlang.org/docs/data-classes.html. Accessed on 2023-03-30.

33. *Null safety* [online]. 2023. Available also from: https://kotlinlang.org/docs/null-safety.html#the-operator. Accessed on 2023-04-12.
34. FELKER, Donn. *176: Kotlin's !! Operator is a Code Smell* [online]. 2019. Available also from: https://open.spotify.com/episode/2OGsY6iuKx6ZL5EKLxJyVn. Accessed on 2023-04-12.
35. *Extensions* [online]. 2023. Available also from: https://kotlinlang.org/docs/extensions.html. Accessed on 2023-03-30.
36. *Functions* [online]. 2023. Available also from: https://kotlinlang.org/docs/functions.html. Accessed on 2023-04-12.
37. *Conditions and loops* [online]. 2023. Available also from: https://kotlinlang.org/docs/control-flow.html. Accessed on 2023-03-30.
38. *Sealed classes and interfaces* [online]. 2023. Available also from: https://kotlinlang.org/docs/sealed-classes.html. Accessed on 2023-03-30.
39. *Kotlin multiplatform* [online]. 2023. Available also from: https://kotlinlang.org/docs/multiplatform.html. Accessed on 2023-07-05.
40. JANSSEN, Thorben. *Design patterns explained – dependency injection with code examples* [online]. 2023. Available also from: https://stackify.com/dependency-injection/. Accessed on 2023-03-30.
41. *Koin - The pragmatic Kotlin dependency injection framework* [online]. 2023. Available also from: https://insert-koin.io. Accessed on 2023-03-30.
42. MONTIEL, Ivan. *Low Coupling, High Cohesion* [online]. 2018. Available also from: https://medium.com/clarityhub/low-coupling-high-cohesion-3610e35ac4a6. Accessed on 2023-03-30.
43. BAJPAYEE, Vikas. *Clean architecture Android— important points and terms* [online]. 2020. Available also from: https://medium.com/@vikas.bajpayee/clean-architecture-android-important-points-and-terms-23272265e262. Accessed on 2023-03-31.
44. *Compose layout basics* [online]. 2023. Available also from: https://developer.android.com/jetpack/compose/layouts/basics. Accessed on 2023-03-31.
45. *MVC (Model View Controller) Architecture Pattern in Android with Example* [online]. 2020. Available also from: https://www.geeksforgeeks.org/mvc-model-view-controller-architecture-pattern-in-android-with-example/. Accessed on 2023-03-28.
46. MUNTENESCU, Florina. *Android Architecture Patterns Part 1: Model-View-Controller* [online]. 2016. Available also from: https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6. Accessed on 2023-03-28.
47. GALLARDO, Estefanía García. *What Is MVVM Architecture?* [Online]. 2023. Available also from: https://builtin.com/software-engineering-perspectives/mvvm-architecture. Accessed on 2023-03-29.
48. CHUGH, Anupam. *Android MVVM Design Pattern* [online]. 2022. Available also from: https://www.digitalocean.com/community/tutorials/android-mvvm-design-pattern. Accessed on 2023-03-29.
49. GALOS, Maciej. *Modern Android Architecture with MVI design pattern* [online]. 2021. Available also from: https://amsterdamstandard.com/story/modern-android-architecture-with-mvi-design-pattern. Accessed on 2023-03-29.
50. GAZZAH, Rim. *MVI Architecture with Android* [online]. 2020. Available also from: https://medium.com/swlh/mvi-architecture-with-android-fcde123e3c4a. Accessed on 2023-03-29.

51. VASUDEV, Shwetha. *MVI Architecture* [online]. 2020. Available also from: https://blog.mindorks.com/mvi-architecture-android-tutorial-for-beginners-step-by-step-guide/. Accessed on 2023-03-29.
52. STRULOVICH, Omer. *From zero to 10 million lines of Kotlin* [online]. 2022. Available also from: https://engineering.fb.com/2022/10/24/android/android-java-kotlin-migration/. Accessed on 2023-04-12.
53. *Metrics for OkHttp's Kotlin Upgrade* [online]. 2019. Available also from: https://publicobject.com/2019/05/13/metrics-for-okhttps-kotlin-upgrade/. Accessed on 2023-04-12.
54. KOMEN, Benjamin. *Using Kotlin in a Java project: 10 lessons learned* [online]. 2021. Available also from: https://xebia.com/blog/using-kotlin-in-a-java-project-10-lessons-learned/. Accessed on 2023-04-12.
55. ALVELO, Nikolas. *Make It Kotlin! — Migrating a Purely Java Android App at Scale* [online]. 2021. Available also from: https://medium.com/draftkings-engineering/make-it-kotlin-migrating-the-purely-java-android-app-at-scale-906395c6a34e. Accessed on 2023-04-12.
56. *Migrating to Jetpack Compose* [online]. 2023. Available also from: https://developer.android.com/codelabs/jetpack-compose-migration. Accessed on 2023-04-12.
57. *Dokka* [online]. 2023. Available also from: https://github.com/Kotlin/dokka. Accessed on 2023-04-18.
58. *Test apps on Android* [online]. 2023. Available also from: https://developer.android.com/training/testing. Accessed on 2023-04-01.
59. DIJKSTRA, Edsger Wybe et al. *Notes on structured programming* [online]. Technological University, Department of Mathematics, 1970. Available also from: http://dea.unsj.edu.ar/informatica1/recursos/Apuntes/Unidad6/2-NotesOnStructuredProgramming-Dijkstra.PDF.
60. *Fundamentals of testing Android apps* [online]. 2023. Available also from: https://developer.android.com/training/testing/fundamentals. Accessed on 2023-04-01.
61. VOCKE, Ham. *The Practical Test Pyramid* [online]. 2018. Available also from: https://martinfowler.com/articles/practical-test-pyramid.html. Accessed on 2023-04-01.
62. *Use test doubles in Android* [online]. 2022. Available also from: https://developer.android.com/training/testing/fundamentals/test-doubles. Accessed on 2023-04-01.
63. FOWLER, Martin. *Mocks Aren't Stubs* [online]. 2007. Available also from: https://martinfowler.com/articles/mocksArentStubs.html. Accessed on 2023-04-01.
64. *Automate UI tests* [online]. 2022. Available also from: https://developer.android.com/training/testing/instrumented-tests/ui-tests. Accessed on 2023-04-01.
65. FOWLER, Martin. *TestPyramid* [online]. 2012. Available also from: https://martinfowler.com/bliki/TestPyramid.html. Accessed on 2023-04-01.
66. *What are Flaky Tests?* [Online]. [N.d.]. Available also from: https://www.jetbrains.com/teamcity/ci-cd-guide/concepts/flaky-tests/. Accessed on 2023-04-01.
67. COHN, Mike. *Succeeding with agile: Software development using scrum*. 1st ed. Addison-Wesley, 2013. Addison-Wesley Signature Series. ISBN 0321579364.
68. FELKER, Donn; GOPAL, Kaushik. *174: Testing RxJava, Debugging and More* [online]. 2019. Available also from: https://open.spotify.com/episode/5O3XA61zsBtXacxG4uV2pd. Accessed on 2023-04-01.

69. *Testing* [online]. 2022. Available also from: https://moodle-vyuka.cvut.cz/pluginfile.php/530559/course/section/84272/2020_2021/06_Testing.pdf. Accessed on 2023-04-02. Translated by author on 2023-04-02.

70. *Testing your Compose layout* [online]. 2023. Available also from: https://developer.android.com/jetpack/compose/testing. Accessed on 2023-04-02.

Appendix A
Acronyms

MVC	Model View Controller
MVVM	Model View ViewModel
MVI	Model View Intent
API	Application Programming Interface
DI	Dependency Injection
XML	Extensible Markup Language
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
SSOT	Single Source of Truth
OOP	Object-Oriented Programming
HTML	HyperText Markup Language
NPE	NullPointerException
DTO	Data transfer object
UI	User interface
JVM	Java virtual machine

Appendix B

Attachment contents

- `files` — Folder containing files with code snippets
  - `DTOClasses` — Folder containing examples of DTO classes
    - `Person.java` — Simple Java DTO class
    - `PersonLombok.java` — Simple Java DTO class with Lombok
    - `Person.kt` — Example of an DTO class in Kotlin
  - `JavaToKotlin` — Folder containing gradle files for adding Kotlin support
    - `app_build.gradle` — App level gradle file with Kotlin support
    - `project_build.gradle` — Project level gradle file with Kotlin support
  - `MVC-MVVM` — Folder containing MVVM set-up and after migration files
    - `Set-Up`
      - `activity_main.xml` — Activity layout with working ViewModel
      - `build.gradle` — App level gradle file with added ViewModel support
      - `MainActivity.kt` — Activity with working ViewModel
      - `MainViewModel.kt` — Simple ViewModel with one property
    - `After-Migration`
      - `activity_main.xml` — Activity layout with functionality with ViewModel
      - `MainActivity.kt` — Activity observing data from ViewModel
      - `MainViewModel.kt` — ViewModel handling Activity interactions
  - `Testing` — Folder containing examples of different types of tests
    - `ComposeUnitTest.kt` — Unit test testing Composable
    - `EndToEndTest.kt` — Compose end-to-end test
    - `IntegrationTestWithFake.kt` — Integrated test using Fake testing double
    - `UnitTestEmailValidator.kt` — Unit test testing EmailValidator
  - `ViewToCompose` — Folder containing simple example of working Compose Activity
    - `activity_main.xlm` — Activity layout file
    - `Main.kt` — Kotlin file containing Composable functions
    - `MainActivity.kt` — Activity setting ComposeView content
    - `MainViewModel.kt` — ViewModel with MainViewState class
  - `Implementation`
    - `EmailtextFieldError` — Previous implementation of email text field error
    - `LogInButtonEnabled` — Previous implementation of enabling log in button
    - `ProgressBar` — Previous implementation of Progress bar
- `thesis.pdf` — Thesis in PDF format
- `dokka` — Folder containing documentation
- `thesis` — Source code of the thesis
- `app` — Source code of the application