Analysis of the Usage of Specific Technologies in Android Development

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING
INGEGNERIA INFORMATICA

Author: **Robert Medvedec**

Student ID: 962348
Advisor: Luciano Baresi
Academic Year: 2021-22

Abstract

Android is a world-leading operating system for smartphone devices. Being an open-source platform it has spread to devices from many different manufacturers and is supported by a wide range of hardware. Development of Android applications is an advanced process in which a number of technologies are being used to allow for creation of applications for many different use cases. The state of the Android development is rapidly changing due to both software and hardware evolving every year. The goal of this thesis is to capture the current state of Android development as well as present the most used technologies and their use cases for specific types of applications.

To achieve this, an analysis was made on 27 widely used open-source Android applications. The analysis includes individual scan of each of those applications with a deep dive into the language, components, patterns, and services used in the development. The overall analysis takes into the individual results of analyzed applications to create an snapshot of a current state in Android development.

Key-words: Android development, open-source applications, Java, Kotlin, software architecture

Abstract in italiano

Android è il sistema operativo leader mondiale per i dispositivi smartphone. Essendo una piattaforma open source, si è diffusa su dispositivi di molti produttori diversi ed è supportata da un'ampia gamma di hardware. Lo sviluppo di applicazioni Android è un processo avanzato in cui vengono utilizzate una serie di tecnologie per consentire la creazione di applicazioni per molti casi d'uso diversi. Lo stato dello sviluppo di Android sta cambiando rapidamente di anno in anno a causa dell'evoluzione di software e hardware. L'obiettivo di questa tesi è catturare lo stato attuale dello sviluppo di Android e presentare le tecnologie più utilizzate e i loro casi d'uso per specifici tipi di applicazioni.

Per raggiungere quest'obiettivo, è stata effettuata un'analisi su 27 applicazioni Android open source ampiamente utilizzate. L'analisi include la scansione individuale di ciascuna di queste applicazioni con un'analisi approfondita del linguaggio, dei componenti, dei modelli e dei servizi utilizzati nello sviluppo. L'analisi complessiva prende in considerazione i singoli risultati delle applicazioni analizzate per creare un'istantanea dello stato attuale nello sviluppo di Android.

Parole chiave: Sviluppo Android, applicazioni open-source, Java, Kotlin, architettura software

Contents

Abstract .... **i**
Abstract in italiano .... **iii**
Contents .... **v**
Glossary .... **viii**
Introduction **9**
1 Android basics **11**
1.1. Short history of Android 11
1.2. Android by versions (API levels) 11
1.3. Android development 13
1.4. Android development environments 14
2 Main characteristics of technologies used in Android development **15**
2.1. Programming languages 15
2.1.1. Java 15
2.1.2. Kotlin 15
2.1.3. Java vs Kotlin 16
2.1.4. C/C++ 18
2.2. Architectural patterns 18
2.2.1. MVC (Model – View – Controller) 19
2.2.2. MVP (Model – View – Presenter) 21
2.2.3. MVVM (Model – View – ViewModel) 22
2.3. Android libraries 23
2.3.1. AndroidX 23
2.3.2. Android Jetpack 23
2.4. Design patterns 29
2.4.1. Creational patterns 29
2.4.2. Structural patterns 31
2.4.3. Behavioral patterns 32
2.5. Android application components 33
2.6. Other commonly used products and services 34
2.6.1. Google Firebase 34

2.7. ProGuard (R8) 38
3 Analysis results 39
3.1. Analysis goals 41
3.2. Problems with certain metrics during the analysis 41
3.2.1. Measuring source code size and the number of files 42
3.2.2. Information about the first app version release 42
3.2.3. Measuring the exact number of screens 42
3.2.4. Counting dependencies 42
3.3. Application analysis 42
3.3.1. Browsers 43
3.3.2. Commercial applications 47
3.3.3. Media players 52
3.3.4. Messaging and email 57
3.3.5. Other 62
3.3.6. Tech demo 67
4 Notable selected applications 71
4.1. Mozilla Fenix (Firefox) 71
4.2. Muzei 74
4.3. Kickstarter and WordPress 77
4.4. NotyKT - Views and Compose version 79
4.4.1. Shared components 79
4.4.2. Individual components 79
5 Conclusions and results comparison 83
5.1. Application complexity and size 84
5.2. Programming language 87
5.3. User interface (UI) 90
5.4. Google libraries and services 93
5.5. Architectural and design patterns 95
5.6. Conclusion 97
Bibliography 101
A Appendix A 105
A.1. Complete table of applications analysis results 105
A.2. List of analyzed applications' repositories 111
List of Figures 113
List of Tables 115
Acknowledgments 119

Glossary

Android - a mobile operating system based on Linux kernel, runs on many different mobile smartphones from different manufacturers

API - Application Programming Interface - a way for two or more computer programs to communicate with each other, in the context of Android development used as a way of saying that one software is using the services and the data that is fetched from another library or website

IDE - Integrated Development Environment - software for building applications that combines common developer tools into a single graphical user interface, the main tool being a source code editor

iOS - a mobile operating system developed by Apple, runs exclusively on their line of products (iPhones, iPads, iPods)

JVM – Java virtual machine – a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode

Open-source - denoting software for which the original source code is made freely available and may be redistributed and modified

OS - Operating system - system software that manages computer hardware, and software, and provides common services for computer programs

QA - Quality assurance - a customary practice that assures that the developed product meets certain expectations and is ready to be publicly available

UI - User interface - referred to everything that the user can see on the screen and interact with – similar to GUI – Graphical user interface

XML - Extensible Markup Language - a markup language for storing structured data, referred to the files that describe the UI on the screen in the View presentation blocks that are stored in that specific file format (.xml)

Introduction

Android devices make up more than two-thirds of the smartphone market as of 2022. With the vast majority of the human population in developed countries owning such a device, applications that run on smartphones are not only shaping the way people use their phones but also shaping how everyone leads their lives.

Development of Android applications ever since Android OS inception in 2008 has been rapidly evolving and quite often rashly changing due to quick technological advancements in both mobile and computer hardware capabilities. Defining the current state of Android development and pinpointing the most used languages, technologies, IDEs, architectural patterns, and other elements have never been easy tasks due to such rapid changes.

In recent years Android OS creators, a consortium led by Google, managed to slow down the evolution of the development by sticking to a certain approach and technologies, however good or bad they might be in a general sense, creating at least some sort of stability in the Android app development world. The thrust from other developers to make development for other systems, most notably iOS, as closely as possible connected to the Android development goes hand in hand with Google's intention and is further stabilizing the technologies used. All of this is to allow developers to create better and more innovative apps with their focus shifted to execution rather than catching up with the recent technologies.

In this work, the current state of the most used technologies in Android application development has been thoroughly researched, explained, and presented. Therefore, a "snapshot" of the state of technologies used has been created. The aim is not only to determine which technologies are the most used but also to try and find out which combination of them achieves the best results and what the future of Android applications is going to look like, assuming there are no drastic technological changes in the upcoming years.

The research is done on the most downloaded and rated open-source applications that still have active repositories and recent releases, listed in Table A.2. Many of these applications are used by big companies and quite often function as companion apps for selling their main product.

Analysis of these applications is done on multiple different parameters, with most of them regarding technologies and architectural patterns used, but also considering the size, complexity, and functionalities of the app.

The main hypothesis going into this work is that the latest technologies offer the most flexibility and the best performance, with the newest applications that integrate the most recent technologies being the easiest to use, analyze, and develop. Adding to that is that most of the developers have or will have switched to the newer technologies knowing that they will be supported and updated for many years, having the full backing of Google and its developers, making development faster and easier.

The document is divided in the following sections:

- Chapter 1 introduces the concept of Android OS and Android development. The main focus is on what Android as a platform actually represents, its evolution through the years, and description of the development process on the platform. The chapter also touches on brief history of different Android OS versions as well as IDEs that are used most often for its development.
- Chapters 2 puts the focus on the Android stack and the whole technical background of the OS. It contains the descriptions of the most common architectural patterns, languages, design patterns, and other technologies surrounding the Android software development, with multiple examples of how they work as a system. The latest Android libraries and development practices are presented with more detail.
- Chapter 3 contains a concise description of the applications used in the analysis, the process of app selection, and the main questions that are attempting to be answered by the analysis. The bigger part of the chapter focuses on individual analysis of each app, which consists of a brief description, analysis results, and a post-analysis comment.
- Chapter 4 is divided into four subchapters, each of them providing a deeper analysis or a deeper dive into some of the analyzed applications. This chapter is meant to describe some of the most common cases in Android development in greater depth and to apply the previous knowledge from the work to the real-life cases.
- Chapter 5 features a conclusion on the current state of the Android development based on the analysis results. Explanations of the findings and the most common patterns are also located in this chapter. Comparison of the previously made hypothesis with the actual results is located at the end of this work together with the prognosis for the future of Android development.

1 Android basics

1.1. Short history of Android

Android is a mobile operating system that came to focus with the emergence of smartphones in the late years of the first decade of the 21st century. Based on a modified version of the Linux kernel it is mostly used on touchscreen devices, phones, and tablets, although its use has expanded to other products like TVs and watches. It is currently, as it has been almost since its inception, the most common operating system on mobile phones with 71.55% of the world market share, while iOS-based devices hold all the remaining 28% percent. [1]

Android is developed by a consortium of developers called Open Handset Alliance, although it is primarily sponsored and supported by Google, the de facto owner of the system. The core version of Android is open source, although most of the devices come with pre-installed proprietary software adapted to the specific device by different companies like Samsung, Huawei, and OnePlus.

Throughout the years Android has been known as the more "open" of the two main mobile OS systems and has managed to firmly establish itself as the most used one, mostly since the price range of the devices supporting Android is very wide, which cannot be said about the iOS supported devices. Despite that, there have been several different systems that have not managed to win the competition of the operating systems, most notably Windows Phone, which has been in production by Microsoft.

Along with the previously stated, Android has been more severe in evolution and has had major changes in the releases, which has often led to bad reactions from both developers and users. In the end, after 13 major releases and more than twenty minor ones, it has established its identity and has grown to be a preferred system for developers around the world, with Google's efforts in making it more standardized being a huge factor.

1.2. Android by versions (API levels)

To understand the current state of the OS it is also necessary to know its past and the path it has taken to reach this level.

The first version called Android 1.0 was released in September 2008 for HTC Dream, the first commercially available Android device. From the next big release in 2008, Android was named after a certain sweet, different one for each version, which was the naming convention up to Android 10.

Android 1.5 Cupcake vastly improved the Android interface by adding an on-screen keyboard, enabling the phones to be mostly screen-based with only a few buttons. Along with that, third-party widgets were added, which were for years an Android-only feature on smartphones, until Apple introduced them on iOS more than 10 years later. At this time Android minor and major updates were separated by only weeks, as the platform was improving dramatically. This was to settle in the future as Google started adopting the more common yearly update model.

Android 2.0 Eclair added voice-guided turn-by-turn navigation integrated into maps and pinch-to-zoom capability, which was up to that point exclusive to Apple - this would launch an unofficial war between the two companies and "borrowing" each other's software characteristics would become more common between them.

Expansion to tablets came in 2011 with Android 3.0 Honeycomb as it opened up an entirely new range of devices for Android.

Android 4.0 Ice Cream Sandwich introduced widgets, a feature that is going to be a huge identity element of the Android OS and that will differentiate it from the competitors.

Android 4.4 KitKat featured Google Assistant integration for the first time.

In the next several features Android was slowly adding some minor updates, visual changes, and some novelties that would only prove to be more important in the years to come (like split-screen mode, support for fingerprint readers, support for USB-C, etc.).

Android versions 9 and 10 focused more on the user's privacy and security, battery and performance optimization, productivity, visuals, and navigation, most of which look the same today. These two versions made the Android look and feel more authentic and differentiated it even more from its competitor.

The biggest update when it comes to development came with a new set of Android Jetpack libraries, called *androidx*, which required API level 28 (Android 9) and featured full compatibility with older API levels and libraries. It featured a whole new set of libraries for the most used functionalities in app development. A new base was successfully set up and Google has been building on it ever since.

The last three versions that came out - 11, 12, and 13 - are mainly minor reskins and feature mostly quality-of-life improvements. As Android OS has already mostly reached its recognizable look and feel, Google is shifting its focus on the more functional areas of the whole user experience and is experimenting with new features that are currently not available for iOS phones. This stability and the lack of major changes also allowed for the development experience to be more streamlined, stable, and documented, with Google staying on course with technologies used for it and for the first time in years trying to keep the main parts of the development intact.

The market share of Android OS versions as of October 2022 is presented in the following table [2]:

Table 1.1: Usage of Android OS versions

	Percentage of usage (%)
Android 12	29
Android 11	26
Android 10	19
Android 9	9
Android 8	8
Other API levels	9

1.3. Android development

Characteristics of Android development experience have vastly changed throughout the years and are now much more positive than before. In the early phase, Android developers complained about the lack of documentation, frequent bugs in the OS and IDEs used for Android development, and no QA support which made apps more prone to bugs. Google has often listened to the wishes of the developer community and made great improvements in these areas. These days the whole experience has changed, with extremely detailed documentation and a vast number of examples made by Google, which can easily be found on GitHub pages. Since the community has grown rapidly many those examples can also be found in other open-source projects and many different forums, making the whole development a lot easier for new developers. Google also supports Android by creating many different videos and functionalities previews which they consistently post and write extensive articles about.

Official Google conferences all around the world made for sharing and acquiring knowledge are available to everyone live or on-demand for free. With the addition of Google Nexus and later Google Pixel phones to their line-up, a big step was made in making new Android versions as good as possible on their release dates. These devices get exclusive access to the newest updates, which allows Google to test some new features on a limited number of users, gathering feedback and improving the features before the full release. As of 2022, Android development has reached an exceedingly prominent level and is regarded as one of the most documented and advanced technologies on the market. With Google finally stabilizing the developing environment and going for a long-term focus on technologies and language, it is safe to say that the platform is as advanced and as stable as it has ever been.

1.4. Android development environments

Early in the Android development, there was no official IDE as the technology was still catching on.

Eclipse IDE, originally created by IBM but was later taken over by the Eclipse Foundation, was a natural choice for most Android developers since most people used it as their preferred Java programming environment, a language that was natively supported at the time by Android.

Google released a set of tools called ADT (Android Development Tools) that integrated native Android dev support into Eclipse. Google was improving the experience and releasing new updates to the tools regularly until they announced their IDE for Android development called Android Studio, which was to be made in collaboration with JetBrains, the creators of the already popular IDE IntelliJ IDEA on which Android Studio is heavily based.

With the launch of Android Studio in late 2014, Google's official support for Eclipse IDE was discontinued, and Eclipse Foundation released its plugin called *Andmore: Development Tools for Android*. Andmore only lasted two years and never reached a stable version as it was discontinued in early 2017 with version 0.5.1. Android Studio took over the market share ever since the beginning. Continuous Google and IntelliJ support and vast improvements meant that no other software was able to compete with it. As of 2022, almost all the native Android development is done in Android Studio, while there are still some other solutions that are used for cross-platform programming. In 8 years, Android Studio has received more than twenty major updates and several new features that have vastly improved the development experience. Like the naming convention of Android OS versions, Google has dropped the numbering system and decided to go alphabetically with different animal names.

The version of Android as of writing this work is called Dolphin and was released in September 2022. Two additional versions are planned to be released in 2023 by the names of Electric Eel and Flamingo.

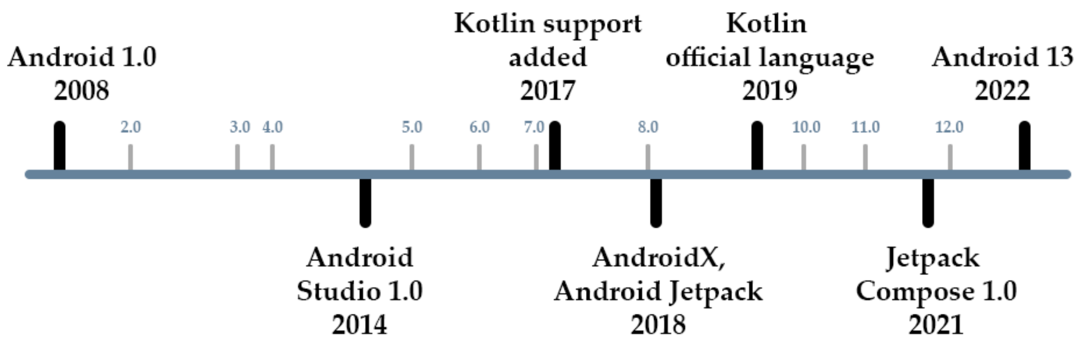


Figure 1.1: Android development timeline

2 Main characteristics of technologies used in Android development

2.1. Programming languages

The native Android development environments support two programming languages - Java and Kotlin. Additional C/C++ code can be interpolated in the app by using other tools that help with the internal translation to Java language. One of the focuses of this work are the main differences between Java and Kotlin, the number of apps that are using one or another, and their advantages and disadvantages.

2.1.1. Java

Java is a language that first appeared in 1995, developed by Sun Microsystems. It is a high-level, class-based, object-oriented language that is designed to have as few dependencies as possible.

It quickly became one of the most used languages in the world, a status which still maintains to this day.

Java was originally the preferred and default language of the platform.

One major advantage of developing software with Java is its portability. Once you have written code for a Java program on a notebook computer, it is extremely easy to move the code to a mobile device. When the language was invented in 1991 by James Gosling of Sun Microsystems (later acquired by Oracle), the primary goal was to be able to "*write once, run anywhere*."

The main technical advantages of Java are interoperability, scalability, and adaptability. It is an object-oriented language that allows for a creation of modular programs and reusable code, which is perfect for scaling applications.

2.1.2. Kotlin

Kotlin is an Android-compatible language that is concise, expressive, and designed to be type- and null-safe. It works with the Java language seamlessly, so it makes it easy for developers who love the Java language to keep using it but also incrementally add Kotlin code and leverage Kotlin libraries.

Kotlin is a relatively new language in the programming world - created by JetBrains in 2011 it is a cross-platform, statically typed, general-purpose programming language. It was designed to be fully interoperable with Java, with more concise syntax and many additions that make programming easier and faster. It mainly targets JVM but can also be compiled in JS or native code.

Android Kotlin compiler produces Java 8 bytecode by default, but it also supports other Java versions from 9 to 18.

2.1.3. Java vs Kotlin

Kotlin officially became the preferred Android dev language in 2019. Up until that point, Java has firmly held the position as the main and preferred language, with only C++ being supported from the other languages, and Kotlin being added as the supported language in 2017.

There are many alleged reasons why Google decided to officially "switch" to Kotlin as the main language (although Java is still supported in the same way as it was before), but Google never officially listed the main reasoning behind it. This work will not focus on the reasoning, but rather on the advantages and disadvantages of the respective languages.

Both languages can still be used on the same projects and there is even a language translator directly integrated into Android Studio which allows for seamless code translation. Some projects are still written in both languages, as rewriting the old code from Java to Kotlin brings no major advantages and is still time-consuming, so some developers opted for keeping the old code and writing all the new code in Kotlin.

Their real-time performance when it comes to execution is on-par. [3] Java keeps the advantage in compile times due to the fact that Kotlin needs to first be “translated” to Java bytecode, which adds additional overhead. Google is claiming to be dropping the compilation time difference with every new update. The execution of the code is approximately the same and offers no functional differences besides the number of lines of code.

Table 2.1: Java and Kotlin differences

[4] [5] [6] [7]	**Java**	**Kotlin**
Amount of code	More declarative, no lambda or inline functions	Less verbose, allows quick constructors/lambda functions, inline functions
NullPointerException (NPE)	Uses NPE to prevent access to undefined object, causes crashes if not caught	Uses safe calls (.?) which will prevent the execution of the method if the object is undefined, no crashes
Coroutines	Doesn't support coroutines, provides other less-efficient methods	Has full coroutines support
Performance	Faster compilation, executes on JVM	Slower compilation, executes on JVM
Data typing	Requires variable specification	Doesn't require variable specification (uses *var* and *val)*
Android support	Supports of all of the basic features, most of the native libraries still written in Java	Many new features (like Jetpack Compose) are built on Kotlin and are meant to be used with Kotlin, new annotation types support, continuous platform updates that make the coding experience better
Smart cast	Doesn't support smart casting	Supports smart casting
Primitive types	Has primitive types that are not classes	Doesn't have pritive types that are not classes – byte code may use them when possible
Overall conclusion	More difficult to write, easier to debug, older and bigger community, more online support and learning materials, most of the libraries are still being written in Java	Easier to write, harder to debug, more user-friendly, features many improvements from Java, still has a young community, has massive support from Google

2.1.4. C/C++

2.1.4.1. JNI

Another way of programming Android apps (or some elements of them) is by using Java Native Interface (JNI) - a programming framework that enables Java code running in a Java virtual machine to be called by native applications. This allows for other languages to run their code and libraries, mostly in C and C++.

Development in C++ for Android apps is not common, but many apps use some very low-level code and libraries that are non-existent in Java/Kotlin, mostly due to the complexity and the performance drop they would achieve. This allows the developers to code in C++ and implement it in their Java/Kotlin code without having to rewrite all of the code to the native language. JNI features some overhead, as the translation from C/C++ to Java is not seamless. It also allows direct access to assembly code, shortening the overhead time.

Since Java and Kotlin are native to Android development C/C++ is not destined to run better or faster than the other two languages. Native C++ development, most often used for cross-platform is also possible in Android development, although the emergence of KMM (Kotlin Multiplatform Mobile) and Flutter is making cross-platform development much easier to access.

2.1.4.2. Android NDK

There exists a toolset that allows for the implementation of apps in native code by using libraries in other languages like C and C++. Certain types of apps and some app functionalities which require high performance and are most often low-level require these libraries to run fast and efficiently.

The main difference between Android NDK and JNI is that JNI uses some of the functions from native (C/C++) code and is inserting them into the Java language environment, still using other Java functionalities and compiling normally in Java.

Android NDK compiles the native code into a native library which can then later be reused. These two technologies are not interchangeable and are generally not used for the same solutions.

2.2. Architectural patterns

Aside from the programming language and platform, the most important choice in android development is the architecture of the software. Software architecture is a set of fundamental structures of a software system that defines software elements and the relations between them.

The term "architecture" was originally taken from the architecture in buildings to make a connection with the foundations of the structure and emphasize the importance of the internal part of the system. Although the general significance behind the term has mainly remained the same in the public, computer scientists have had different opinions on what the term means in the computer world. Also, as the technology itself evolved and got more complicated, the definitions grew more complex.

The choice of the right architecture for a certain system depends on many varied factors - the goal of the app, available resources, technologies used, and many more. When it comes to Android development the evolution of architectures used has been relatively steady, with them being mostly smaller adjustments to the previous system. Of course, they are not real evolutions, as many of the previously most popular architectures are still being used both in Android development and outside of it. The change in the most popular architecture was always followed by a major technological shift, although it is still not rare to see hybrids of architectures due to the fast pace of technological advancement in the mobile world.

The most used architectural patterns in Android development are:

- **MVC (Model - View - Controller)**
- **MVP (Model - View - Presenter)**
- **MVVM (Model - View - ViewModel)**

2.2.1. MVC (Model – View – Controller)

MVC is an architectural pattern in software design commonly used to implement user interfaces, data, and controlling logic. [8] It emphasizes the separation between the software's business logic and display. This "separation of concerns" provides for a better division of labor and improved maintenance. Other architectural patterns are based on MVC, such as MVVM (Model-View-ViewModel), MVP (Model-View-Presenter), and MVW (Model-View-Whatever).

The three parts of the MVC software design pattern can be described as follows:

Model: Manages data and business logic.

View: Handles layout and display.

Controller: Routes commands to the model and view parts.

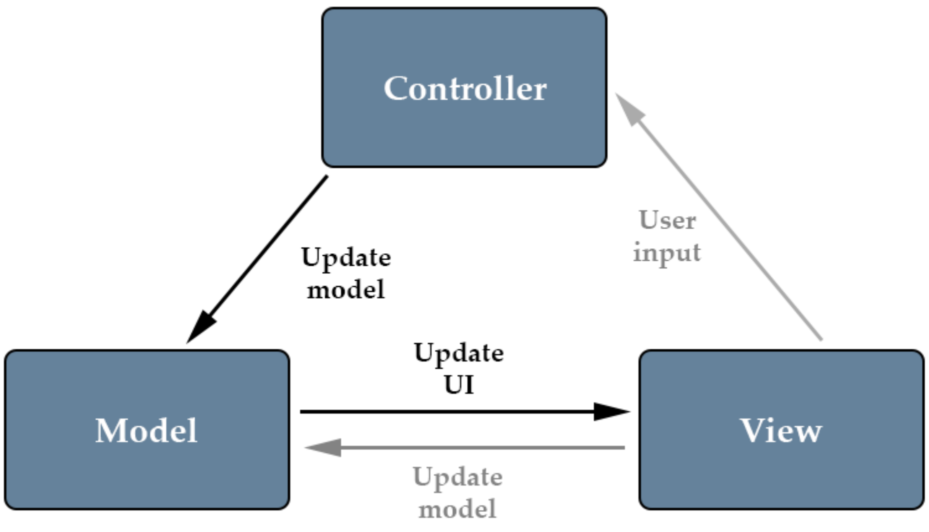


Figure 2.1: MVC scheme

The important thing to notice is that the View component, which is the part that is visible to the user, has no direct control over the data. It can communicate only with the Controller which then also communicates with the Model creating a chain of communication and adding another abstraction and control layer. The three elements are very distinctly separated and allow for better control over specific parts of the system. Programming these elements can therefore be completely separated and each component should work the same regardless of the ongoing changes in the other ones.

MVC decouples views and models by establishing a subscribe/notify protocol between them. A View must ensure that its appearance reflects the state of the Model. Whenever the model's data changes, the model notifies views that depend on it. In response, each View gets an opportunity to update itself. This approach allows to attach multiple views to a Model to provide different presentations.

This architecture allows for all of the three major elements to be completely independent of each other which has many advantages - interchangeability, interoperability, reusability, and faster debugging, just to name a few.

This architecture has been the most popular in mobile app development for many years while also evolving due to changes in other technologies used in development.

2.2.2. MVP (Model – View – Presenter)

MVP is a derivation of the MVC architectural pattern, mostly used for building user interfaces where the presenter assumes the functionality of the "middle-man" and holds the entire presentation logic. [9]

It builds up upon the MVC pattern and improves on some of the main disadvantages that it has - most notably unit testing and the size of the Controller layer, which holds most of the provided logic.

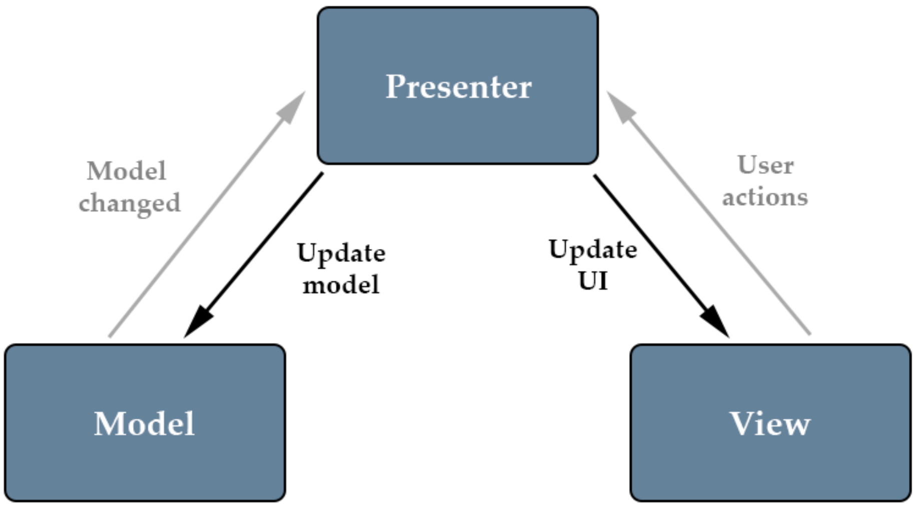


Figure 2.2: MVP scheme

Improvements in modularity and testing are done by completely separating View and Model layers with the presenter handling all of the updates and responses. This allows for centralized control and fully observable communication between the layers.

For every View class, there is also a Presenter class with a one-to-one relationship, while there is only one model that communicates with many presenters. Model and View layers are in no way connected.

One of the main potential issues in MVC on Android is having a lot of the application logic in the Activity thus limiting the developer to having to do everything logic-wise through it. MVP takes away the entire business logic from the Activity which now only holds the Views.

There also exists a version of MVP called *Supervising controller* which directly connects the View layer with the Model layer to allow for faster data flow through data binding, although this architecture re-introduces some of the same issues that MVC has.

2.2.3. MVVM (Model – View – ViewModel)

MVVM is an architectural pattern in software design that facilitates the separation of the GUI from the development of the business logic. View is therefore not dependent on any specific model platform.

In this architecture, ViewModel serves as a value converter from the Model to the View, allowing for easy and flexible use regardless of the technologies used. ViewModel also holds the most logic behind the View. The easiest explanation would be to describe the ViewModel as the state of the data in the model, which is then shown through the View to the user. [10]

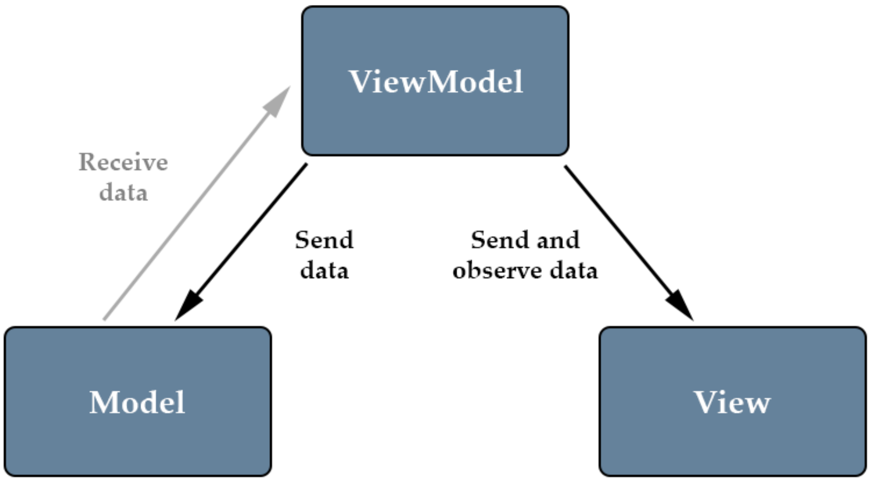


Figure 2.3: MVVM scheme

The rationale is using the data binding functions through Binder to fully remove any logic behind the View code and therefore offer full separation of the View layer. While being remarkably similar to MVP in many aspects, it is more event-driven and communicates with the user by representing the current state of the data. ViewModel has no reference to the View and the View only extracts specific data from the ViewModel when it needs it. The amount of code in classes is generally smaller than in other architectural patterns and due to a high degree of separation, unit testing is simple and effective.

2.3. Android libraries

Google provides a set of libraries and support services that aim to improve Android development and make it more streamlined. These libraries have initially been scattered and were not a part of any group which would specify to the developers what should they use for the best development experience. This has changed in recent years and Google has released two big sets of libraries that are recommended to use in all new Android projects.

2.3.1. AndroidX

One of the major breakthroughs in Android development was the release of the Android Extension Library, also known as AndroidX. [11] It is a set of libraries that comprises all of the Android Jetpack libraries, which not only bring new features to Android developers but also provide backward compatibility across Android releases. It was released as the replacement for the Android Support Library, replacing all the previous features and adding many others. The first stable release of AndroidX 1.0.0 was in September 2018, coinciding with the release of Android 9.

AndroidX improved the coding experience by having libraries under a unique namespace, reworking some of the functionalities, and making it all more accessible to developers. New sub-libraries are still being added. It currently holds around one hundred libraries, with the majority of them already having at least one stable version.

The biggest advantage of the library is having much smaller and more focused packages, which allow developers to include in their code only the sub-libraries they need, therefore reducing the size of both the code and the app. Renaming and restructuring of some libraries also resolved the problems of the old Support Library which has become cluttered with many sub-libraries, some of which were not even used, and featured many naming inconsistencies often creating confusion among developers about which libraries are needed.

Therefore *android.** namespace has been reserved for libraries that ship with the Android OS and *androidx.** for libraries that are unbundled and that are not directly connected to the Android OS.

2.3.2. Android Jetpack

Android Jetpack is a set of libraries, tools, and architectural guidance to help make it quick and easy to build great Android apps. It is a subset of the AndroidX library set.

It provides common infrastructure code so you can focus on what makes your app unique. It helps developers to follow best practices, reduce boilerplate code, and write code that works consistently across Android versions and devices.

Using libraries from Jetpack can significantly reduce code size and the number of crashes. [12] The four main categories of Android Jetpack are Foundation, Architecture, Behavior, and UI.

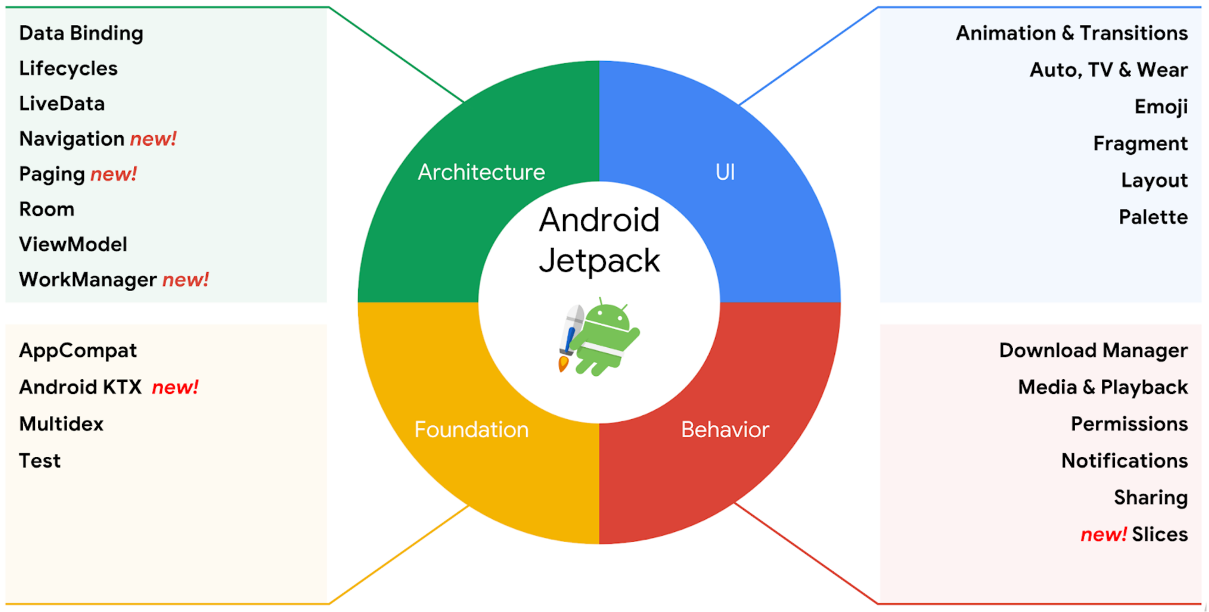


Figure 2.4: Android Jetpack components

List of the most used Jetpack libraries: [13]

- Activity - Access composable APIs built on top of Activity
- Camerax - Add camera capabilities to the app; the library provides several compatibility fixes and workarounds to help make the developer experience consistent across many devices
- Compose - Define UI programmatically with composable functions that describe its shape and data dependencies
- Hilt - Extend the functionality of Dagger Hilt to enable dependency injection of certain classes from androidx libraries
- Lifecycle - Build lifecycle-aware components that can adjust behavior based on the current lifecycle state of an activity or fragment
- Navigation - Build and structure the in-app UI, handle deep links, and navigate between screens
- Room - Create, store, and manage persistent data backed by an SQLite database
- Test - Testing in Android
- Work - Schedule and execute deferrable, constraint-based background tasks

Some of the most used libraries will be further explained either in this or the following subchapters as they are expected to be found in many of the analyzed apps and offer significant advantages to other alternatives.

2.3.2.1. Room persistence library

In 2017, Google introduced what was to become one of their most used libraries in the Android Jetpack package - Room. It is meant to allow inexperienced users to have an easier way of creating and handling database actions as its native integration to the app together with the SQLite language allowed for quick setup and easy use.

Many Room features are automatically integrated and many of the most basic DB functions do not require any code as they can easily be implemented just by annotating a method. The benefits of using Room are compile-time verification of SQL queries, annotations that minimize repetitive and error-prone boilerplate code and streamlining database migration paths.

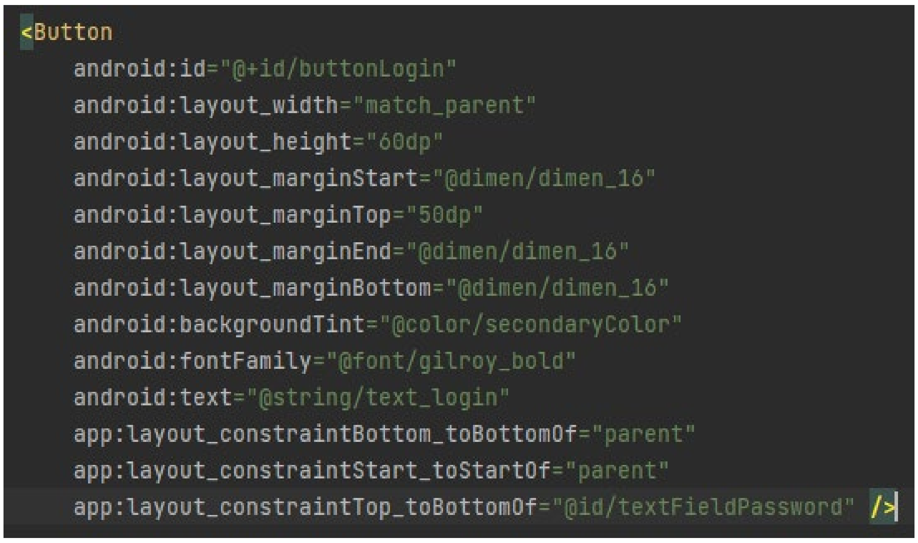
It is to be used as a local database that can then be synchronized and implemented with any traditional online SQL database. Room makes the offline experience much simpler and synchronizing between an online and offline version of the database almost seamless.

Data Access Objects (DAO) are used to define the relations between the app and the database and create methods that are going to be used when fetching and storing data from the database. Entities are then defined and stored in the database like regular class objects. On top of everything, Room Database is defined with both the entities and DAOs.

2.3.2.2. Jetpack Compose

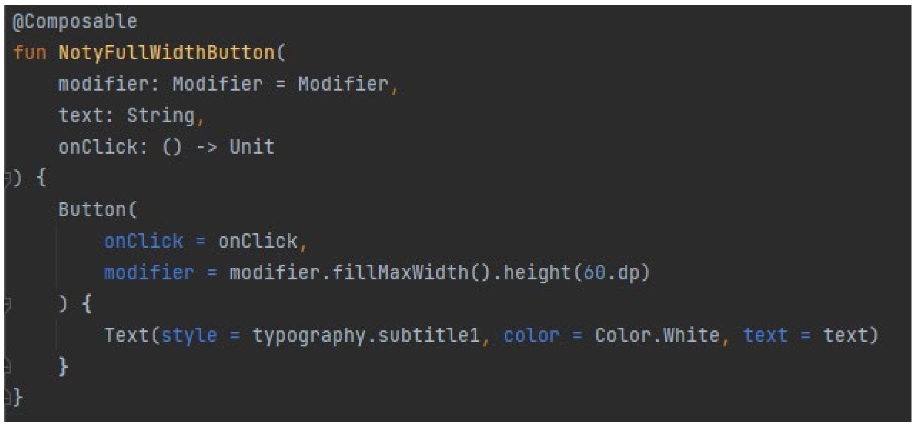
Jetpack Compose is Android's recommended, modern toolkit for building native UI. It simplifies and accelerates UI development on Android, bringing an app to life with less code, powerful tools, and intuitive Kotlin APIs. Compose is a new way of creating UIs aimed by Google to be a replacement for the older Views in XML form.

The first stable version of Jetpack Compose appeared in 2021, with the technology being available on the market for a couple of years before that in alpha and beta versions. During those years Google experimented with a lot of different functionalities and features and received mixed feedback from the developers. Many parts of the tool that were being complained about the most are still present in the stable version. Google is making a big effort to push Compose to the market and to "force" all of the new developers to use it in their projects.



```xml
<Button
    android:id="@+id/buttonLogin"
    android:layout_width="match_parent"
    android:layout_height="60dp"
    android:layout_marginStart="@dimen/dimen_16"
    android:layout_marginTop="50dp"
    android:layout_marginEnd="@dimen/dimen_16"
    android:layout_marginBottom="@dimen/dimen_16"
    android:backgroundTint="@color/secondaryColor"
    android:fontFamily="@font/gilroy_bold"
    android:text="@string/text_login"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/textFieldPassword" />
```

Figure 2.5: Button code example in XML



```kotlin
@Composable
fun NotyFullWidthButton(
    modifier: Modifier = Modifier,
    text: String,
    onClick: () -> Unit
) {
    Button(
        onClick = onClick,
        modifier = modifier.fillMaxWidth().height(60.dp)
    ) {
        Text(style = typography.subtitle1, color = Color.White, text = text)
    }
}
```

Figure 2.6: Button code example in Compose

2.3.2.3. Views

Views represent a basic building block for UI components. It is what the user sees on the screen and the first connection point between the user interaction and the application components. A single View occupies a rectangular area on the screen and is responsible for drawing and event handling.

Multiple Views can be combined to create multifunctional screens by using the ViewGroup, a subclass that can hold other Views and ViewGroups, and that serves as a base for layouts that are used to organize the way they are going to be presented.

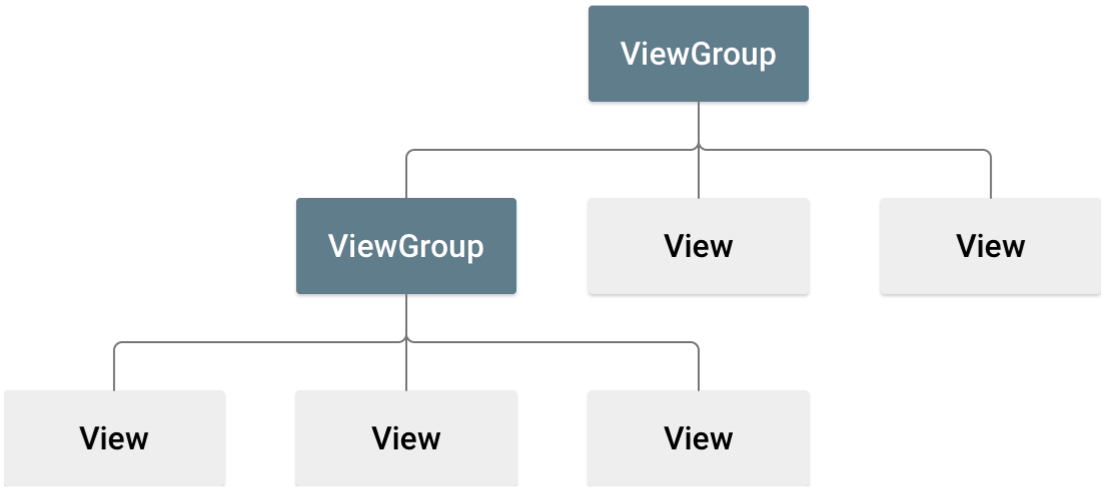


Figure 2.7: View hierarchy

Some of the most common Views are TextView, EditText, Button, ImageButton, ImageView, and RadioButton. The most common layouts of ViewGroup are LinearLayout, RelativeLayout, FrameLayout, and ListView. View components are attached to the Activity in the initial part of the Activity creation. Views are represented by XML files, which define the structure and hierarchy of all the elements contained in the View along with their attributes and references to other elements.

2.3.2.4. Composables

Composable elements, or functions, are Jetpack Compose elements meant to replace View objects and provide an alternative to representing the UI.

Composables are organized into functions that are then called upon by the Activity. They change their appearance based on the provided set of arguments and can be recomposed if any of those data arguments change. This provides an improvement to the system since only certain parts of the UI need to be recalculated and redrawn, and not the whole UI as was the case before.

Since the philosophy on how to represent the UI with Compose is completely different from the one with Views, a whole different approach needs to be taken when using these components. [14]

They represent a big step towards a declarative UI model, putting more focus on single components rather than the entire UI. Recomposition of the composables can be set by the developer and is generally a lightweight task that can be done multiple times a second without affecting the performance too much.

When a composable is called, it is typically passed some data and a set of properties that define how the corresponding section of the user interface is to behave and appear when rendered to the user in the running app. In essence, composable functions transform data into user interface elements. Composables do not return values in the traditional sense of the Kotlin function, but instead, emit user interface elements to the Compose runtime system for rendering.

2.3.2.5. Composables vs Views (XML)

The main advantages of Jetpack Compose compared to the Android View system, as stated by Google, are the following: [15]

- **Less Code**
    - Compose does everything in one language - Kotlin. There is no back-and-forth between Kotlin/Java and XML, where referencing elements and Views can quite often get messy and difficult, and where errors are inevitable. Compose also allows for better reusability - all of the components are written as classes, so to get a new component all that is needed is to create another instance of it and fill it with proper attributes and data. XML View system is not classed but rather file-based, therefore reusing Views often means either replicating the same file or creating subcomponents that are going to adapt the attributes of that file to specific needs.
- **Intuitive**
    - Compose uses a declarative API, meaning that only the UI description is needed, and the system takes care of the rest. Theming and coloring of the components are handled in a much simpler way and there is no need for multiple XML files defining different aspects of the look of the components. Compose elements work on a state basis, meaning that they can easily be changed by updating the state of it and recomposing it, which takes no time, rather than changing XML file attributes and re-creating the View.
- **Accelerated development**
    - Compose is compatible with all the previous code. It can be used interchangeably with Views and there is no need to rewrite the old code written in View architecture.
- **Powerful**
    - Compose features support for Material Design, dynamic theming, animations, and more. Animations and dynamic pages are easy to implement, bring a high degree of configuration, and have predefined libraries which do not require a lot of designers and animators work for bringing the app to life.

On the other hand, Compose does not offer the main feature of traditional Views - simplicity. Due to the clear sections of the components that are easy to understand and visualize, the development experience with Views is much easier for newer developers. Elements can also be directly imported in several different forms, from vector to pixel types of files, which removes the step of "language" translation between the designers and the developers. Both methods have their advantages and flaws, but as Google keeps supporting one technology and completely ignoring another, it is not impossible for Views to completely be removed as a development option in the future.

2.4. Design patterns

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. [16]

Design patterns are used throughout software development since its early beginnings. One of the first major design patterns were classes - the base of object-oriented programming which ultimately became the most used design pattern. [17]

Some patterns are specific (more often used) in specific architectures as their advantages are best exploited only in certain use cases. There are three main classifications based on the use and the background of the patterns. In this work, only the ones that are used most often and appear in the analyzed applications are presented and described.

2.4.1. Creational patterns

These design patterns are used during the class instantiation. Inheritance and delegation are used in some of these patterns to make creating and attributing new instances easy and effective. Further subdivisions can be made into class-creation and object-creation patterns.

2.4.1.1. Builder

Builder pattern is used to simplify the creation of objects, deconstructing the construction of an object into multiple steps during which certain attributes of the object can be specified. This pattern offers easier construction of an object, type control, attribute constraints control, and makes code less prone to bugs and crashes.

2.4.1.2. Dependency injection (DI)

DI is a pattern in which an object or a function that needs to use other objects or functions, that on which it depends for its functionalities, is having them directly provided as a parameter, rather than having them instantiated again or grabbing them indirectly from some other classes. This greatly reduces the unnecessary duplication of code and reduces the overhead of having to create multiple new objects in complex classes. DI libraries make things even easier for developers by having automated parameters in objects and functions, eliminating the need for ever to worry about forgetting to insert any dependencies via the parameter. Ease of factory, ease of testing, reusability of code, and reduction of the amount of code are just some of the advantages that DI provides, as it is very widely used in modern app development. DI is most commonly used by provided libraries, as manual dependency injection requires a lot more code and attention to detail.

Dagger/Hilt

Dagger is a fully static compile time for Java/Kotlin in Android environment, Hilt is an additional layer built on top of it that allows for even easier integration and use in Android apps. Hilt is more commonly used due to providing the same functionalities of the Dagger but with improved ease of use, all while keeping the additional overhead minimal. Hilt is a part of the previously mentioned Android Jetpack. These dependency injections offer fast runtime performance, but have longer build times, which makes them easier to debug. [18]

Koin

Kotlin-based library focused on exploiting the advantages of Kotlin such as conciseness of code, all by using DSL (Domain-specific language). It offers shorter build times, which makes it harder to debug because of fewer checks. Runtime performance is somewhat slower than the one of Dagger/Hilt, but it doesn’t generate any additional code. Koin can only be used with files and components written in Kotlin, making it unusable for applications that do not use it. [19]

2.4.1.3. Singleton

Singleton is a pattern that specifies that only a single instance of a class should exist which can be accessed from every single point of the system. This pattern is often used when modeling real-world objects that are meant to have only one instance. Singleton classes are the most common network or database instances, where having multiple instances may cause data mixing and duplicate sources. It is a pattern that is present in almost every app, and it has virtually no disadvantages if used correctly.

2.4.1.4. Factory

Factory pattern is used to take care of all of the creational logic of an object. Factory class controls which object to instantiate and chooses between multiple choices of objects that might be created at that time. If the observed objects have remarkably similar attributes and functionalities, Factory pattern is used to determine which one is the most adequate for the situation, using additional information to make the choice. It also hides the creation logic from the client allowing him only to focus on the functionality of the required instance, rather than on the details of how to create it.

2.4.2. Structural patterns

Structural patterns are used to organize the details of classes and objects into familiar arrangements that perform typical tasks. By using these patterns, the inheritance in classes and adding functionalities to objects is greatly simplified.

2.4.2.1. Adapter

Adapter's main purpose is to adapt a certain type of data or a method so that it can be used with the other parts of the system. It is mostly useful when combining classes and functions that were not originally meant to be used in combination or when fetching data from multiple sources that often have different formats. Adapter is used as a wrapper for these types of objects and is used to "translate" the data/code to be compatible with other functions. [20]

The main advantage of the Adapter pattern is that there is no need to rewrite other functions, but just create an additional layer of translation. This reduces development time and allows the developers to get the data from multiple sources that have no common attributes.

When using Adapter pattern with classes, the result is a new interface that provides similar functionalities as the adapted class, whilst being compatible with the connecting class.

When using Adapter pattern with objects, it implements the interface of one of the objects and wraps the other one.

2.4.2.2. Decorator

Decorator's main purpose is to dynamically extend the object's functionalities at runtime. This is done by placing the object in a special wrapper object that contains new behaviors. This pattern allows the client to not use inheritance, since it only allows for a static change in the functionality and requires a whole other interface to be created, with classes being impossible to inherit twice.

Decorator, therefore, allows for an extension of the functionalities by just wrapping the object and is therefore sometimes called the *wrapper* pattern.

2.4.2.3. Façade

Facade provides a simplified interface to a library, a framework, or any other complex set of classes. It simplifies what could be a complex process of handling multiple interfaces and dependencies and having to worry about all of the method implementations. It is generally used to simplify the usage of some classes and functions that are overly complex and makes it easier to use for clients that don't know all of the insides of that particular function.

2.4.3. Behavioral patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. They define the behavior of certain components and the way they can communicate and share information with other components.

2.4.3.1. Observer

Observer pattern allows defining a subscription mechanism to notify multiple objects about any events that they are observing. It defines a one-to-many dependency between the objects. This pattern serves as a "notification" system and greatly simplifies the data update instead of multiple objects having to check every once in a while, whether the data has changed or instead of every object having a copy of the same data. The pattern is most often implemented in Listener classes and is found in nearly every app due to its simplicity and importance in the system.

Observable objects in Android will emit values and it was first introduced by the RxAndroid framework. Subscriber objects will listen and receive data updates as they arrive. More recently there has been a native way to implement this behavior called LiveData, which can be found in some of the newer apps.

2.4.3.2. State

A pattern that allows objects to alter their behavior on some event or change of some of the internal variables. State pattern is the base of the new Jetpack Compose UI tooling. Every time a "state" of the composable object is changed, the object is recomposed, and an action is run, which most often means that the UI changes.

MVVM architectural pattern also uses similar behavior, with ViewModels usually having multiple different states, which then affect how the UI and the model will behave.

State pattern has become increasingly popular in the last several years, with other exceedingly popular technologies outside the Android world, most notably React, having completely adopted the pattern and using it as the base of its behavior.

2.4.3.3. Iterator

A pattern that allows traversing through a collection of objects without exposing its underlying representation. This will allow the client to read and change all of the data objects in the collection in the same way, without knowing whether the collection is a list, an array, a tree, or any other type. All iterators usually implement the same interface, making them universally usable in the same way.

2.5. Android application components

Analysis description These components make the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter the app. Some of the components are interdependent. To understand how an Android app works and how each part of the app is connected to others, it is important to know what each of these components does. [21] [22]

A unique part of the Android system is that any component of another app can activate (or request to activate) any other app's components, allowing for the functionalities of certain apps to be reused.

Four major Android application components are Activities, Services, Providers, and Receivers, and all are defined in Android Manifest file together with the application specifications.

2.6. Other commonly used products and services

2.6.1. Google Firebase

Firebase is a set of hosting services for any type of application. Due to Google's association with the Android platform, it is mostly used in Android app development. It offers NoSQL and real-time databases, content, social authentication, notifications, and many other services. It was launched in 2012 and since then has grown in the number of services and usage around the community.

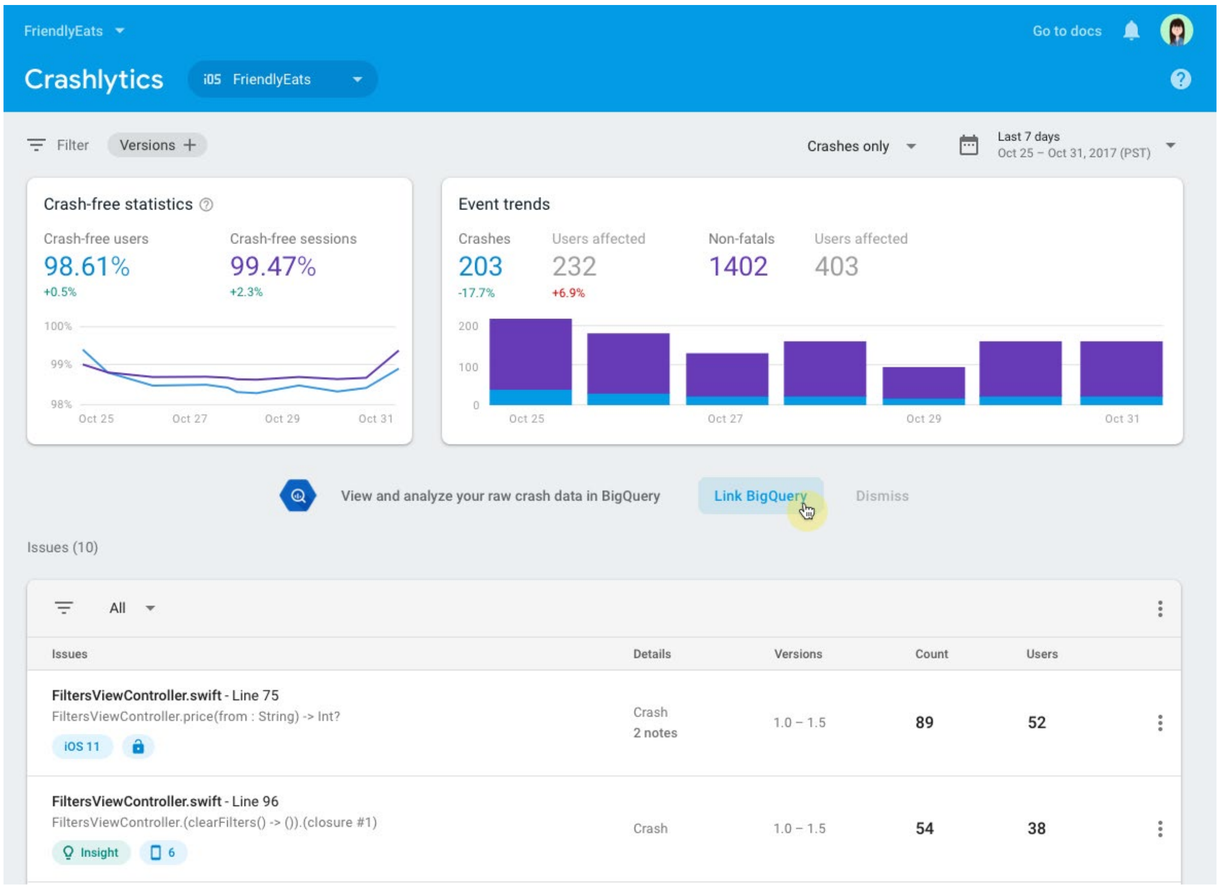
Firebase products are more commonly used in smaller applications due to their easy implementation and good documentation provided by Google. Most of the services are completely free of charge up to a certain point, which allows developers to improve quality and shorten development time in the initial stages of app development. Most of these services also scale with extreme ease to a bigger number of users, although in that case, they become somewhat pricey which makes them less attractive for applications that aim to have a large user base.

Although primarily intended to be used by smaller apps and by smaller development teams, due to its increased popularity, known stability, and continuous support by Google, they are also used by the commercial apps. [23] Firebase also provides hosting services that are being frequently used by many developers. [24]

2.6.1.1. Analytics and Crashlytics

App measurement solution that provides insight into app usage and user management. It is one of the most used Firebase products since it provides detailed data, most notably about user signups and app crashes/issues, completely free of charge.

Analytics integrates across Firebase features and provides reporting for up to five hundred distinct events. Analytics reports are also used for tracking user behavior and debugging the app.



Issues	Details	Versions	Count	Users
FiltersViewController.swift - Line 75 FiltersViewController.price(from : String) -> Int? iOS 11	Crash 2 notes	1.0 – 1.5	89	52
FiltersViewController.swift - Line 96 FiltersViewController.(clearFilters() -> ()).(closure #1) Insight 6	Crash	1.0 – 1.5	54	38

Figure 2.8: Crashlytics dashboard

2.6.1.2. Database (Realtime Database & Cloud Firestore)

Firebase offers two different services for storing data. [25] None of them are real SQL databases, which account for a vast majority of database systems in the industry, although the use of NoSQL databases is steadily increasing in recent years. [26]

A survey by *StackOverflow*, one of the most visited websites for developers, shows that NoSQL databases like Firebase and MongoDB are increasingly used every year. The most used database models in the industry (by ranking) are MySQL and PostgreSQL, which are real SQL databases.

- StackOverflow survey 2017 [27] – 21% used MongoDB
- StackOverflow survey 2018 [24] – 26% used MongoDB
- StackOverflow survey 2021 [28] – 28% used Mongo DB, 17% used Firebase

Firebase Realtime Database is a cloud-hosted NoSQL database that lets developers store and sync data between users in real-time. It is also optimized for offline use and features synchronization as soon as the device gets a connection. It is a document-based database that stores all of the data as one large JSON tree. Due to its simplicity and easy manipulation, it is a satisfactory solution for simple queries.

Cloud Firestore is a newer and more complex version of the Realtime Database. It is also document-based, although it offers a more complex structure, with both documents and collections that offer a more organized structure. Google recommends its usage for bigger systems that require better data organization.

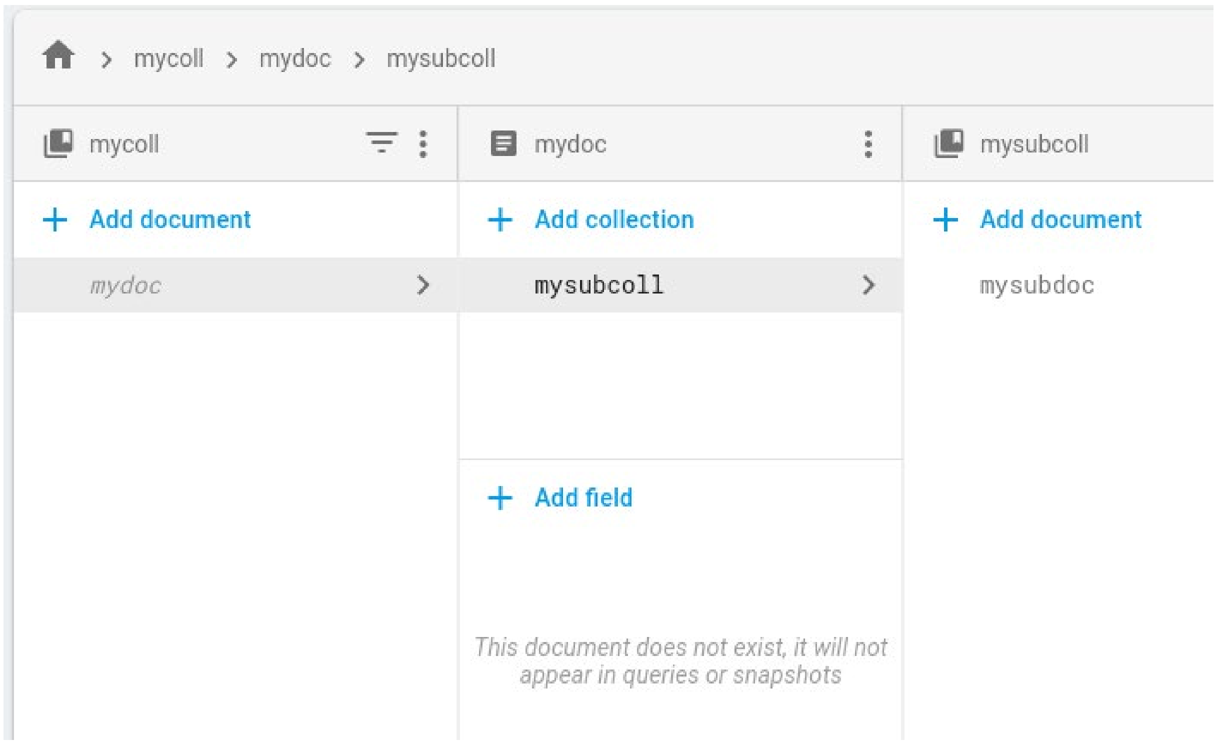

mycoll > mydoc > mysubcoll
mycoll
mydoc
mysubcoll
Add document
Add collection
Add document
mydoc
mysubcoll
mysubdoc
Add field
This document does not exist, it will not appear in queries or snapshots

Figure 2.9: Firebase Cloud Firestore example

Both databases offer exceedingly high uptime, relatively fast responses, and easy scalability. Many additional features and read/write rules can also be accessed via the Firebase Console. The biggest downsides of these databases are platform limitations and pricing which increase at higher usages. Other most used database systems in the industry are all SQL based with MySQL, PostgreSQL, and SQLite headlining the list.

2.6.1.3. Remote Config

Gives developers visibility and fine-grained control over the app's behavior and appearance without making app updates and releasing new versions. It allows for dynamic turning features on and off, personalization by audience segments, and running experiments on a smaller number of customers without setting up complex infrastructure. The most common use is the releasing of test features which can then be observed on a smaller number of users within a controlled environment.

2.6.1.4. Cloud Functions

Service that provides access to the serverless backend and allows triggering of distinct functions by actions on other Firebase products, like data changes in the databases, sign-ups via Firebase Auth, and events in Analytics. It is based on JavaScript functions that are being run in a Node.js environment that can be executed at specific times or after certain events. The advantage of Cloud Functions is also keeping logic hidden on the server side, with no direct code in the application files.

2.6.1.5. Authentication

Authentication service that allows for easy sign-up and login with a secure authentication system. The service supports multiple widely used platforms like Facebook, Twitter, GitHub, and more, as well as a simple email authentication. Accounts can be manipulated, merged, and control all from a single interface. It is also one of the most used Firebase services due to easy implementation, wide support, and an elevated level of security.

2.6.1.6. Cloud Messaging

Firebase Cloud Messaging (FCM) provides a reliable and battery-efficient connection between the server and devices that provide messages and notifications support for all platforms. The service also features predefined segments for different demographics and behavior groups, subscriptions to specific topics, and granularity. This service is one of the most used services when it comes to sending notifications.

During the analysis of the applications services by Firebase that are used by the apps, it has been noted to see how many developers have taken the "easy way out" of implementing pre-done services, sacrificing flexibility and cost of service for stability, security, and ease of implementation.

2.7. ProGuard (R8)

ProGuard is one of the technologies that is used to reduce the size of the app and to hide the code from being reverse engineered. The process is being done at compile-time. A newer version of ProGuard is called R8 and it is used in newer versions of Android Gradle from 3.4.0. [29]

To make your app as small as possible, you should enable shrinking in your release build to remove unused code and resources. When enabling shrinking, you also benefit from obfuscation, which shortens the names of your app's classes and members, and optimization, which applies more aggressive strategies to further reduce the size of the application.

The plugin handles the following compile-time tasks:

- **Code shrinking** - detects and safely removes unused classes, methods, attributes, and library dependencies
- **Resource shrinking** - removes unused resources from the packaged app including resources from library dependencies
- **Obfuscation** - shortens the names of classes and members which results in reduced DEX file sizes
- **Optimization** - inspects and rewrites code to further reduce the size of the app

When building the release version of the app, by default, R8 automatically performs the compile-time tasks described above. However, certain tasks can be disabled or customized through ProGuard rules files. R8 works with all of your existing ProGuard rules files, so updating the Android Gradle plugin to use R8 should not require a change of the existing rules. ProGuard and R8 can often be found in apps, and it is an industry-standard.

3 Analysis results

There are a total of twenty-seven applications that have been analyzed in this research. (**Table 3.1**) All of them are open-source apps with their repositories being located on GitHub. All of the apps have more than one thousand stars, which means that a fair amount of people are following their repositories and are interested in the development of that app, which is usually a good indicator that the repository is popular and provides some interesting content for the users. All of the apps have had at least one recent version release and pull request since 2020, while the vast majority are being updated regularly even to this day. None of the repositories have been archived up to October 2022.

Apps have initially been divided into categories based on what they are used for. Every app is listed with a description and a lot of other key details about the size, architecture, and everything that might be relevant to the analysis. A large table containing all of this information can also be found in the Appendix A.

Most of them were found on the Wikipedia page of open-source android applications [30] and searching different forums for GitHub repositories.

Table 3.1: Analyzed apps listed by category

App number	App name	App category
1	**Brave**	Browser
2	**DuckDuckGo**	Browser
3	**Fenix**	Browser
4	**Orbot**	Browser
5	**Bitwarden**	Commerical
6	**Kickstarter**	Commerical
7	**Shadowsocks**	Commerical
8	**Wikipedia**	Commerical
9	**Wordpress**	Commerical
10	**Antenna**	Media player
11	**NewPipe**	Media player
12	**Phonograph**	Media player
13	**Shuttle**	Media player
14	**Timber**	Media player
15	**K9**	Messaging and email
16	**QKSMS**	Messaging and email
17	**Signal**	Messaging and email
18	**Telegram**	Messaging and email
19	**Wire**	Messaging and email
20	**Google I/O**	Other
21	**Habitica**	Other
22	**Materialistic**	Other
23	**Muzei**	Other
24	**Omni Notes**	Other
25	**Kotlin Pokedex**	Tech demo
26	**NotyKT**	Tech demo
27	**Pokedex**	Tech demo

3.1. Analysis goals

The goal of the analysis of these apps is to see in which "real" state are the industry standard apps, not regarding what Google thinks is a standard. Some of the main questions presented by this analysis, divided into different categories, are the following:

- Application complexity and size
  - What is the average application install size?
  - What affects application size the most?
- Programming language
  - How many applications are using Kotlin or Java as a primary language?
  - How many applications use more than one language?
  - How many applications have made the transition from Java to Kotlin?
  - What is the number of Activities and Fragments in Kotlin-based and Java-based apps?
- User interface (UI)
  - Which technology is being used for the UI - Compose, or Views (XML)?
  - What is the number of Activities, Fragments, and screens in the respective technologies?
- Google libraries and services
  - How much do developers trust Google and their services?
  - Which non-native libraries and services are being used?
- Architecture and design patterns
  - What is the most common architectural pattern?
  - Which are the most common design patterns?
- Conclusion
  - How many years does it take for a certain technology to become widely used in the industry since becoming stable and what were the technologies that have become industry standard the quickest and why?
  - What is the future Android development?

3.2. Problems with certain metrics during the analysis

During the analysis phase several scenarios haven been encountered that have in some way affected the results. Most of these issues represent a parts of certain metric categories, meaning that the conclusions can still be drawn from other metrics that are in many ways connected.

3.2.1. Measuring source code size and the number of files

Most of the apps are modular and have several layers of code and components that support multiple platforms. This makes it difficult to completely make a distinction between which files are used in which modules. Therefore, some metrics regarding app size are not 100% correct. Thus, the main metric for determining the app size is the installation size on a real device.

3.2.2. Information about the first app version release

Some applications appeared on GitHub years after their release. This made tracking some older versions and real release dates more difficult. Regardless of the problem, the overall age of the app and the design choices that follow the era in which the app was made was relatively easy to determine based on other information.

3.2.3. Measuring the exact number of screens

Some applications have very deep screen trees and feature functionalities which are only accessible when the user is logged in or has a premium subscription. Also, counting the same type of screen that appears multiple times in an app with different data did not seem like the right choice. The number of screens in an app is therefore more of an approximation, although in many cases a correct one. There still may be some data that is not 100% correct, but this in no way influences the overall analysis and the conclusions regarding the complexity of the apps.

3.2.4. Counting dependencies

The continuation of the modularity problem. Counting the exact number of dependencies (libraries) that the app uses was problematic due to the several different modules that are included in an app, with every one of them having its own dependencies. Some source codes use diverse ways of bringing multiple different modules together, meaning that it is not always truly clear which exact submodules have been included with which dependencies. These numbers are not 100% correct, but they do not influence the analysis in any way.

3.3. Application analysis

In this section, the individual app results from the analysis are presented, together with short descriptions and post-analysis comments for every one of them.

3.3.1. Browsers

3.3.1.1. Brave

Description

Chromium-based browser that prioritizes user experience, speed, privacy, and a better advertisement system. One of the key features of the browser is blocking advertisements and website trackers, as well as providing optional ads to users in return for Basic Attention Tokens (BAT) cryptocurrency. It was reported to have nearly 60 million active users in August of 2022.

Post-analysis comment

Extremely complex multiplatform source-based environment. Exceedingly difficult to extract the data size and find the correct structure. Highly modular for different platforms uses a lot of C/C++ code through JNI.

Table 3.2: Brave analysis table

Category	Value
App size	Large
Languages used	Java and C/C++ (through JNI)
State	Active - Latest release in November 2022
First release /repository created	January 2012
Analyzed version	v.1.46.59 (October 2022)
Google Play downloads	100M+ (4.7)
GitHub repository stars	14.1k
Architecture	MVVM
Design patterns	Builders, Factories, Adapters
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	50+
Number of screens	20
Number of Activities	25
Number of Fragments	57
Uses Firebase services	No
Uses Room	No
Uses ProGuard/R8	Yes

3.3.1.2. DuckDuckGo

Description

Privacy-oriented browser (and a search engine) that emphasizes protecting searchers' privacy and avoiding the filter bubble of personalized search results.

Primarily made as a search engine it has also evolved into a browser on Android and iOS devices. It is one of the fastest-growing search engines on the market. The number of queries per day increasing exponentially since its release in 2008.

Post-analysis comment

Extremely complex multiplatform source-based environment, hybrid architecture, multiple layers ranging from backend to frontend.

Table 3.3: DuckDuckGo analysis table

Category	Value
App size	Medium
Languages used	Kotlin
State	Active - Latest release in November 2022
First release /repository created	December 2017
Analyzed version	v5.138.1 (October 2022)
Google Play downloads	10M+ (4.7)
GitHub repository stars	2.8k
Architecture	Hybrid (MVVM + MVP)
Design patterns	DI, Builders, Factories, Adapters, Decorators, Facades
Application components	Activities + Fragments, Services, Providers, Receivers
UI technology	Views (XML)
Depdendency injection	Dagger
Number of dependencies	50+
Number of screens	22
Number of Activities	40
Number of Fragments	42
Uses Firebase services	No
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.1.3. Mozilla Fenix

Description

Fenix (internal codename) is the all-new Firefox for Android browser, based on Mozilla Android Components. It is an open-source official Firefox clone coded from zero that uses newer technologies and patterns, to make for a better and cleaner code (and therefore app). It recently replaced Mozilla Firefox on Google Play and is one of the most active open-source Android app repositories on GitHub, with multiple commits on the main branch every day and more than 370 version releases. The old Firefox version was based on the older architectural patterns, used Java as a programming language, and Views for the UI instead of Jetpack Compose.

Post-analysis comment

Exceptionally clean and modern architecture, uses newest technologies patterns. Contains a neat folder and file structure, which is easy to observe and analyze. Usable, responsive, and a great app overall.

Table 3.4: Mozilla Fenix analysis table

Category	Value
App size	Large
Languages used	Kotlin
State	Active - Latest release in October 2022
First release /repository created	June 2019
Analyzed version	v105.1.0 (September 2022)
Google Play downloads (average rating)	100M+ (4.5)
GitHub repository stars	6.4k
Architecture	MVC + MVVM
Design patterns	Adapters, Builders, Decorators
Application components	Activities, Fragments, Services, Providers, Receivers
UI technology	Compose
Depdendency injection	No
Number of dependencies	118
Number of screens	18
Number of Activities	12
Number of Fragments	91
Uses Firebase services	Yes – Analytics, Crashlytics, and Cloud Messaging
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.1.4. Orbot

Description

Orbot is a freely licensed open-source application developed for the Android platform. It acts as a front-end to the Tor binary application, and also provides an HTTP Proxy for connecting web browsers and other HTTP client applications into the Tor SOCKS interface. It acts as an instance of the Tor network, free and open-source software for enabling anonymous communication, on such devices and allows traffic routing from a device's web browser, email client, map program, etc., through the Tor network, providing anonymity for the user.

Post-analysis comment

Modular and simple application with architecture made to support multiple device families.

Table 3.5: Orbot analysis table

Category	Value
App size	Small
Languages used	Java
State	Active - Latest release in October 2022
First release /repository created	March 2017
Analyzed version	v16.6.2 (July 2022)
Google Play downloads	10M+ (4.1)
GitHub repository stars	1.2k
Architecture	MVP
Design patterns	Adapters
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	25
Number of screens	5
Number of Activities	9
Number of Fragments	9
Uses Firebase services	No
Uses Room	No
Uses ProGuard/R8	Yes

3.3.2. Commercial applications

3.3.2.1. Bitwarden

Description

A password management service, stores sensitive information such as website credentials in an encrypted vault. Offers both a free cloud-hosted service as well as the ability to self-host. Client functionalities include 2FA login, biometric unlock, random password generator, password strength tool, and many other functionalities. It is one of the most popular password managers on the market.

Post-analysis comment

This app uses Xamarin which is a free cross-platform service that allows for the apps to be written in .NET and C# and programmed for both iOS and Android. Because of this, Bitwarden does not have traditional Android architecture and components like Activities and Services.

Table 3.6: Bitwarden analysis table

Category	Value
App size	Medium
Languages used	C# (Xamarin)
State	Active - Latest release October 2022
First release /repository created	August 2016
Analyzed version	v2022.10 (October 2022)
Google Play downloads	1M+ (4.5)
GitHub repository stars	4k
Architecture	MVVM
Design patterns	N/A
Application components	N/A
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	50+
Number of screens	6
Number of Activities	N/A
Number of Fragments	N/A
Uses Firebase services	Yes - Cloud Messaging
Uses Room	No
Uses ProGuard/R8	No

3.3.2.2. Kickstarter

Description

Kickstarter application for Android. Implemented with RxJava in logic filled with view models. Kickstarter as a service is a crowdfunding platform where users can present their projects and receive money from backers. It is the most visited and used crowdfunding platform on the market.

Post-analysis comment

A mix of modern and traditional architecture, a lot of well-known patterns have been used. The app seems to be stuck between transitioning to the new approach and staying in the old way of programming Android apps.

Table 3.7: Kickstarter analysis table

Category	Value
App size	Large
Languages used	Kotlin and Java
State	Active - Latest release October 2022
First release /repository created	February 2017
Analyzed version	v3.5.0 (September 2022)
Google Play downloads	1M+ (3.7)
GitHub repository stars	5.7k
Architecture	MVVM
Design patterns	DI, Builders, Factories, Adapters, Decorators
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	Dagger
Number of dependencies	50
Number of screens	24
Number of Activities	49
Number of Fragments	13
Uses Firebase services	Yes - Analytics + Crashlytics + Cloud Messaging
Uses Room	No
Uses ProGuard/R8	No

3.3.2.3. Shadowsocks

Description

High-performance cross-platform secured socks5 proxy. The main point of the services is to allow the user to surf the internet privately and securely. It is widely used in China to circumvent Internet censorship. The socks5 proxy is similar to an SSH (Secure Shell Tunnel), but unlike an SSH tunnel can also proxy UDP (User Datagram Protocol) traffic.

Post-analysis comment

Multiplatform modular app with maximum code reuse for different platforms, well organized and simple. Does not use that many design patterns and traditional Android architecture. The app has gone through the transition from Java to Kotlin and is now fully written in Kotlin.

Table 3.8: Shadowsocks analysis table

Category	Value
App size	Small
Languages used	Kotlin and C/C++ (JNI)
State	Active - Latest release September 2021
First release /repository created	June 2014
Analyzed version	v5.2.6 (September 2021)
Google Play downloads	5M+ (4.5)
GitHub repository stars	33.1k
Architecture	Hybrid
Design patterns	Adapters
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	27
Number of screens	6
Number of Activities	11
Number of Fragments	16
Uses Firebase services	Yes - Analytics and Crashlytics
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.2.4. Wikipedia

Description

Multilingual free online encyclopedia app, one of the most visited websites on the Internet. The official application allows you to access the entire content of one of the greatest sources of information on the Internet, just by making a few movements on the screen of your Android device. Features a simple interface that adapts the Wikipedia pages to mobile screens and has all of the same features as a full web page.

Post-analysis comment

Nicely organized and structured app. Very deep and modular structure, good code organization with an elevated level of integration of Google services and libraries.

Table 3.9: Wikipedia analysis table

Category	Value
App size	Small
Languages used	Kotlin
State	Active - Latest release November 2022
First release /repository created	January 2012
Analyzed version	v2.7 (October 2022)
Google Play downloads	50M+ (4.5)
GitHub repository stars	1.8k
Architecture	MVVM
Design patterns	Builders, Factories, Adapters, Facades
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	50+
Number of screens	9
Number of Activities	47
Number of Fragments	42
Uses Firebase services	Yes - Cloud Messaging
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.2.5. WordPress

Description

Free and open-source content management system (CMS) written in hypertext processor language and paired with a MySQL or MariaDB database with supported HTTPS. It allows for easy and efficient website creation, editing, and management. The Mobile version supports the creation and editing of posts and pages, photos and videos upload and managing user communication.

Post-analysis comment

Combined and messy architecture combining old Java way of programming and new Android Jetpack. Featuring a lot of design patterns, the app feels like it is in a transition phase between the old Java and the new Kotlin ways of Android development.

Table 3.10: WordPress analysis table

Category	Value
App size	Large
Languages used	Kotlin and Java
State	Active - Latest release November 2022
First release /repository created	December 2015
Analyzed version	v20.9 (October 2022)
Google Play downloads	10M+ (4.4)
GitHub repository stars	2.7k
Architecture	MVVM
Design patterns	Builders, DI, Factories, Adapters, Decorators, Facades
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	Hilt
Number of dependencies	25
Number of screens	50+
Number of Activities	99
Number of Fragments	167
Uses Firebase services	Yes - Remote Config and Cloud Messaging
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.3. Media players

3.3.3.1. Antenna

Description

Easy-to-use, flexible, and open-source podcast manager for Android. Gives access to millions of free and paid podcasts from independent podcasters to large publishing houses like BBC, NPR, and CNN. It allows streaming and downloading of all of the podcasts that can be found on the iTunes podcast database or other places on the Internet. It is completely free and features no ads.

Post-analysis comment

Features a highly modular and reusable architecture, does not follow any traditional architectural patterns.

Table 3.11: Antenna analysis table

Category	Value
App size	Small
Languages used	Java
State	Active - Latest release October 2022
First release /repository created	February 2014
Analyzed version	v2.7.1 (October 2022)
Google Play downloads	500K+ (4.7)
GitHub repository stars	4.5k
Architecture	Hybrid
Design patterns	Builders, Factories, Adapters, Decorators, Facades
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	46
Number of screens	20
Number of Activities	11
Number of Fragments	45
Uses Firebase services	No
Uses Room	No
Uses ProGuard/R8	Yes

3.3.3.2. NewPipe

Description

A libre lightweight streaming front-end for Android. Does not use any Google framework libraries or YouTube APIs, making it usable on devices without installed Google Services. It supports services like YouTube, YouTube Music, PerrTupe, Bandcamp, SoundCloud, and media.ccc.de. The functionalities of the app range from simple music playing to watching 4K videos, listening to audio in the background, picture-in-picture mode, live streams, and much more.

Post-analysis comment

A highly modular app that successfully avoids the usage of any Google-based external service, uses a lot of distinctive design patterns, and has an exceptionally clean architecture based on MVVM.

Table 3.12: NewPipe analysis table

Category	Value
App size	Small
Languages used	Java
State	Active - Latest release November 2022
First release /repository created	September 2015
Analyzed version	v0.24.0 (October 2022)
Google Play downloads	50K+ (1.7)
GitHub repository stars	21.9k
Architecture	MVVM
Design patterns	Builders, Facades, Adapters
Application components	Activities + Fragments, Services
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	47
Number of screens	10
Number of Activities	12
Number of Fragments	31
Uses Firebase services	No
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.3.3. Phonograph

Description

Material Design-based music player made with the premises to look and feel good. Features simple music player capabilities and a highly customizable UI.

Post-analysis comment

Exceptionally clean code and code structure. Simple, good-looking, and well-functioning app.

Table 3.13: Phonograph analysis table

Category	Value
App size	Small
Languages used	Java
State	Semi-active - Latest release October 2021
First release /repository created	April 2017
Analyzed version	v1.3.5 (September 2020)
Google Play downloads	1M+ (3.8)
GitHub repository stars	2.7k
Architecture	MVP
Design patterns	Builders, Facades, Adapters
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	33
Number of screens	8
Number of Activities	14
Number of Fragments	20
Uses Firebase services	No
Uses Room	No
Uses ProGuard/R8	Yes

3.3.3.4. Shuttle

Description

An open-source, local music player for Android. Shuttle comes in two versions - Shuttle (free) and Shuttle+. The basic version includes features like local playback, equalizer, sleep timer, widgets, and artwork scraping, with the paid Shuttle+ version including additional options like Chromecast support and theming.

Post-analysis comment

Nicely done application based on the MVP architecture by the textbook. Using a lot of design patterns with a perfect folder and file structure, making analysis and development a lot simpler.

Table 3.14: Shuttle analysis table

Category	Value
App size	Small
Languages used	Java and Kotlin
State	Not active - Latest release July 2020
First release /repository created	March 2017
Analyzed version	v2.0.17 (July 2020)
Google Play downloads	1M+ (4.3)
GitHub repository stars	2.2k
Architecture	MVP
Design patterns	DI, Builders, Adapters, Facades
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	Dagger
Number of dependencies	50
Number of screens	19
Number of Activities	9
Number of Fragments	16
Uses Firebase services	Yes – Remote Config
Uses Room	No
Uses ProGuard/R8	Yes

3.3.3.5. Timber

Description

Material theme music player that works across multiple platforms (phones, wear, auto, cast, assistant). This music player's strong point is a Material Design-style interface that can be customized at will. The player supports MP3 and FLAC files and features a quite simple audio equalizer.

Post-analysis comment

Exceptionally clean code and code structure. A simple, functional, and lightweight application.

Table 3.15: Timber analysis table

Category	Value
App size	Small
Languages used	Java
State	Not active - Latest release October 2020
First release /repository created	January 2016
Analyzed version	v1.7 (October 2020)
Google Play downloads	100K+ (4.1)
GitHub repository stars	6.8k
Architecture	MVP
Design patterns	Builders, Factories, Adapters
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (Compose)
Depdendency injection	No
Number of dependencies	22
Number of screens	11
Number of Activities	9
Number of Fragments	15
Uses Firebase services	Yes - Crashlytics
Uses Room	No
Uses ProGuard/R8	Yes

3.3.4. Messaging and email

3.3.4.1. K9

Description

Email app for Android that works with basically every email provider. Designed as an alternative to the stock email clients included with the platform. Supports POP3 and IMAP protocols and IMAP IDLE for real-time notifications.

Post-analysis comment

Overly complex architecture, a lot of modular files with different architectures in different modules. The backend layer and the frontend layer are very clearly divided.

Table 3.16: K9 analysis table

Category	Value
App size	Small
Languages used	Java and Kotlin
State	Active - Latest release in July 2022
First release /repository created	January 2014
Analyzed version	v6.202 (July 2022)
Google Play downloads	5M+ (2.8)
GitHub repository stars	7.4k
Architecture	MVVM
Design patterns	DI, Builders, Factories, Adapters, Facades
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	View (XML)
Depdendency injection	Koin
Number of dependencies	19
Number of screens	30
Number of Activities	30
Number of Fragments	42
Uses Firebase services	No
Uses Room	No
Uses ProGuard/R8	Yes

3.3.4.2. QKSMS

Description

SMS messenger for Android is aiming to replace the stock version. Offering a clean and customizable design and theming, with support from SMS and MMS messages, delayed message sending, and group messaging.

Post-analysis comment

Unusual architecture featuring all three architectural patterns (MVC, MVVM, MVP). Highly modular with modules divided based on their role to presentation, model, domain, and common layers.

Table 3.17: QKSMS analysis table

Category	Value
App size	Small
Languages used	Kotlin and Java
State	Semi-Active - Latest release in February 2021
First release /repository created	December 2015
Analyzed version	v3.9.4 (February 2021)
Google Play downloads	1M+ (4.0)
GitHub repository stars	3.9k
Architecture	Hybrid (MVVM, MVP, MVC)
Design patterns	DI, Builders, Facades, Adapters, Factories
Application components	Activities, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	Dagger
Number of dependencies	18
Number of screens	10
Number of Activities	12
Number of Fragments	0
Uses Firebase services	Yes - Crashlytics
Uses Room	No
Uses ProGuard/R8	Yes

3.3.4.3. Signal

Description

A messaging app for simple private communication with friends. Signal uses phone data connection (WiFi, 3G, 4G) to communicate securely, optionally supports plain SMS/MMS to function as a unified messenger and can also encrypt the stored messages on the phone.

Post-analysis comment

Incredibly detailed and complex modular folder structure and a complex architecture based on MVP. Advanced level of programming using a lot of distinctive design and architectural patterns.

Table 3.18: Signal analysis table

Category	Value
App size	Large
Languages used	Java and C/C++ (JNI)
State	Active - Latest release November 2022
First release /repository created	October 2013
Analyzed version	v5.53.2 (October 2022)
Google Play downloads	100M+ (4.4)
GitHub repository stars	23k
Architecture	MVP
Design patterns	Builders, DI, Factories, Adapters, Decorators
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	Yes - Dagger
Number of dependencies	60
Number of screens	23
Number of Activities	73
Number of Fragments	118
Uses Firebase services	Yes - Analytics and Cloud Messaging
Uses Room	No
Uses ProGuard/R8	Yes

3.3.4.4. Telegram

Description

Telegram is a messaging app with a focus on speed and security, it is super-fast, simple, and free. Telegram can be used on multiple devices at the same time — messages synchronize seamlessly across any number of phones, tablets, or computers. Telegram has over seven hundred million monthly active users and is one of the ten most downloaded apps in the world.

Post-analysis comment

Complex app that uses a lot of C/C++ code through JNI. An exceedingly high number of Activities compared to the number of screens.

Table 3.19: Telegram analysis table

Category	Value
App size	Medium
Languages used	Java and C/C++ (JNI)
State	Active - Latest release in November 2022
First release /repository created	December 2019
Analyzed version	1000M+ (4.3)
Google Play downloads	21k
GitHub repository stars	v8.8.2 (June 2022)
Architecture	MVC
Design patterns	Builders, Factories, Adapters, Facades
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	26
Number of screens	19
Number of Activities	103
Number of Fragments	4
Uses Firebase services	Yes - Remote Config and Cloud Messaging
Uses Room	No
Uses ProGuard/R8	Yes

3.3.4.5. Wire

Description

Wire is the most secure collaboration platform. It claims to increase the productivity in the team while keeping the information private. Wire allows its users to communicate and share information easily and securely - messages, files, conference calls or private conversations - always in context.

Post-analysis comment

Simple and well-organized messaging app, with a lot of different build variants, allowing for easy version control and testing.

Table 3.20: Wire analysis table

Category	Value
App size	Medium
Languages used	Scala, Java, and Kotlin
State	Active - Latest release in August 2022
First release /repository created	August 2016
Analyzed version	v3.82.38 (August 2022)
Google Play downloads	1M+ (2.9)
GitHub repository stars	2.5k
Architecture	MVC
Design patterns	DI, Factories, Adapters, Decorators, Facades
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	Koin
Number of dependencies	50+
Number of screens	12
Number of Activities	11
Number of Fragments	5
Uses Firebase services	Yes - Cloud Messaging
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.5. Other

3.3.5.1. Google I/O

Description

An app used for the Google I/O developer conference with several days of deep technical content featuring technical sessions and hundreds of demonstrations from developers. The app displays a list of conference events - sessions, office hours, app reviews, code labs - and allows the user to filter these events by types and topics. Users can also star certain events and reserve a seat. The app was initially used for the 2019 conference, and even though the following two conferences have been canceled, it was still updated with the newest technologies and improvements.

Post-analysis comment

Modular architecture, divided folders and different builds for different purposes, exceptionally clean and consistent code. The app includes some of the newest Google libraries like LiveData and DataStore, trying to stay up to date with the newest releases. App also has a partial integration with Google Compose, which is integrated on another branch and is not a part of the official release.

Table 3.21: Google I/O analysis table

Category	Value
App size	Small
Languages used	Kotlin
State	Semi-active - Latest release in 2021
First release /repository created	February 2016
Analyzed version	v2021 (2021)
Google Play downloads	1M+ (4.3)
GitHub repository stars	21.7k
Architecture	MVVM
Design patterns	DI, Adapters
Application components	Activities + Fragments, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	Hilt
Number of dependencies	7
Number of screens	6
Number of Activities	5
Number of Fragments	34
Uses Firebase services	Yes
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.5.2. Habitica

Description

Habit building program which treats your life like a Role-Playing Game. It allows the user to level up as it succeeds, lose HP as it fails, and earn money to buy weapons and armor. It looks and feels like a video game, but it replicates real-work actions from the user, all to make him work harder and better in real-life to get in-game rewards.

Post-analysis comment

Post-analysis comment: Another one of the apps that made a nearly full transition from Java to Kotlin. The architecture of the app is extremely well organized and clean, allowing for multi-layer type organization and structure which makes components easier to find and maintain.

Table 3.22: Habitica analysis table

Category	Value
App size	Medium
Languages used	Kotlin and Java
State	Active - Latest release September 2022
First release /repository created	November 2015
Analyzed version	v4.0.2 (September 2022)
Google Play downloads	1M+ (4.2)
GitHub repository stars	1.1k
Architecture	MVVM
Design patterns	DI, Builders, Factories, Adapters, Facades
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	Hilt
Number of dependencies	46
Number of screens	50+
Number of Activities	29
Number of Fragments	67
Uses Firebase services	Yes - Analytics, Remote Config, and Crashlytics
Uses Room	No
Uses ProGuard/R8	Yes

3.3.5.3. Materialistic

Description

Material design Hacker news client for Android that uses official HackerNews/API, Dagger for dependency injection, and Roboelectric for unit testing.

Post-analysis comment

Unstructured files and folders, messy architecture, and app. No modularity, everything is in the same folder, offers no real support for any new developers wanting to improve the app.

Table 3.23: Materialistic analysis table

Category	Value
App size	Small
Languages used	Java
State	Not active - Latest release March 2019
First release /repository created	April 2016
Analyzed version	v3.3 (March 2019)
Google Play downloads	N/A
GitHub repository stars	2.2k
Architecture	MVP
Design patterns	DI, Builders, Adapters, Facades
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	Dagger
Number of dependencies	20
Number of screens	16
Number of Activities	23
Number of Fragments	6
Uses Firebase services	No
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.5.4. Muzei

Description

Muzei is a live wallpaper that gently refreshes your home screen each day with famous works of art. It also recedes into the background, blurring and dimming artwork to keep your icons and widgets in the spotlight. Simply double touch the wallpaper or open the Muzei app to enjoy and explore the artwork in its full glory. It also allows setting your favorite photos in the background.

Post-analysis comment

Exceptionally clean architecture, simple and modular client for multiple platforms.

Table 3.24: Muzei analysis table

Category	Value
App size	Small
Languages used	Kotlin, Python, and Java
State	Active - Latest release January 2022
First release /repository created	February 2014
Analyzed version	v3.4.1 (January 2022)
Google Play downloads	1M+ (4.1)
GitHub repository stars	4.4k
Architecture	MVVM
Design patterns	Builders, Factories, Adapters
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	17
Number of screens	7
Number of Activities	11
Number of Fragments	16
Uses Firebase services	Yes - Analytics + Crashlytics
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.5.5. OmniNotes

Description

A note-taking application aimed to have a simple interface while keeping smart behavior. The application expands on the generic note-taking features of other basic applications and allows users to attach image and video files, use a variety of widgets, tag and organize notes, search through notes, and customize the application's UI.

Post-analysis comment

Clean and organized architecture. Smooth and fast application with a lot of listeners, helpers, and providers. Genuinely nice application to use that offers quite a few functionalities and does it well.

Table 3.25: OmniNotes analysis table

Category	Value
App size	Small
Languages used	Java
State	Active - Latest release March 2022
First release /repository created	August 2015
Analyzed version	v6.1.0 (March 2022)
Google Play downloads	100K+ (4.0)
GitHub repository stars	2.5k
Architecture	MVP
Design patterns	Factories, Adapters
Application components	Activities + Fragments, Services, Receivers, Providers
UI technology	Views (XML)
Depdendency injection	No
Number of dependencies	40
Number of screens	5
Number of Activities	13
Number of Fragments	8
Uses Firebase services	No
Uses Room	No
Uses ProGuard/R8	Yes

3.3.6. Tech demo

3.3.6.1. Kotlin Pokedex

Description

Pokedex app build with Kotlin that uses most of the latest technologies from Android, such as LiveData, Navigation, Room, and Databinding. Pokedex allows for quick and easy Pokemon search, fetching the Pokemon details, seeing their evolutions and other connections, as well as seeing some news from the Pokemon world.

Post-analysis comment

The app has all of the modern android development features but Compose and is intended more as a demo app to showcase all of the latest features of Android development. On GitHub, another similar version of the app can be found that uses Jetpack Compose, but the level of quality of the app architecture and the number of stars/downloads were not enough to be included in this list.

Table 3.26: Kotlin Pokedex analysis table

Category	Value
App size	Small
Languages used	Kotlin
State	Semi-active - Latest release in May 2020
First release /repository created	February 2020
Analyzed version	v2020 (May 2020)
Google Play downloads	N/A
GitHub repository stars	1.3k
Architecture	MVVM
Design patterns	DI, Factories, Adapters
Application components	Activities
UI technology	Views (XML)
Depdendency injection	Koin
Number of dependencies	26
Number of screens	2
Number of Activities	1
Number of Fragments	0
Uses Firebase services	No
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.6.2. NotyKT

Description

A complete Kotlin-stack note-taking application built to demonstrate the use of Kotlin programming language in server-side and modern android development tools.

Post-analysis comment

A complete Kotlin-stack application built to demonstrate the use of modern android development tools with best practices. There are two versions of the app - the regular that uses Views and Compose version. The two versions are mostly using the same components and share all of the same characteristics. When using the app, the difference between them is almost nonexistent. A deeper analysis of the two versions is done in Chapter 6 of this work.

Table 3.27: NotyKT analysis table

Category	Value
App size	Small
Languages used	Kotlin
State	Active - Latest release October 2022
First release /repository created	October 2020
Analyzed version	v2.1.0 (October 2022)
Google Play downloads	100M+ (4.5)
GitHub repository stars	1.4k
Architecture	MVVM
Design patterns	DI, Builders, Factories, Adapters
Application components	Activities + Fragments
UI technology	Two versions - Compose and Views (XML)
Depdendency injection	Hilt
Number of dependencies	28 (38 Compose version)
Number of screens	6
Number of Activities	1
Number of Fragments	8 (0 Compose version)
Uses Firebase services	No
Uses Room	Yes
Uses ProGuard/R8	Yes

3.3.6.3. Pokedex

Description

A Pokedex application written in Kotlin that demonstrates modern Android development using all of the most popular and modern Android tools like Material Motion, Coroutines, Flow, and Jetpack, all based on MVVM architecture.

Post-analysis comment

Post-analysis comments: A modern app with a lot of dependencies and external libraries that are often used in the industry. Features a clean code and solid architecture, a very responsive UI, and overall, a very solid app.

Table 3.28: Pokedex analysis table

Category	Value
App size	Small
Languages used	Kotlin
State	Active - Latest release August 2021
First release /repository created	December 2019
Analyzed version	v1.1.0 (August 2021)
Google Play downloads	N/A
GitHub repository stars	6.1k
Architecture	MVVM
Design patterns	DI, Factories, Adapters
Application components	Activities, Providers
UI technology	Views (XML)
Depdendency injection	Hilt
Number of dependencies	27
Number of screens	3
Number of Activities	4
Number of Fragments	0
Uses Firebase services	No
Uses Room	Yes
Uses ProGuard/R8	Yes

4 Notable selected applications

Some of the applications from the analyzed set offer an interesting approach whether it comes to architectural patterns, UI design, modular code, or something else. In this chapter, these apps will be analyzed in depth to try and explain what makes them stand out and are of special interest to this research. Six applications have been selected for this chapter.

4.1. Mozilla Fenix (Firefox)

Fenix app is of great interest due to it being the only major app that uses Compose and is completely influenced by the most modern Android development libraries and services.

It originally started as a project of rewriting an entire Mozilla Firefox app for Android by using Kotlin and Compose, the opposite of the previously used technologies of Java and Views. After years of development, Fenix was finally completed and replaced the previous version of Firefox completely in 2022.

Fenix is an all-new Firefox browser based on GeckoView and Mozilla Android Components. GeckoView is a tool that allows apps to use the entire power of the Web in their applications, providing the functionalities of a full WebView through an API, but offering many more features than the default Android built-in WebView. Mozilla Android Components is a collection of independent, reusable Android library components to make it easier to build browsers and browser-like applications.

Fenix also implements several different build variants, which allows the developers to launch different versions of the app through different channels:

- Debug - default variant for developers, allows debugging and adds tools like LeakCanary for troubleshooting
- Nightly - used to ship Firefox Nightly, minor version updates with small features that ship on a nightly basis, using GeckoView Nightly
- Beta - a more stable version of Nightly, featuring more new features and being released less often

- Release - full release versions of the app, fully stable, released even less often than beta

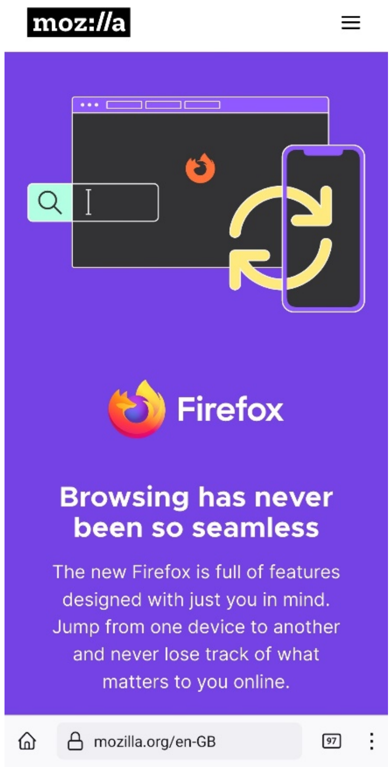
This approach allows Mozilla developers to easily test out new features, receive and test pull requests from other developers, and have continuous development without affecting the users that don't want to be affected. What allowed this to exist is the modular structure of the app that easily allows the developers to include or exclude some parts of the project from the build version.

Fenix folder structure is feature-based, which means the folders are organized so that the files inside do similar tasks. Since it is using Compose, it is only natural that Fenix is based on MVVM architecture, which goes together with this UI technology due to its state-based functionalities. It still uses some Controllers and has some elements of MVC, but since MVVM naturally expands and inherits MVC architecture, it is expected for such a complex app to use both architectures to match its needs.

Despite being such a large and complex app, Fenix only uses twenty-nine different composable elements, meaning that most of the elements are being reused within the app. This allows developers to easily switch a certain style or functionality of a component since it has to be changed only once to be reflected in all the places throughout the app, a feature that does not exist in the Views approach.

The Mozilla team takes a serious approach when it comes to code stability and efficiency. Within the source folder, additional two modules can be found that are specifying the use of Lint and Rekt, which are both used for checking and notifying the developer about potential bugs in the code. Their proper use often leads to much better, cleaner, and more efficient code, along with reducing the app size.

The application itself runs very cleanly, without any jitter or crashes, and is a good example of how a good, clean, and organized code can lead to having a better app and providing the users with a great experience.



Browsing has never been so seamless

The new Firefox is full of features designed with just you in mind. Jump from one device to another and never lose track of what matters to you online.

Figure 4.1: Mozilla Fenix (Firefox) on Android

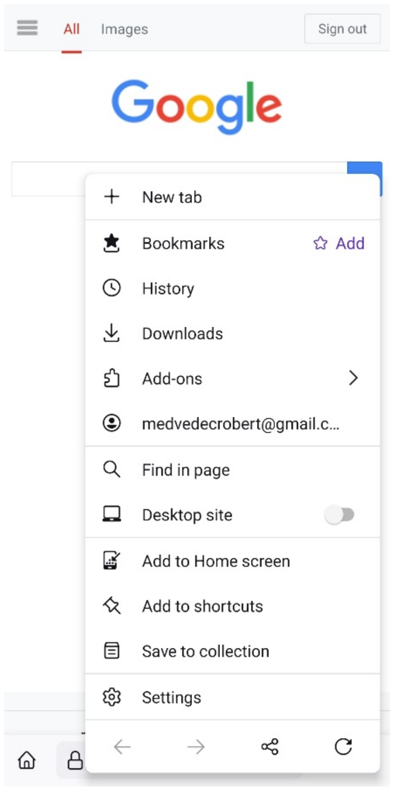


Figure 4.2: Mozilla Fenix (Firefox) settings on Android

4.2. Muzei

Muzei is a quite simple, lightweight, and unique app, which has little interaction with the user but can change the whole mobile experience. It dynamically changes the background of a phone and replaces it with a new picture of a famous painting from the world's famous museums every day. The functionality also exists as an API, making it possible to be implemented into other apps as well.

What makes Muzei interesting for this analysis is the way it uses modularity to achieve multi-device support. The main folder structure is divided into thirteen different submodules. They are the following:

- android-client-common
- example-unsplash
- example-watchface
- extensions
- el-wallpaper
- legacy-common
- legacy-standalone
- main
- muzei-api
- source-featured-art
- source-gallery
- source-single
- wearable

The main submodule is a module that holds the base of the code and is included in all of the app versions. The muzei-api submodule is used only for the API that can be included by other apps.

Legacy modules are used for some older versions of the app that are only supported on some older devices for some architectural reasons. Source modules define from where the background images are being fetched, whether from a museum collection, user gallery, or is it just a single photo.

Android-client-common and wearable submodules hold most of the functionalities for their respective platforms of Android and Wear OS.

This structure is a good example of a preferred structure for multi-device support. All of the app versions will have a main submodule included, but the mobile app will not have a wearable submodule, the same way a wearable app will not have android-client-common. The API itself will then be minimal for the same reason, making it easy for other developers to include it in their projects due to such a small size.

Each of those submodules has its own gradle file with defined dependencies. This allows these submodules to only implement certain dependencies when they are being used, meaning that the app size and the app compilation time will be much smaller when it is being used on a certain family of devices since not all of the dependencies are being included.

Ultimately, this also means much less code since the code itself is being reused. When a certain class of functionality is needed from another submodule, the included submodule takes up much less space than having to include all of the classes, especially considering that submodules contain classes and methods that are generally used together.

Muzei is therefore one of the smallest analyzed apps, despite having a sizable number of code lines and dependencies, as well as providing more functionalities than some other apps that take up much more space on the phone. Not to mention it runs very smoothly and does perfectly for what is intended.

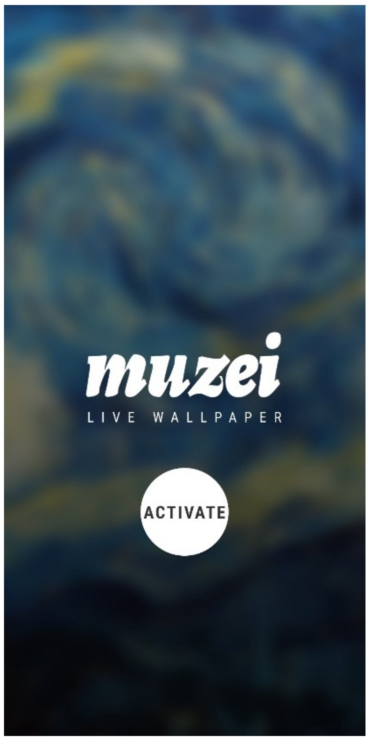


Figure 4.3: Muzei on Android #1

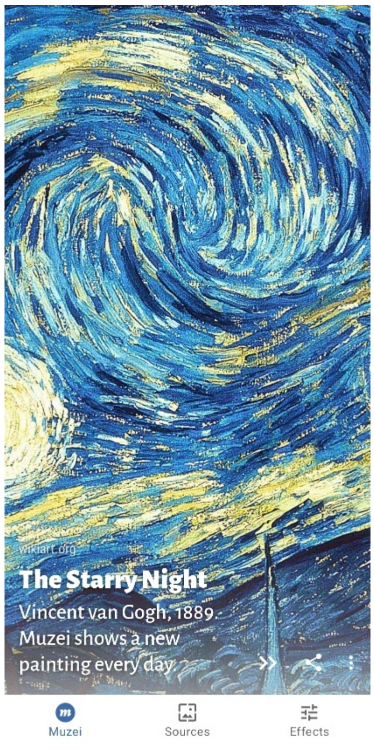


Figure 4.4: Muzei on Android #2

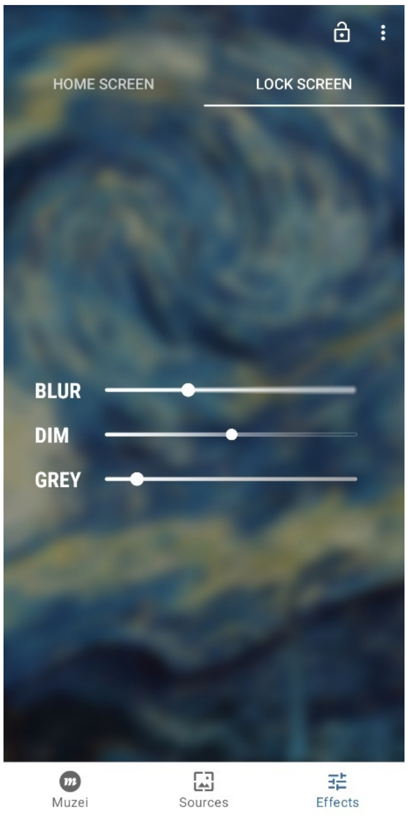


Figure 4.5: Muzei on Android #3

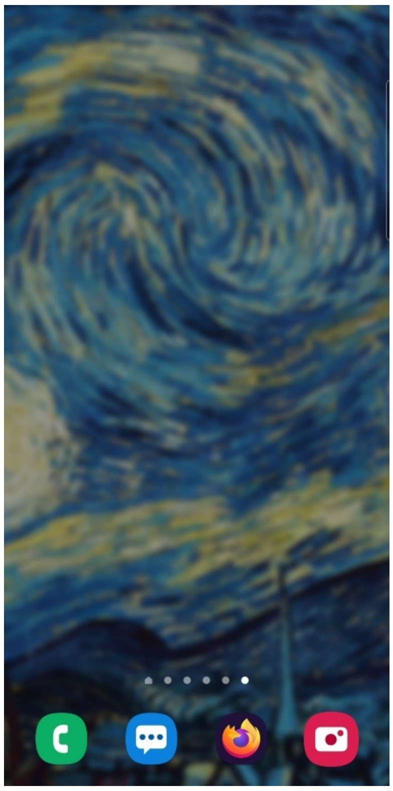


Figure 4.6: Muzei on Android #4

4.3. Kickstarter and WordPress

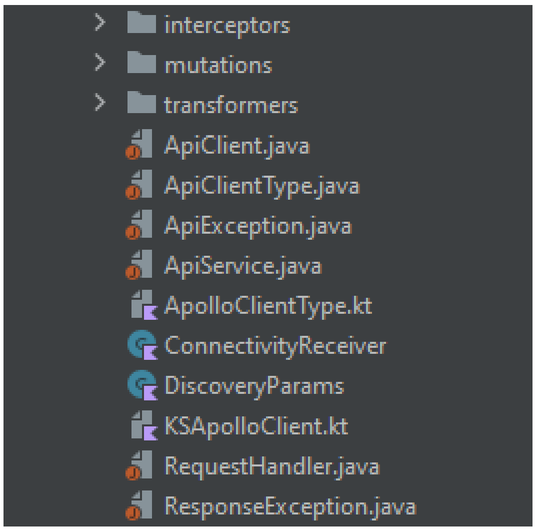
Kickstarter and WordPress applications represent two remarkably similar examples. Both are mobile versions of services that are primarily meant to be accessed via the web and offer limited functionalities compared to their web versions. Both are commercially used apps with a large number of users and very advanced functionalities. Both have overly complex architecture and folder structures. And finally, both are currently in a transition phase, moving from the old ways of Android development in Java using older patterns and making a move to Kotlin and new development ways.

Most of the code in both of these apps is in Kotlin, although some major parts are still written in Java. The folder and file structure of these apps are very messy, with files in the same folder being written in two different functionalities and often interpolating. While this works due to the interoperability of Java and Kotlin it can be very frustrating for a developer having to switch the syntax and the way of thinking very often when editing and rewriting some code.

Both of the apps have made a transition to MVVM architecture, which brings them one step closer to modern ways of development. Since all of those ViewModel classes are written in Kotlin, it can be assumed that the change of architecture came at the same time or after the decision to change the primary language.

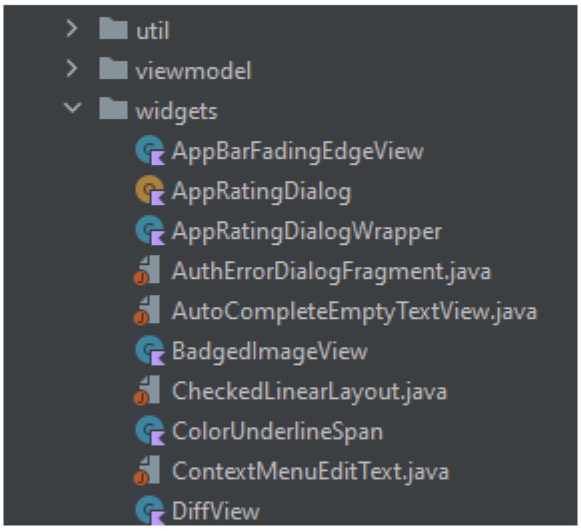
Neither of these applications has submodules, making it even more difficult for developers to change specific parts of the subsystem and potentially slowing down the transition. That is why it is unlikely they will make a full transition to Compose since doing that with such a messy code structure would be a very brave move that could potentially backfire.

In these code bases, only one gradle file is present, which limits developers' approach options when it comes to testing and trying out new apps. These apps in their current state offer a particularly good look at how refactoring and code maintenance can be a messy process if the early phase of the development and project setup is not done in a recommended way. These statements do not necessarily have to reflect on the user experience, but the fact that they are by far the biggest apps in their category and one of the biggest apps in total may deflect users from using them.



```
> interceptors
> mutations
> transformers
  ApiClient.java
  ApiClientType.java
  ApiException.java
  ApiService.java
  ApolloClientType.kt
  ConnectivityReceiver
  DiscoveryParams
  KSApolloClient.kt
  RequestHandler.java
  ResponseException.java
```

Figure 4.4: Mix of Kotlin and Java files in Kickstarter source code



```
> util
> viewmodel
v widgets
    AppBarFadingEdgeView
    AppRatingDialog
    AppRatingDialogWrapper
    AuthErrorDialogFragment.java
    AutoCompleteEmptyTextView.java
    BadgedImageView
    CheckedLinearLayout.java
    ColorUnderlineSpan
    ContextMenuEditText.java
    DiffView
```

Figure 4.3: Mix of Kotlin and Java files in WordPress source code

4.4. NotyKT - Views and Compose version

One of the analyzed applications comes in two versions, which are in practice almost identical but offer slightly diverse backgrounds and technologies. One has a UI made by using Jetpack Compose and another one by using Views with XML. Even though it is an amazingly simple and lightweight app, it can still provide some answers on what the main differences between the two UI technologies are when everything else around it is mostly the same.

4.4.1. Shared components

The app itself has a modular architecture that allows both versions of the app to reuse the same code within the same files. The app is based on the MVVM architecture.

Shared components of the app are task manager, connectivity components, repository, utilities, dependency injection modules, and all of the view models with states. More than 50% of the app files are shared and reused, meaning that the apps are not only similar in practice but also have a vast majority of the code base identical.

Shared dependencies include web communication library Retrofit, local database Room, some basic androidx libraries, and Hilt dependency injection library.

Apps also share some of the resources like strings, values, and images.

4.4.2. Individual components

The Views version of the app uses the more traditional approach of using Fragments. Since the app only has one Activity, Fragments act like individual screens or small components. There are six screens in total, with eight Fragments, that besides the screens also display two additional sub-components that appear on user action. Other individual components are also two dialogs, one adapter, and resources, mostly XML layout files that are used to represent the UI.

The Compose version of the app uses a more modern approach and does not have any additional Fragments. Everything is accessed by activity. This version has twenty-four composable elements, with additional theming and utility files, but has no extra resources due to them being presented by composables. The file count is twice as big as with the other version, as the structure is even more modular than before.

While apps do feel and behave the same, while only having minimal visual differences, behind the scenes some of the aspects are not the same. The difference in the size of the app is quite different, with the Compose version being more than twice as big as the Views with XML version. This is most likely due to a lot more dependencies that the Compose version has as the libraries that include Compose are not included by default. The app is not quite complex enough to notice any significant performance differences, but it does bring to the conclusion that Compose apps tend to be somewhat bigger than their Views with XML counterparts.

Table 4.1: NotyKT application XML and Compose version comparison table

NotyKT app	Views (XML) version	Compose version
App install size	9	19
Programming language	Kotlin	Kotlin
Number of dependencies	28	38
Number of Activities	1	1
Number of Fragments	8	1
Number of screens	6	6

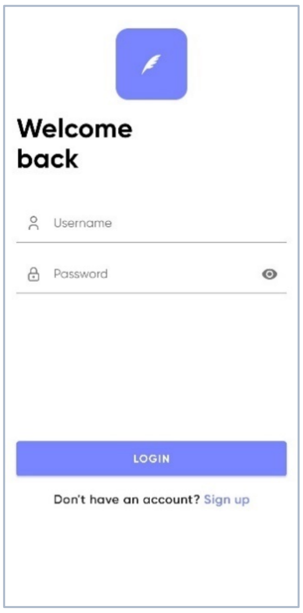


Figure 4.5: NotyKT (XML) login screen

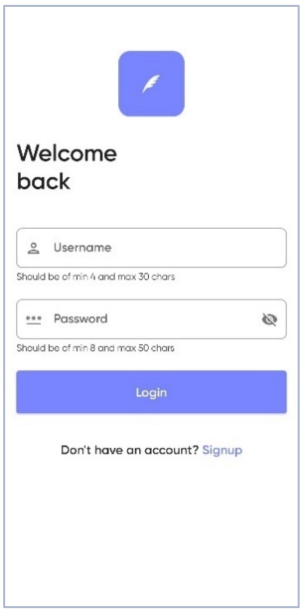

Welcome back
Username
Should be of min 4 and max 30 chars
Password
Should be of min 8 and max 50 chars
Login
Don't have an account? Signup

Figure 4.6: NotyKT (Compose) login screen

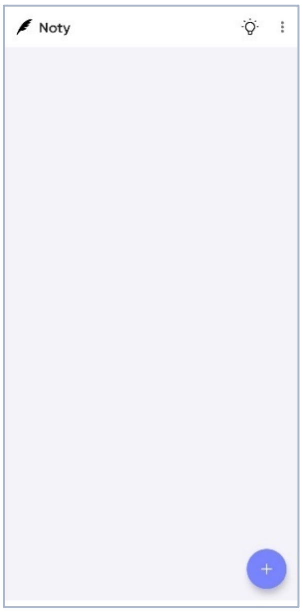


Figure 4.7: NotyKT (XML) main screen

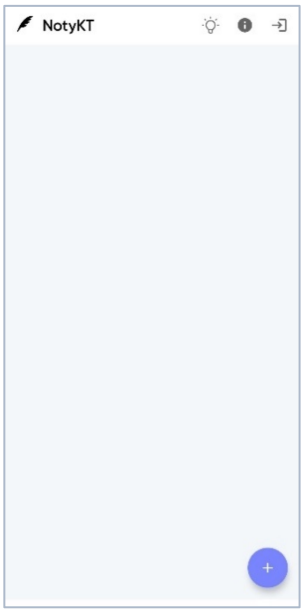


Figure 4.8: NotyKT (Compose) main screen

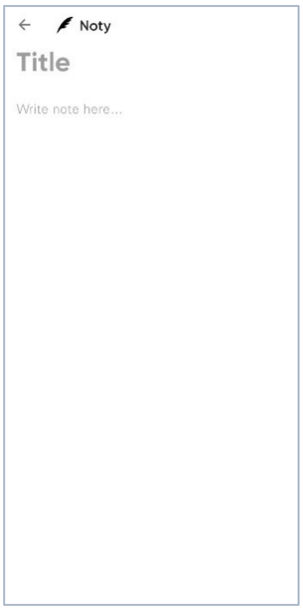


Figure 4.9: NotyKT (XML) add note screen

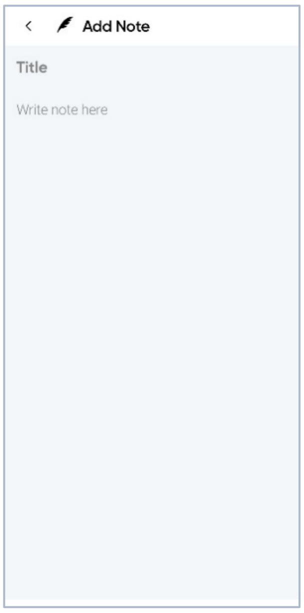


Figure 4.10: NotyKT (Compose) add note screen

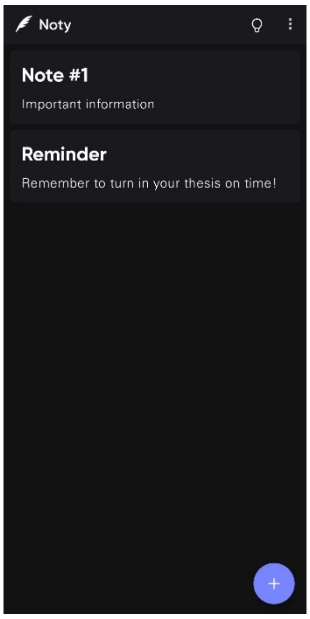


Figure 4.11: NotyKT (XML) added notes night mode

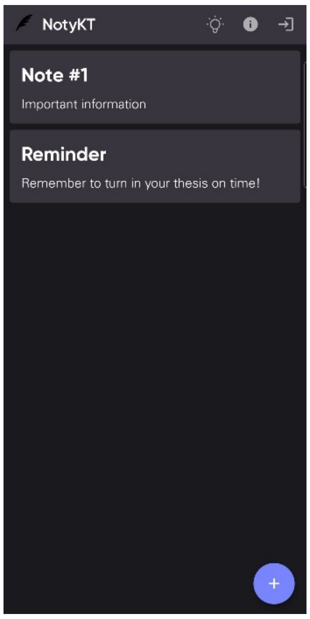


Figure 4.12: NotyKT (Compose) added notes night mode

5 Conclusions and results comparison

Analysis of this work is thoroughly done in this chapter. Some of the main hypotheses made in the initial part of the work are analyzed and looked at by numbers. Statistics in the previous chapter are being used as a base for the analysis. Some of the other non-numerical parameters are also going to be taken into the account. Not all of the applications can utilize all of the technologies and some of them have clear reasoning for choosing the right one.

The overall complexity and size of the application are big factors when determining what can be expected from the app. Another big one is functionality, as it would make sense to compare browsers to browsers, and media players to media players. Some of the applications are not being used at a mass scale. More focus will be put on the ones that are commercial and that have the most users, as those usually present a clearer picture when it comes to certain design choices and real-life flow when it comes to app development.

The ultimate goal of the analysis is to determine the current state of Android development, its progress through the years, which technologies are being used in which scenarios, and finally, how it is all meshing with Google's ideas and a big push toward Kotlin, Jetpack libraries, and Jetpack Compose UI.

5.1. Application complexity and size

All of the detailed data about the specific folder sizes, app installation sizes, and many other analyzed numbers can be found in the analysis tables located in the appendix part of the work. Apps regarded as large in the list take up more than 100MB after installation, medium apps take up between 50MB and 100MB, and small apps take up less than 50MB. Average app sizes on Google Play are somewhere in the medium region, around 60MB. [31]

Application size can be measured in multiple ways, depending on from which side is it being looked at. All but two applications use ProGuard or R8 for reducing the app size, which means that the statistics of app sizes should be in line and have the same starting point. From the developers' standpoint, this would be done by measuring the size of the source code folder, or the number of lines in the source code, as well as the number of included libraries and gradle files that are needed to compile the app. From the users' standpoint, it would mean the size of the installed app on the device.

Since developers usually do not have limited space on their devices and tend to use many more libraries in their app development and testing than in the actual final product, a metric that was taken as the most important one is the installed app size. Users can only have so many apps on their phones before it gets clogged with too many of them, so having the app size proportional to its value for the user is important. Not every app can be small, but the ones that take up much more space than some other apps that offer similar functionalities can easily be discarded by many users.

What is the average application install size?

When looking at the final numbers, several different conclusions can be drawn. In the analysis of twenty-seven apps, only four of them are bigger than 100MB when installed, with the average size after installation being 58MB.

The biggest of the four are two browser apps, Fenix (**3**) and Brave (**1**). This is to be expected since browsers contain a full stack of code and have extensive features that include a lot of different libraries. The other two big apps are Signal (**17**) the messaging app and Kickstarter (**6**) mobile version. Signal, similar to browser apps, contains several layers of full-stack architecture and implements many security features in the messaging system. Kickstarter as an app is very exhaustive has a large number of different screens and offers many unique features to the users.

Table 5.1: Application size table

	Small (< 50MB)	Medium (50-100 MB)	Large (>100MB)	Total
Application count	17	6	4	**27**
Average application installation size	24MB	75MB	179MB	**58MB**
Code size (in MB)	7MB	32MB	36MB	**17MB**
Kotlin (primary language) size	21MB	77MB	179MB	**59MB**
Java (primary language) size	26MB	81MB	179MB	**58MB**
Average number of dependencies	30	41	70	**38**
Average number of screens	10	27	21	**16**
Average number of activities	13	47	40	**25**
Average number of fragments	18	48	70	**32**

What affects application size the most?

One common denominator in the app size is the number of included libraries in the project. Every library has a certain size and adds a certain amount of additional code which attributes to the final app size. Since libraries generally don't have a certain size, the number of them doesn't necessarily determine how big the app will be, since most of them may be very small. Nevertheless, a large number of them guarantees some increase in size.

Every app in the set that has 50 or more external library dependencies is over 25MB big when installed, with most apps in the range of 50MB+ having almost exclusively as many. Code size does not necessarily relate to the final app size as there are a few apps that have several hundreds of MB in source folders yet take up less than 30MB on the phone. This part of the analysis as previously mentioned is also quite challenging due to many apps supporting multiple platforms and reusing some parts of the code for many of them, thus making it difficult to know which files are exactly included in which builds. Finally, even though bigger apps tend to have several tens of different screens that the user can access, there are also smaller apps with a large number of screens. The same set of conclusions applies to the number of activities and fragments since these are often closely correlated to the number of screens.

Conclusion

The programming language used, whether Java, Kotlin, C#, or C/C++ through JNI does not offer any real conclusions, as there are many applications in all of the languages that are both big and small. The bigger apps tend to use C/C++ more, but that is rather due to the complexity and additional functionalities of the app.

One interesting conclusion, although drawn from an exceedingly small dataset, is that Jetpack Compose apps tend to be bigger in size than the apps that use traditional Views for the UI. The biggest app in the set, Mozilla Fenix, uses Jetpack Compose and it takes up more than 250MB when installed, with Brave browser being around 5% smaller whilst offering similar functionalities and using Views.

An even more interesting comparison is between the two versions of the NotyKT app (**Table 4.1**). Two almost identical versions exist that share most parts of the code and offer the same functionalities, with one using Views and the other one Compose. The Compose version is more than twice as big, as it takes up 19MB compared to the 9MB of the Views version. These examples offer clear evidence that Compose applications are more demanding on the user's memory, but unfortunately, this statement would have to be supported with bigger numbers to be fully valid. This can also be attributed to the number of dependencies, as Compose applications require a number of additional libraries to be included and even a simple app like NotyKT has 10 more dependencies in Compose version.

The final conclusion from all of the data is that the application install size mainly increases with the high number of dependencies, with the number of Activities, Fragments, screens, programming language, and code size being much less of a factor.

5.2. Programming language

Using the data from the Stack Overflow survey, which is annually filled by about 70,000 developers, the shift towards Kotlin throughout the years is noticeable, but not dramatic. [32] [33] [24] Kotlin as a programming language only first appeared on the survey in 2018 when it had been used by 4.5% of developers after Google had already made a certain push to try and place it on the market. It was also the second most "loved" language, with a 75% approval rate. Java was at that time much higher on the usage list, being used by 45% of developers, but with a much lower approval rate of 51%. In only four years, during which Google made it the default Android language, Kotlin's usage doubled to 9%, while Java's dropped off to 33%. Java is still a much broader language, used for more than just Android apps, which is not the case for Kotlin, so the difference is noticeable. Despite this, the data confirm that Kotlin is becoming more popular and popular which is also emphasized by this research.

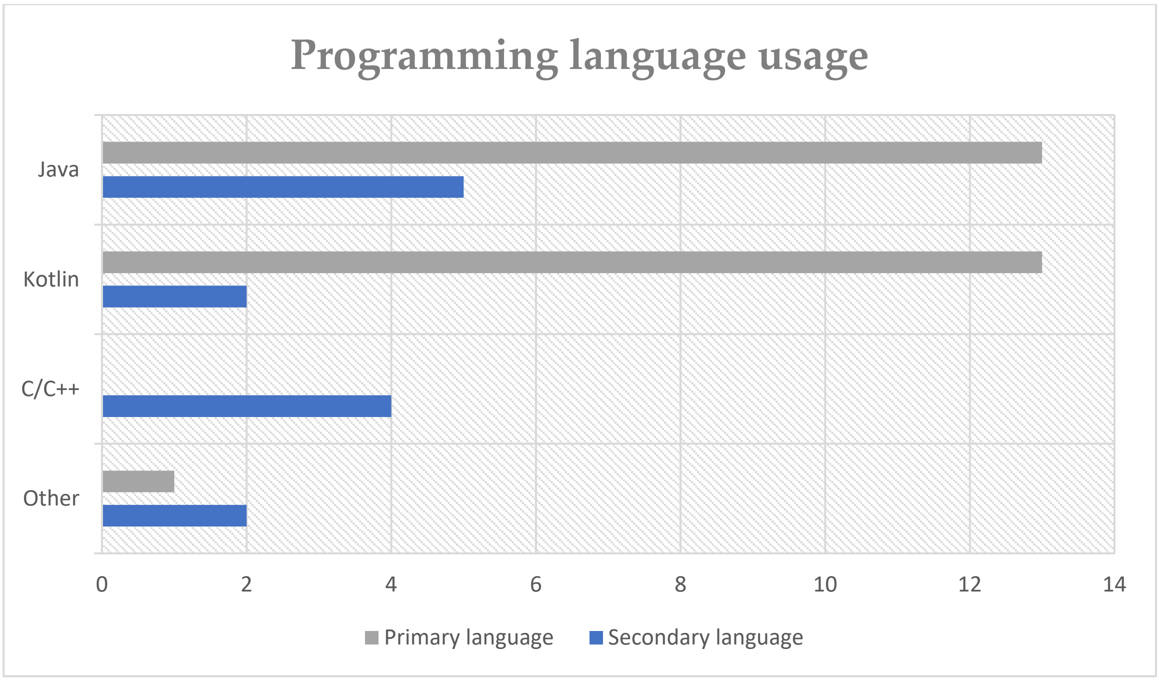


Figure 5.1: Programming languages usage chart

How many applications are using Java or Kotlin as a primary language?

Thirteen out of twenty-seven analyzed apps use Kotlin as their primary language, with another four currently in the transition phase where most of the code is still written in other languages, mainly Java. Only one application doesn’t use these languages with C# being represented once as a primary language (**5**).

How many applications use more than one language?

Thirteen out of twenty-seven applications are using more than one language. Most of them, five, are using Java together with Kotlin as a primary language, meaning that they are currently in the transition phase.

Multiplatform applications and the ones that require some low-level coding (**1**) use a lot of C/C++ code through JNI. This allows developers to have the same base of the app in one code, which is then dynamically translated to other platforms, thus keeping the consistency among functionalities. In total, four applications use C/C++ code (**1, 7, 17, 18**) and they all share similar characteristics when it comes to multiplatform support and low-level access. Only one application (**19**) uses Scala as its secondary language.

Table 5.2: Programming languages distribution table

	Java	Kotlin	C/C++	Other
Primary language	13	13	0	1 (C#)
Secondary language	5	2	4	2 (Scala, Python)
Average number of Activities based on a primary language	26	25	0	0
Average number of Fragments based on a primary language	29	38	0	0

Have any applications made the transition from Java to Kotlin?

Three of the apps (**3, 7, 16**) made a full transition from Java in the previous years and use minimal to no Java code, with another three (**6, 21, 23**) still using some Java code. Mozilla Fenix (Firefox) is the only large app that has made a full transition to Kotlin. This can be attributed to the large team that Mozilla has as well as the wide popularity among the developer community which helped out with the coding during the process.

Complex apps with a large number of files and complex structures are still not being refactored and translated to Kotlin. It can be concluded that the teams are unwilling to put in the enormous effort required to do this with no direct benefit to the users. Applications that use C/C++ code have not made the transition most likely due to the complexity of the app and a lot more refactoring that would be needed. Those apps also contain a lot of code and are categorized as medium/large.

What is the number of Activities and Fragments in Kotlin-based and Java-based apps?

The number of Activities and Fragments doesn’t bring too many conclusions. The average number of Activities in Kotlin applications is almost identical to the ones written in Java, with Kotlin having around 30% higher number of Fragments, which is still not big enough margin to draw any major conclusions and can be attributed to the characteristic of the dataset.

Conclusion

Java is still a number one programming language for Android, but Kotlin is rapidly taking over. All of the apps newer than 2017 are written in Kotlin and many older ones are being translated to Kotlin from Java.

5.3. User interface (UI)

Considering that Jetpack Compose is a relatively new technology that completely changes the way the UI works it is to be expected that it hasn't completely caught on yet. Unlike making the transition from Java to Kotlin, the transition from Views to Compose is much harder to be done gradually. Views and Compose do not mesh very well together, even though it is possible, but the whole idea behind Compose and the way it works requires completely different architecture. Switching from Java to Kotlin can be done gradually, as both of the languages are the same in the core, and interoperability is very well supported. It is easy to conclude that the switch to Compose is a much greater step for development teams and it is not clear why would the teams do it.

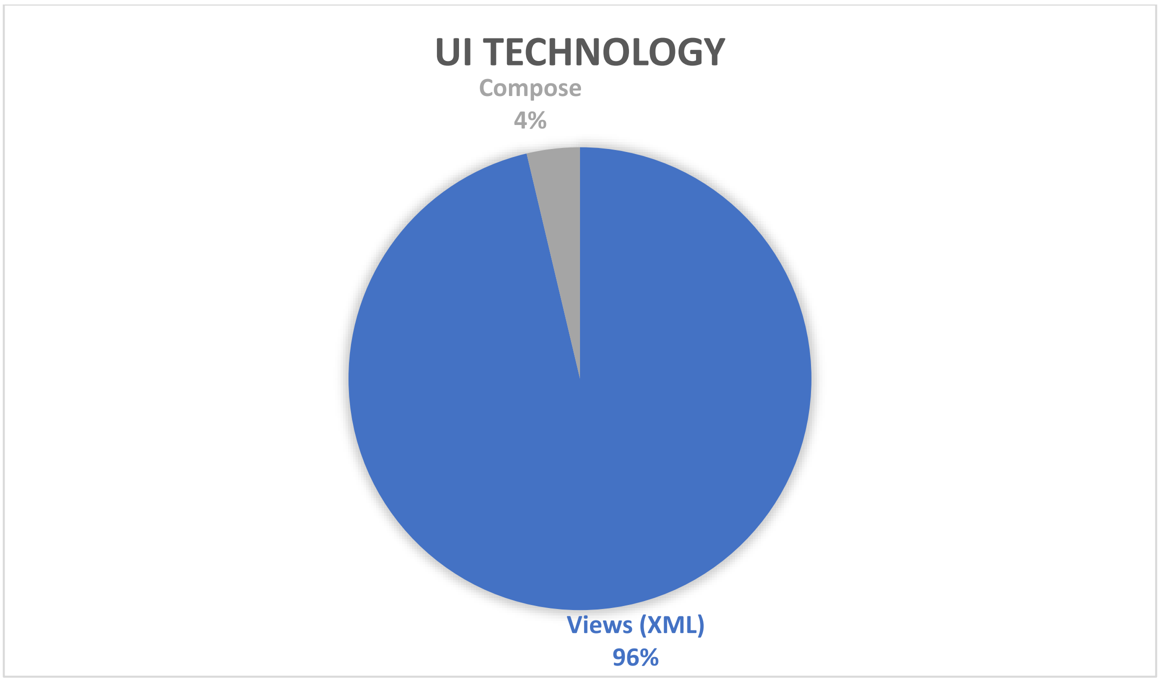


Figure 5.2: UI technology distribution chart

Which technology is being used for the UI - Compose, or Views (XML)?

Out of all the analyzed apps, only two of them are using Jetpack Compose. One of those is an additional app having a Compose version next to the Views one NotyKT (**26**), and the other one already mentioned, refactored Mozilla Fenix (Firefox) (**3**). Despite Compose being on the market for a few years now and having a stable version for more than a year, no applications seem to catch on.

It is even less likely that the other non-open-source apps have transitioned to it as it would take a lot of working hours for the whole operation, without any direct benefits for the user.

What is the number of Activities, Fragments, and screens in the respective technologies?

The number of Compose applications doesn't allow for a deeper answer to this question. In theory, Compose applications should be made with only one Activity (or at least as less as possible) which is reflected even on the only Compose example, Mozilla Fenix, where the most complex and the biggest application of all has the least number of Activities. Some of the bigger XML applications from the dataset have a much bigger (**9** and **17)** have a much larger number of Fragments than Mozilla Fenix meaning that the number of Fragments could also be much lower in Compose apps.

Table 5.3: UI technology distribution table

	Views (XML)	**Compose**
Number of apps using a UI technology	26	1
Average number of Activities based on a UI technology	25	12
Average number of Fragments based on a UI technology	30	91
Average number of screens based on a UI technology	16	18

Conclusion

Google is most likely aware that not a lot of commercial applications are using Compose - most likely because the developers do not want to do the entire refactoring process like Mozilla did, with no real benefits for the users. Some of the basic examples even showed that Compose performance is worse than View performance on equivalent screens [34] [35]. The provided research is limited to a small number of simple examples, but it does offer a good base for further research. What are the real advantages of using Compose over Views (XML), other than cleaner and reusable code for developers?

Working with designers is much easier with Views, since components made in Figma, the most used app design tool, can easily be exported in all types of formats and are nearly seamlessly integrated into Android apps. Google also recently announced better compose integration with Figma, a plugin called Relay [36] that will bring the compose way of coding UI closer to designers. Whether this will be a step in the right direction that will attract more developers and designers to the platform, is to be determined.

The industry still clearly states that the Views using XML are the preferred technology for the UI development.

5.4. Google libraries and services

This chapter is trying to answer a simple question - how much do developers trust Google and their services? The answer is obtained by answering several different questions all regarding smaller modules that are made by Google. Coincidentally, another question will be answer in this chapter – which other non-native libraries and services are being used?

Do these apps use the Room database?

Ever since the introduction of the Android Jetpack there has been a lot of talk among developers on which libraries are useful. One that was needed was a local database, which is provided by Room. It offers a great deal of flexibility, and integration with any online database, and is relatively easy to implement. Room first appeared in the first half of 2018 and can now be found in many apps. In the research data set, it is located in fourteen out of twenty-seven analyzed apps. Even though the number may not seem remarkably high, some of the tested apps do not require a local database. Room is a tool that is used among developers.

Which Dependency Injection libraries are being used?

Fourteen out of twenty-seven applications use dependency injection libraries. Eleven out of those fourteen use Dagger or Dagger - Hilt. The other three use Koin, which offers some other advantages but is only supported on Kotlin. This is to be expected since both Dagger and Dagger - Hilt are official Google DI libraries that work with both Java and Kotlin, and even when refactoring the app and changing the language from Java to Kotlin it serves no real purpose to also changing a DI library.

Table 5.4: Libraries and DI usage table

Library	Number of apps using it
Room	13
ProGuard/R8	25
Dagger	5
Dagger - Hilt	6
Koin	3

Google Firebase - which services are being used and which are not?

Firebase is often labeled as a set of services that are mostly used by inexperienced developers to ease their way into development, or by smaller teams who do not have the workforce to manually do every layer of the system. It is quite rare to see the most popular applications that are using Firebase for database or authentication services. None of the analyzed apps use them, most likely due to their simplicity. On the other hand, some other services provide a set of functionalities that are more than adequate for an average app.

Fifteen out of twenty-seven apps use some of the Firebase services. The most common ones are Analytics, Crashlytics, and Cloud Messaging. These services provide enough data for an average developer to know what has gone wrong with the app at certain moments and to deliver quick messages to users through notifications. Another service that is found in Remote Config, allows developers to make minor changes within the app without changing any code or publishing a new version.

Table 5.5: Firebase services usage table

Firebase service	Number of apps using it
Cloud Messaging	8
Analytics	6
Crashlytics	6
Remote Config	4
Any	15

Conclusion

Developers trust Google and are quite keen on using their services if they make the experience easier and better. Despite some of Google's services not offering the complexity and features of some other tools, the ease of implementation, good support, and overall stability are more than enough to obtain many developers. With Google investing a lot of time and money into not only improving their services but also expanding the domain, the trend seems to be going in the direction where more and more layers of apps are completely handled by native Google services. This could lead to a much more streamlined developer experience and many more stable apps.

5.5. Architectural and design patterns

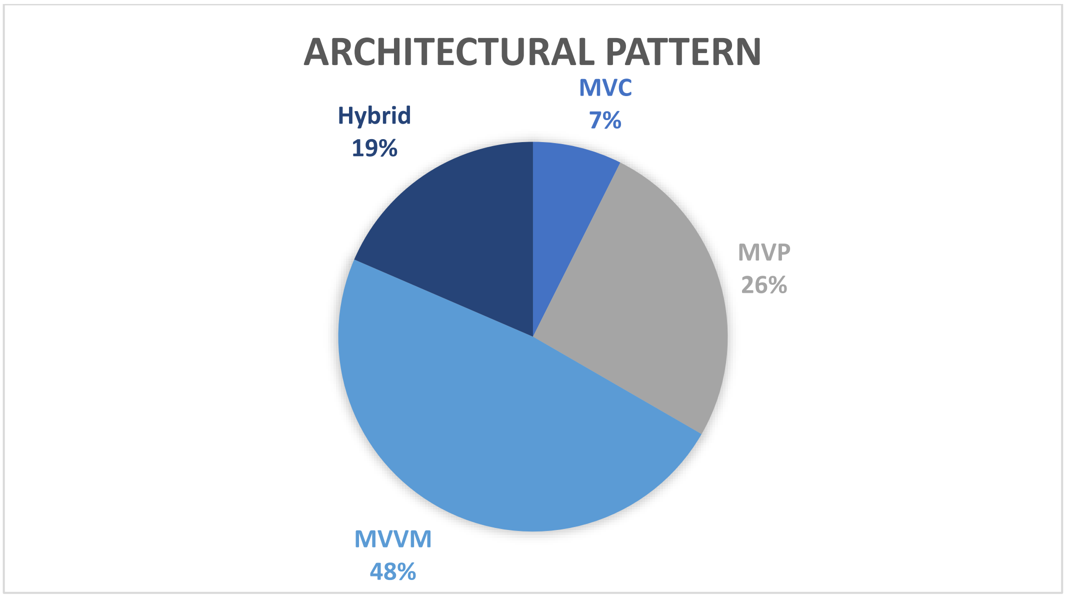


Figure 5.3: Architectural patterns distribution chart

What is the most common architectural pattern?

The most popular architecture among the apps is the MVVM. This is in no way connected with Compose, which is almost exclusively used with MVVM, as almost no apps use it. MVVM has gradually taken over the market and is most often found as a recommended architecture in guides and tutorials. There are no clean architecture usages, which could be attributed to the fact that it rarely works well with medium and large apps due to bad scalability.

Transition from MVC through MVP to MVVM is also visible from the numbers as only two applications are using MVC, with seven using MVP, and thirteen using MVVM. Seven applications are still in the transition phase or are just using more than one pattern to better suit the needs.

Table 5.6: Architecture patterns distribution table

Architecture patterns	Number of apps using it
MVC	2
MVP	7
MVVM	13
Hyrbid	5

Which are the most common design patterns?

Using design patterns has become a standard a long time ago. Every tested app contains more than one. Only the most used design patterns, which have been previously mentioned in this work in the Chapter 3, have been analyzed. Behavioral patterns like observers (which are represented by listeners), states, and iterators are so common that there are almost no apps that use them. A similar thing can be said for singletons, which are generally used when creating a local database, and for adapters, which are used for adapting a certain type of data to the app.

The most common out of the rest are builders, with nineteen appearances, followed by factories with sixteen, and the previously mentioned dependency injection with fourteen. The least used are facades, represented by Managers, and decorators.

It is clear that design patterns are a standard and that their role of helping the developers effectively resolve common problems is well fulfilled.

Table 5.7: Design patterns usage table

Design patterns	Number of apps using it
Builder	19
DI	14
Singleton	26
Factory	16
Adapter	26
Decorator	7
Facade	13
Observer	26
State	26
Iterator	26

Conclusion

Most of the applications have embraced the latest ways of MVVM architectural pattern and use a large number of design patterns. These, already established and well documented patterns, are making both the design and execution of the app much simpler while saving a lot of time for the developers.

5.6. Conclusion

"The main hypothesis going into this work is that the latest technologies offer the most flexibility and the best performance, with the newest applications that integrate the most recent technologies being the easiest to use, analyze, and develop. Adding to that is that most of the developers have or will have switched to the newer technologies knowing that they will be supported and updated for many years, having the full backing of Google and its developers, making development faster and easier."

Almost none of the applications use the newest technology by Google for the UI - Jetpack Compose, which is a relatively new technology but is going to have a huge support from Google for many years and is only going to be improved in the future. This did not urge any developers, other than the team from Mozilla that developed Fenix, to completely switch to it. This point about Compose also could not have been thoroughly researched, since the first stable version of Compose came out only in 2021. That is a noticeably brief period for an application to be developed and to gain a certain number of stars on GitHub or downloads for Google Play which was the criteria for making this app selection. All of the applications in this set have had their initial release a year or more before. Most of the bigger teams that could do that amount of work in a brief period, like previously mentioned Mozilla, tend not to have their repositories open source. There is a possibility that newer Compose applications will improve and grow to be more downloaded in the upcoming years. The belief is that most of the applications that have been in development since 2021 would have used Compose as their primary UI tool.

Kotlin as a programming language, with big backing from Google after making it the default Android language in 2019, is more common among the newer apps. Almost all of the apps that had initial releases in 2017 or later use Kotlin as their primary language, which goes hand-in-hand with the hypothesis. Kotlin is much easier to learn and use than Java and it is probably likely that new developers, and new apps, will be even more oriented toward it.

Several applications have made partial or full transitions from Java, with some of them still in progress. Some of the other ones have just recently begun their process and are currently in the transitioning phase. The processes were gradual, approximately starting around 2018 for all of them, with either only new files being done in Kotlin, or also the old files being replaced. The industry seemed to have completely jumped onto the Kotlin wave, although still, a fair number of older applications are to stay in Java. This especially seems to be the case with the applications that also use some C/C++

code through JNI, as their architectures tend to be more complex and are much more difficult to refactor.

When it comes to Android Jetpack and its libraries and tools beside Compose, it appears that it is very well accepted among the developers. Jetpack originally was not meant to revolutionize development but rather to standardize what is already being used in the industry and make it even more streamlined for developers. It was almost impossible to find usages of some of the older libraries which have the Jetpack equivalent library.

The most "modern" architecture, MVVM, is by far the most used. Almost all of the newer applications use it, with MVP still being somewhat utilized in some of the older ones. There also are a few instances of some hybrid architectures, which are the result of developers trying to implement new methods without rewriting the entire code from scratch. The rise of the popularity of MVVM cannot be contributed to the rise of the popularity of Compose, as the apps that use Compose tend to also use MVVM due to the state-focused architecture. MVVM is currently the most used architectural pattern, and it appears that is not going to change in the near future.

There are still a couple of questions that need getting answered. First of the two is the following - **How many years does it take for a certain technology to become widely used in the industry since becoming stable?**

The industry seems to be following the trends and quickly catching on to the newer technologies that offer numerous advantages when it comes to language, libraries, and architecture. The same cannot be said about the Jetpack Compose UI, which despite Google's enormous efforts to bring it to the market, still cannot be found very often in any of the applications. Since the industry seems to catch on more quickly to the things that are proven to work, another question must be asked - is Compose truly the right choice for the UI, or are the developers refusing to change because of the lack of quality of this technology? Regardless of the answer, one year of stable release is too short of a period for a technology to stick. Kotlin gives a bit better approximation, with five years of being stable and three years of being recommended by Google, it was found in 50% of the applications with all of the recent applications seemingly catching on. The same can be said for Android Jetpack library which is available for a similar period of time and is being utilized very often. Anywhere between three and five years seems to be a good approximation based on the numbers of this research.

The second and final question - **What is the future Android development?**

When it comes to the programming language, Kotlin will most likely completely take over and only older applications that are either made by smaller teams or do not have any major updates will stay in Java. All of the new applications as well as a good number of older ones will be written and translated to Kotlin. A push for Kotlin multiplatform support will also be another strong argument for sticking to Kotlin.

As most of the applications are using many different Google services and libraries it seems that Google has developed a range of products that are accepted and are getting more recognition in the developer community. A growing rate of usage of these libraries in the last few years suggest that we could, eventually, have a completely Google-controlled Android development experience, with all of the libraries and services being directly developed by them while offering easy integration and full support. More complex tools are still going to be used, but most likely only by the more complex applications, as Google's focus seems to be on low-to-mid complexity.

Finally, if Google continues to develop Jetpack Compose in the same way it has its other technologies, its usage could grow drastically. As Compose is the major talk of every new Google technology update and conference, it seems that a lot of time and effort is going into it. Only time will tell, but there should not be any surprises if in 5 years most of the apps are using Jetpack Compose and only older ones stick to Views with XML.

Bibliography

[1] "https://gs.statcounter.com/os-market-share/mobile/worldwide," 2022. [Online].

[2] "https://gs.statcounter.com/os-version-market-share/android," 2022. [Online].

[3] "https://www.baeldung.com/kotlin/kotlin-java-performance," 2022. [Online].

[4] "https://kotlinlang.org/docs/comparison-to-java.html#what-java-has-that-kotlin-does-not," 2022. [Online].

[5] "https://apiumhub.com/tech-blog-barcelona/java-vs-kotlin/," 2018. [Online].

[6] "https://developer.android.com/kotlin," 2022. [Online].

[7] "https://www.xenonstack.com/blog/kotlin-andriod#:~:text=Any%20chunk%20of%20code%20written,run%20in%20a%20Kotlin%20project," 2022. [Online].

[8] Mozilla, "https://developer.mozilla.org/en-US/docs/Glossary/MVC," 2022. [Online].

[9] Boodhoo, "Design Patterns: Model View Presenter," August 2006. [Online]. Available: https://learn.microsoft.com/en-us/archive/msdn-magazine/2006/august/design-patterns-model-view-presenter.

[10] Kouraklis, MVVM in Delphi, Apress, Berkeley, CA, 2016.

[11] "https://developer.android.com/jetpack/androidx," 2020. [Online].

[12] "https://developer.android.com/stories/apps/monzo-camerax," 2022. [Online].

[13] "https://developer.android.com/jetpack/androidx/explorer," 2022. [Online].

[14] "https://developer.android.com/jetpack/compose/mental-model," 2022. [Online].

[15] "https://developer.android.com/jetpack/compose," 2022. [Online].

[16] H. J. V. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[17] "https://www.kodeco.com/18409174-common-design-patterns-and-app-architectures-for-android#toc-anchor-002," 2021. [Online].

[18] "https://dagger.dev/," 2022. [Online].

[19] "https://insert-koin.io/," 2022. [Online].

[20] "https://refactoring.guru/design-patterns/adapter," 2022. [Online].

[21] "https://data-flair.training/blogs/android-application-components/," 2020. [Online].

[22] "https://developer.android.com/guide/components/fundamentals," 2022. [Online].

[23] "https://firebase.google.com/products-build," 2022. [Online].

[24] "https://insights.stackoverflow.com/survey/2018," 2018. [Online].

[25] "https://firebase.google.com/docs/firestore/rtdb-vs-firestore," 2022. [Online].

[26] "https://db-engines.com/en/ranking," 2022. [Online].

[27] "https://insights.stackoverflow.com/survey/2017," 2017. [Online].

[28] "https://insights.stackoverflow.com/survey/2021," 2021. [Online].

[29] "https://developer.android.com/studio/build/shrink-code," 2022. [Online].

[30] "https://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications," 2020. [Online].

[31] "https://www.statista.com/statistics/1296527/size-top-android-apps," 2022. [Online].

[32] "https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/," 2022. [Online].

[33] "https://survey.stackoverflow.co/2022/#most-popular-technologies-language," 2022. [Online].

[34] Shelor, "https://engineering.premise.com/measuring-render-performance-with-jetpack-compose-c0bf5814933," 2021. [Online].

[35] Arora, "https://medium.com/okcredit/comparing-jetpack-compose-performance-with-xml-9462a1282c6b," 2022. [Online].

[36] "https://relay.material.io/," 2022. [Online].

[37] "https://support.microsoft.com/en-us/word," [Online].

[41] "https://developer.android.com/studio/releases/past-releases," 2022. [Online].

[42] "https://emulation.gametechwiki.com/index.php/Android_emulators," 2022. [Online].

[43] "https://www.ibm.com/cloud/learn/java-explained," 2019. [Online].

[44] S. E. I. Carneige Mellon University, "What is your definition of software architecture?," 2010.

[45] C. K. Bass, Software Architecture in Practice (2nd edition), Addison-Wesley, 2003.

[46] "ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems," 2000.

[47] Krutchen, "Rational Unified Provess," 1999.

A Appendix A

In this appendix a full analysis table is located. All of the individual applications are provided with in-detail statistics of the analysis in the table in the chapter A.1. Chapter A.2 consists of a table with the list of all the analyzed GitHub repositories.

A.1. Complete table of applications analysis results

Table A.1: Complete table of applications' analysis results

	1 Brave	2 DuckDuckGo	3 Fenix	4 Orbot	5 Bitwarden	6 Kickstarter
Repository opened/ first release	January 2012	December 2017	June 2019	March 2017	August 2016	February 2017
Date of analysis	16th October	16th October	10th October	16th October	16th October	16th October
Analyzed version	v1.46.59 - Oct 2022	v5.138.1 - Oct 2022	v105.1.0 - Sep 2022	v16.6.2 - Jul 2022	v2022.10 - Oct 2022	v3.5.0 - Sep 2022
Primary language	Java	Kotlin	Kotlin (Full transition from Java)	Java	C#	Kotlin (Nearly full transition from Java)
Secondary language	JNI – C/C++					Java
Code size – *src* folder (in MB)	38.2	13.6	29.4	1.71	1.38	14.5
Code + libraries + resources (in MB)	420	48	1620	23	44	26.7
Installed app size (MB)	239	78	257	48	55	101
Number of depdendencies	50+	50+	118	25	50+	50
UI technology	XML	XML	Compose	XML	XML	XML
Number of screens	20	22	18	5	6	24
Number of Activities (Fragments)	25 (57)	40 (42)	12 (91)	9 (9)	0 (0)	49 (13)
Architecture	MVVM	Hybrid (MVVM + MVP)	MVC + MVVM	MVP	MVVM	MVVM
Design patterns	Builders, Factories, Adapters	DI (Dagger), Builders, Factories, Adapters, Decorators, Facades	Builders, Decorators	Adapters	N/A	DI (Dagger), Builders, Factories, Adapters
Application components	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	N/A	Activities + Fragments, Services, Receivers, Providers
Firebase | Room | Proguard (R8)	No | No | Yes	No |Yes | Yes	Yes (Analytics + Crashlytics + Cloud Messaging) | Yes | Yes	No | No | Yes	Yes (Cloud Messaging) | No | No	Yes (Analytics + Cloud Messaging) | No | No

	7 Shadowsocks	8 Wikipedia	9 Wordpress	10 Antenna	11 NewPipe	12 Phonograph
Repository opened/ first release	June 2014	January 2012	December 2015	February 2014	September 2015	April 2017
Date of analysis	10th October	16th October	15th October	16th October	15th October	15th October
Analyzed version	v5.2.6 - Sep 2021	v2.7 - Oct 2022	v20.9 - Oct 20202	v2.7.1 - Oct 2022	v0.24.0 - Oct 2022	v1.3.5 - Sep 2020
Primary language	Kotlin (Full transition from Java)	Kotlin	Kotlin	Java	Java	Java
Secondary language	JNI – C/C++		Java			
Code size – *src* folder (in MB)	1.5	17.1	34.4	33	8.43	4.27
Code + libraries + resources (in MB)	40	17.4	114	160	9.28	60.5
Installed app size (MB)	27	38	97	42	15	7
Number of depdendencies	27	50+	25	46	47	33
UI technology	XML	XML	XML	XML	XML	XML
Number of screens	6	9	50+	20	10	8
Number of Activities (Fragments)	11 (16)	47 (42)	99 (167)	11 (45)	12 (31)	14 (20)
Architecture	Hybrid	MVVM	MVVM	Hybrid	MVVP	MVP
Design patterns	Adapters	Builders, Factories, Adapters, Facades	Builders, DI (Dagger – Hilt), Factories, Adapters, Decorators, Facades	Buildrs, Factories, Adapters, Decorators, Facades	Builders, Facades, Adapters	Builders, Adapters, Facades
Application components	Activities + Fragments, Services, Receivers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services	Activities + Fragments, Services, Receivers, Providers
Firebase | Room | Proguard (R8)	Yes (Analytics + Crashlytics) | Yes | Yes	Yes (Cloud Messaging) | Yes | Yes	Yes (Remote Config + Cloud Messaging) | Yes | Yes	No | No | Yes	No | Yes | Yes	No | No | Yes

	13 Shuttle	14 Timber	15 K9	16 QKSMS	17 Signal	18 Telegram
Repository opened/ first release	March 2017	January 2016	January 2014	October 2015	October 2013	January 2014
Date of analysis	15th October	15th October	16th October	16th October	15th October	16th October
Analyzed version	v2.0.17 - Jul 2020	v1.7 - Oct 2020	v6.202 - Jul 2022	v3.9.4 - Feb 2021	v5.53.2 - Oct 2022	v8.8.2 - Jun 2022
Primary language	Java	Java	Java	Kotlin (Full transition from Java	Java	Java
Secondary language	Kotlin		Kotlin	Java	JNI – C/C++	JNI – C/C++
Code size – *src* folder (in MB)	5	4.45	4	5.8	62	83.7
Code + libraries + resources (in MB)	170	73	176	6	63	357
Installed app size (MB)	26	16	38	21	119	75
Number of depdendencies	50	22	19	18	60	26
UI technology	XML	XML	XML	XML	XML	XML
Number of screens	19	11	34	10	23	19
Number of Activities (Fragments)	9 (16)	9 (15)	30 (42)	12 (0)	73 (118)	103 (4)
Architecture	MVP	MVP	MVVM	Hybrid	MVP	MVC
Design patterns	DI (Dagger), Builders, Adapters, Facades	Builders, Factories, Adapters	Builders, DI (Koin), Factories, Adapters, Facades	DI (Dagger), Builders, Factories, Adapters, Facades	DI (Dagger), Builders, Factories, Adapters, Decorators	Builders, Factories, Adapters, Facades
Application components	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers
Firebase | Room | Proguard (R8)	Yes (Remote Config) | No | Yes	Yes (Crashylitics) | No | Yes	No | No | Yes	Yes (Crashlytics) | No | Yes	Yes (Analytics + Cloud Messaging) | No | Yes	Yes (Remote Config + Cloud Messaging) | No | Yes

	19 Wire	**20 Google I/O**	**21 Habitica**	**22 Materialistic**	**23 Muzei**	**24 OmniNotes**
Repository opened/ first release	August 2016	February 2016	November 2015	April 2016	February 2014	August 2015
Date of analysis	16th October	10th November	16th October	15th October	12th October	15th October
Analyzed version	v3.82.38 - Aug 2022	v2021	v4.02 - Sep 2022	v3.3 - Mar 2019	v3.4.1 - Jan 2022	v6.1.0 - Mar 2022
Primary language	Java, Scala	Kotlin	Kotlin (Nearly full transition from Java)	Java	Kotlin (Nearly full transition from Java)	Java
Secondary language	Kotlin		Java		Python, Java	
Code size – *src* folder (in MB)	45.2	5.5	12	5.35	2.76	5.12
Code + libraries + resources (in MB)	50	8	19.1	51	230	91
Installed app size (MB)	87	13	55	14	25	28
Number of depdendencies	50+	7	46	20	17	40
UI technology	XML	XML	XML	XML	XML	XML
Number of screens	12	6	50+	16	7	5
Number of Activities (Fragments)	11 (5)	5 (34)	29 (67)	23 (6)	11 (16)	13 (8)
Architecture	MVC	MVVM	MVVM	MVP	MVVM	MVP
Design patterns	DI (Koin), Factories, Adapters, Decorators, Facades	DI (Dagger – Hilt), Adapters, Observers	DI (Dagger – Hilt), Builders, Factories, Adapters, Facades	DI (Dagger), Builders, Adapters, Facades	Builders, Factories, Adapters, Facades	Factories, Adapters
Application components	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers	Activities + Fragments, Services, Receivers, Providers
Firebase | Room | Proguard (R8)	Yes (Cloud Messaging) | Yes | Yes	Yes | Yes | Yes	Yes (Analytics + Remote Config + Crashlytics) | No | Yes	No | Yes | Yes	Yes (Analytics + Crashlytics) | Yes | Yes	No | No | Yes

	25 Kotlin Pokedex	26 NotyKT	27 Pokedex
Repository opened/ first release	February 2020	October 2020	December 2019
Date of analysis	20th October	10th October	20th September
Analyzed version	v - May 2020	v2.1.0 - Oct 2022	1.1.0 Aug 2022
Primary language	Kotlin	Kotlin	Kotlin
Secondary language		Kotlin	Java
Code size – *src* folder (in MB)	4.04	1	1.63
Code + libraries + resources (in MB)	68	4.03	14.6
Installed app size (MB)	24	9 (19 Compose version)	13
Number of depdendencies	26	28 (38 Compose version)	27
UI technology	XML	XML (Compose)	XML
Number of screens	2	6	3
Number of Activities (Fragments)	1 (0)	1 (8; 0 Compose version)	4 (0)
Architecture	MVVM	MVVM	MVVM
Design patterns	Adapters, DI (Koin)	Factories, Adapters, DI (Dagger – Hilt)	DI (Dagger – Hilt), Factories, Adapters
Application components	Activities + Fragments, Services, Receivers, Providers	Activities (Fragments)	Activities, Providers
Firebase | Room | Proguard (R8)	No | Yes | Yes	No | Yes | Yes	No | Yes | Yes

A.2. List of analyzed applications' repositories

Table A.2: List of analyzed applications' repositories

App number	App name	Repository link
1	**Brave**	https://github.com/brave/brave-browser
2	**DuckDuckGo**	https://github.com/duckduckgo/Android
3	**Fenix**	https://github.com/mozilla-mobile/fenix
4	**Orbot**	https://github.com/guardianproject/orbot
5	**Bitwarden**	https://github.com/bitwarden/mobile
6	**Kickstarter**	https://github.com/kickstarter/android-oss
7	**Shadowsocks**	https://github.com/shadowsocks/shadowsocks-android
8	**Wikipedia**	https://github.com/wikimedia/apps-android-wikipedia
9	**Wordpress**	https://github.com/wordpress-mobile/WordPress-Android
10	**Antenna**	https://github.com/AntennaPod/AntennaPod
11	**NewPipe**	https://github.com/TeamNewPipe/NewPipe
12	**Phonograph**	https://github.com/kabouzeid/Phonograph
13	**Shuttle**	https://github.com/timusus/Shuttle
14	**Timber**	https://github.com/naman14/Timber
15	**K9**	https://github.com/thundernest/k-9
16	**QKSMS**	https://github.com/moezbhatti/qksms
17	**Signal**	https://github.com/signalapp/Signal-Android
18	**Telegram**	https://github.com/DrKLO/Telegram
19	**Wire**	https://github.com/wireapp/wire-android
20	**Google I/O**	https://github.com/google/iosched
21	**Habitica**	https://github.com/HabitRPG/habitica-android
22	**Materialistic**	https://github.com/hidroh/materialistic
23	**Muzei**	https://github.com/muzei/muzei
24	**Omni Notes**	https://github.com/federicoiosue/Omni-Notes
25	**Kotlin Pokedex**	https://github.com/mrcsxsiq/Kotlin-Pokedex
26	**NotyKT**	https://github.com/PatilShreyas/NotyKT
27	**Pokedex**	https://github.com/skydoves/Pokedex

List of Figures

Figure 1.1: Android development timeline 14
Figure 2.1: MVC scheme 20
Figure 2.2: MVP scheme 21
Figure 2.3: MVVM scheme 22
Figure 2.4: Android Jetpack components 24
Figure 2.5: Button code example in XML 26
Figure 2.6: Button code example in Compose 26
Figure 2.7: View hierarchy 27
Figure 2.8: Crashlytics dashboard 35
Figure 2.9: Firebase Cloud Firestore example 36
Figure 4.1: Mozilla Fenix (Firefox) on Android 73
Figure 4.2: Mozilla Fenix (Firefox) settings on Android 73
Figure 4.7: Mix of Kotlin and Java files in WordPress source code 78
Figure 4.8: Mix of Kotlin and Java files in Kickstarter source code 78
Figure 4.9: NotyKT (XML) login screen 81
Figure 4.10: NotyKT (Compose) login screen 81
Figure 4.11: NotyKT (XML) main screen 81
Figure 4.12: NotyKT (Compose) main screen 81
Figure 4.13: NotyKT (XML) add note screen 81
Figure 4.14: NotyKT (Compose) add note screen 81
Figure 4.15: NotyKT (XML) added notes night mode 81
Figure 4.16: NotyKT (Compose) added notes night mode 81
Figure 5.1: Programming languages usage chart 87
Figure 5.2: UI technology distribution chart 90
Figure 5.3: Architectural patterns distribution chart 95

List of Tables

Table 1.1: Usage of Android OS versions 13
Table 2.1: Java and Kotlin differences 17
Table 3.1: Analyzed apps listed by category 40
Table 3.2: Brave analysis table 43
Table 3.3: DuckDuckGo analysis table 44
Table 3.4: Mozilla Fenix analysis table 45
Table 3.5: Orbot analysis table 46
Table 3.6: Bitwarden analysis table 47
Table 3.7: Kickstarter analysis table 48
Table 3.8: Shadowsocks analysis table 49
Table 3.9: Wikipedia analysis table 50
Table 3.10: WordPress analysis table 51
Table 3.11: Antenna analysis table 52
Table 3.12: NewPipe analysis table 53
Table 3.13: Phonograph analysis table 54
Table 3.14: Shuttle analysis table 55
Table 3.15: Timber analysis table 56
Table 3.16: K9 analysis table 57
Table 3.17: QKSMS analysis table 58
Table 3.18: Signal analysis table 59
Table 3.19: Telegram analysis table 60
Table 3.20: Wire analysis table 61
Table 3.21: Google I/O analysis table 62
Table 3.22: Habitica analysis table 63
Table 3.23: Materialistic analysis table 64
Table 3.24: Muzei analysis table 65

Table 3.25: OmniNotes analysis table........ 66
Table 3.26: Kotlin Pokedex analysis table........ 67
Table 3.27: NotyKT analysis table........ 68
Table 3.28: Pokedex analysis table........ 69
Table 4.1: NotyKT application XML and Compose version comparison table........ 80
Table 5.1: Application size table........ 85
Table 5.2: Programming languages distribution table........ 88
Table 5.3: UI technology distribution table........ 91
Table 5.4: Libraries and DI usage table........ 93
Table 5.5: Firebase services usage table........ 94
Table 5.6: Architecture patterns distribution table........ 95
Table 5.7: Design patterns usage table........ 96
Table A.1: Complete table of applications' analysis results........ 106
Table A.2: List of analyzed applications' repositories........ 111

Acknowledgments

I dedicate this work and my whole tenure in Milan to the one person without whom I would have probably never had come here, Ricardo. And of course, to the people without whom my time in Milan would have been lame – Andrea, Angelo, Emma, Francesca, Franco, George, Gosia, Heitor, Ivana, Kristina, Laura, Margot, Martìn, Pedro, Philipp, Rebeka, Sanja, Theresa, Tom, and most importantly, Toma, just to name a few.