Maintainability Evolution of Open-source Android applications

VU UNIVERSITY AMSTERDAM

Faculty of Sciences

Bojan Filipović

Computer Science

VU University Amsterdam

A thesis submitted for the degree of

Computer Science
Track Software Engineering and Green IT

2017

1. First Reader: Ivano Malavolta
2. Daily Supervisor: Ivano Malavolta
3. Second Reader: Patricia Lago
4. Industrial Supervisor: Magiel Bruntink

Day of the defense: 29 August 2017

Signature from head of MSc committee:

Abstract

Mobile apps are becoming complex systems with numerous functionalities, yet there is a lack of guidance in managing the source code maintainability of these apps. In this Master thesis project a large scale experiment has been performed on 2,238 Android open-source apps, with the purpose of investigating and uncovering maintainability issue evolution trends. The goal of this thesis is to provide the Android developers, project managers and researchers with an up-to-date overview of the maintainability of mobile app's source code, with the help of static code analysis performed on a series of snapshots extracted for each application. This was achieved by applying the static code analysis to every snapshot followed by a statistical assessment of the produced static code analysis metrics. Furthermore, regression models were fitted to each application's evolution of maintainability issues and clustered with the help of k-means clustering technique. In this research maintainability issue evolution trends have been identified for 7 different issue categories allowing the creation of 6 best case practices that provide guidance into keeping a high degree of maintainability of Android apps.

Keywords – Android, maintainability, mobile app, source code quality, k-means

To Tanja, Milorad and Tina

Acknowledgements

I would first like to thank my thesis advisor Ivano Malavolta of the Faculty of Science at VU University Amsterdam. Throughout the whole project, Ivano was always ready to answer my (sometimes quite long) list of questions and steer me in the right direction, while at the same time allowing me enough freedom to explore different possibilities and outcomes.

Special thanks go to Magiel Bruntink, head of the research department at SIG, for providing this research with the necessary expertise. Furthermore, Magiel's thoughtful questions always helped me rethink my approaches and definitely improved the overall quality of this work.

I would also like to thank Jeroen Heijmans from SIG for providing valuable insight into the intricacies of the static code analysis tool, and the complete SIG research team and fellow interns for useful comments and suggestions throughout the project. Also, a big thank you to everyone else at SIG for providing me with the necessary components that constitute this project, and for a warm and friendly environment.

Finally, I must express my deepest gratitude to my parents, my sister and my friends for providing me with unmeasurable support and constant encouragement throughout my studies. Without them, this accomplishment would never see the light of day.

Thank you,

Bojan Filipović

Contents

List of Figures

List of Tables

LIST OF TABLES

1 Introduction

Mobile applications dominate our world today, having reached incredible numbers and their growth shows no signs of slowing down in the near future. Without their ubiquity our lives would be significantly harder. The domination of mobile applications follows from the estimated number of mobile application downloads on the two most common application publishing platforms, Google Play Store and Apple App Store, surpassing 90 billion in 2016. Furthermore, the mobile application market has seen significant increases in application downloads, usage time and revenue paid to the developers. Focusing on the current state of available Android applications on the Google Play Store, Statista[1] chart presented in Figure 1.1 shows that there are more than 2.8 million Android applications available, as of March 2017.

There is an exponential growth trend exhibited in the figure, and an estimated rough average gives between 1,000 - 2,000 applications being published daily. Android applications are not only being published in large numbers, they are also being consumed by users in large numbers. According to the official Android developer website[2] there are now more than 1.5 billion Android application downloads from Google Play Store every month. Furthermore, a study[8] on the mobile developer population shows that there are over 5.9 million developers who chose Android as their first go-to development platform. A platform of such a large scale leads to an extremely crowded market and fierce competition. If developers are to succeed in such a competitive environment, it

[1] https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2] https://developer.android.com/about/index.html

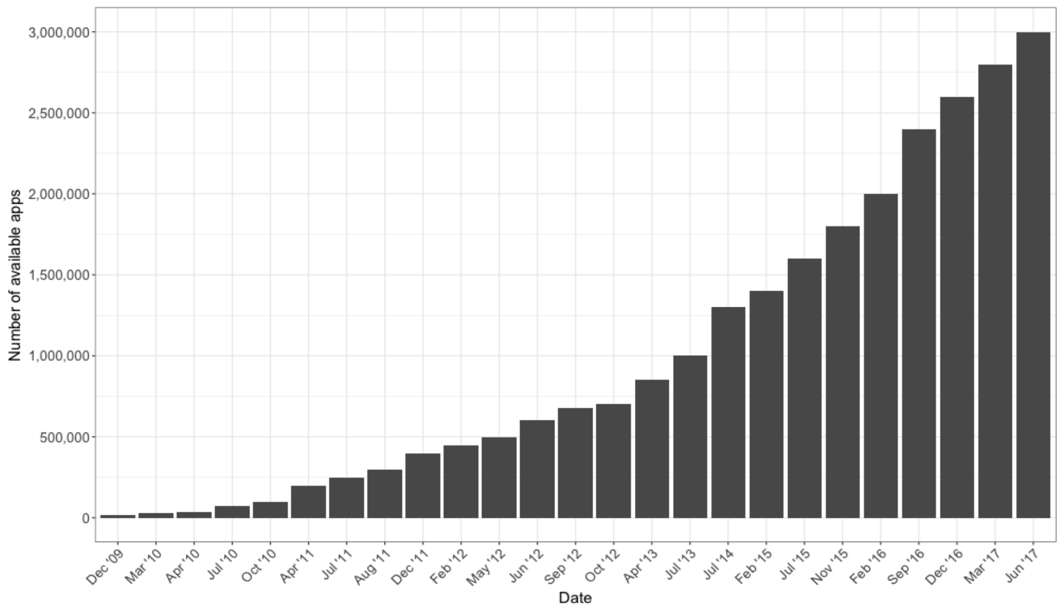
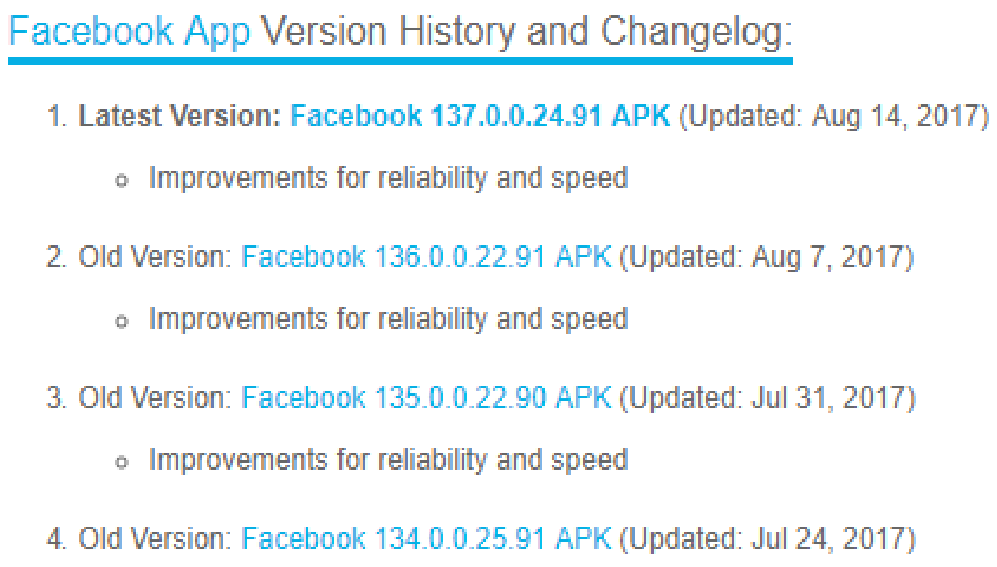


Figure 1.1: Number of available applications in the Google Play Store from December 2009 to March 2017

is of paramount importance that the mobile applications they produce are of extremely high quality.

1.1 Problem Statement

Software maintenance is an important factor in mobile applications lifecycle, as it allows developers to constantly expand their mobile apps and tailor them to their intended user base. Software maintenance is the activity of modifying a software product after its delivery, in order to improve performance, add functionalities or perform corrective tasks on the existing product[7]. Through software maintenance, Android developers can more easily publish new mobile apps updates and provide the users with high quality applications. More importantly, software maintenance amounts for 60 percent on average of the total development costs[14], and it can be seen as one of the most important factors in the mobile application development lifecycle. For example, widely popular mobile apps (*Facebook, Messenger, Instagram*) publish updates for mobile apps consistently once every week. Figure 1.2 presents a screenshot from the Facebook Android



Facebook App Version History and Changelog:

1. Latest Version: Facebook 137.0.0.24.91 APK (Updated: Aug 14, 2017)
   - Improvements for reliability and speed
2. Old Version: Facebook 136.0.0.22.91 APK (Updated: Aug 7, 2017)
   - Improvements for reliability and speed
3. Old Version: Facebook 135.0.0.22.90 APK (Updated: Jul 31, 2017)
   - Improvements for reliability and speed
4. Old Version: Facebook 134.0.0.25.91 APK (Updated: Jul 24, 2017)

Figure 1.2: Facebook for Android app Version History, as found on apk4fun website.

app changelog, as found on the `apk4fun` website[1].

Software maintenance affects software maintainability, which is the property of software that provides insights into how easily a software system (in this case an Android application) can be maintained (more on the topic of software maintenance and maintainability can be found in Background section).

Following presented statistics and information, it is acceptable to assume that such an important factor would be extensively studied and that there would be an abundance of research and available resources that will guide present and future Android developers in conducting software maintenance tasks. For example, software testing, another important factor in the development lifecycle, has been recognized and investigated in numerous occassions, with both academia and industry offering tools and guidances for developers in their testing activities. However, to the best of our knowledge, this is not the case for maintainability of Android applications. Apart from official Android application Core app quality guidelines [2] there is very little evidence describing Android applications maintainability. Furthermore, there is a noticeable lack of resources and guidances providing developers with insights into the maintainability of their Android

[1]https://www.apk4fun.com/history/2430/
[2]https://developer.android.com/develop/quality-guidelines/core-app-quality.html

applications, which hinders the application source code quality and, consequently, increases development costs. Android applications maintainability is still an unknown entity, with a number of unknown points to be explored. This serves as the main motivation for this research, where an investigation into the Android applications maintainability will be conducted, by means of a large scale empirical experiment on Android Open-source applications. **The goal of this research is to uncover maintainability evolution trends across the Android ecosystem and provide maintainability guidelines for Android application developers and researchers.**

1.2 Research Questions

This research conducts an investigation with the focus on density of maintainability issues. For this purpose, maintainability issues are defined as vulnerable sections of source code with the potential risk of increasing maintenance efforts, while density of maintainability issues is defined as the number of identified maintainability issues per Non Commented Kilo Lines of Code (NKLOC). Density of maintainability issues has been chosen as the main metric in order to account for differing application sizes, and it is described in more details in later sections. Aim of the research is to answer the following research questions:

RQ1 How does the density of maintanability issues of Android applications evolve over time?

RQ2 How are maintainability issues distributed across different Android components?

RQ3 How does the size of the development team (number of contributors) impact the density of maintainability issues of Android applications over time?

RQ4 How does the number of source code changes impact the maintainability issue density of an Android application over time?

1.3 Context

This master thesis project consists of performing a large scale empirical experiment on 2,238 Android Open-source applications. The required source code has been mined from Github repositories, taking into account that the apps are both open (i.e. available as Open-source projects in Github) and come from a real-world setting, meaning they are distributed through the Google Play Store market. An analysis of maintainability issue density evolution has been conducted on the available applications. The mobile apps have been processed with a proprietary static code analysis tool to obtain the necessary maintainability issue density evolution metrics. Afterwards, a clustering technique has been applied to cluster the applications according to their similarities in the maintainability issue density evolution trends. Finally, statistical analysis based on the performed clustering has been executed.

1.4 Overview

The rest of this thesis is organized as follows: Chapter 2 provides the reader with necessary understanding of software maintainability and Android specifics, Chapter 3 presents related work conducted on the topics of software evolution and source code quality and Chapter 4 gives an overview into the research questions and methods used in this research. Results of the research are given in Chapter 5, and the Discussion of the results and lessons learned about the evolution trends and their implications are given in Chapter 6. Finally, threats to validity and future directions of the research are presented in Chapters 7 and 8 respectively, and the research is concluded in Chapter 9.

2 Background

This section introduces the notions related to software maintainability, alongside a model for its measurements. Furthermore, a short overview of relevant parts of the Android ecosystem is presented.

2.1 Software Maintainability

Performing source code modifications on an existing software product falls under two terms: Maintenance and Maintainability. While there are numerous definitions given for these two terms, this research focuses on the definitions given by the ISO/IEC/IEEE 14764[1] standards.

Software Maintenance is defined as *"The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment."*

From the given definition, maintenance process can be further split into these four activities:

Corrective Modification of a software product that corrects and solves previously identified and diagnosed errors (such as bugs reported by users).

Adaptive Modifications performed on a software product that keep the product usable when the software environment changes (such as updates to the OS).

Preventive Software product modifications that increase its reliability and prevent faults in the future.

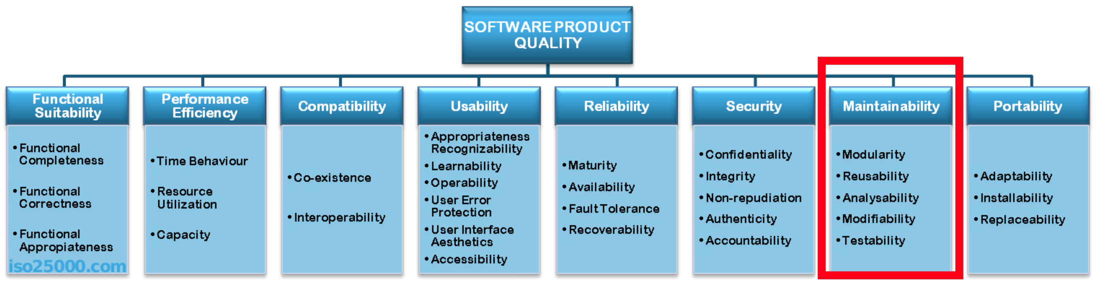


Figure 2.1: ISO 25010 Software Product Quality Model.

Perfective Functional enhancements performed on a software product, often following new user requirements.

While software maintenance is the process of modifying existing pieces of software, software maintainability is the factor that explains how well this process can be executed on an existing software product. Definition of the term comes from the ISO/IEC 9126 standard, which introduces the Software Quality Model. This model presents qualities and characteristics to be taken into account during evaluation of a software system's properties. **Software Maintainability** is defined in this model among other properties, and the definition states that it is *"The degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements."*[2]

Furthermore, the Software Quality Model further divides Software Maintainability into five additional sub-characteristics that compose it.

Modularity Degree of impact of changing one component in relation to others.

Reusability Degree to which an asset can be used in building other systems.

Analysability Extent to which a software product can be analyzed, with the goal of identifying parts to be modified.

Modifiability The extent to which a software product can be modified without lowering its quality.

Testability The extent to which a software product can be tested.

2.1.1 SIG Maintainability Model

The Software Improvement Group (SIG)[1] is a software consultancy company providing insights into software systems' source code quality. Throughout multiple years of expertise they have developed a maintainability measurement model and the accompanying tooling to support it, which are used to conduct this research. The model measures maintainability and its subcharacteristics, as defined by the ISO standards, and is relying on source code measurements rather than functional or behavioural characteristics of a software product. As such, this model is vital for this research due to its compliance with the research goal. The maintainability in the model is expressed on a star rating scale ranging from 1 (lowest maintainability) to 5 (highest maintainability) stars. Furthermore, the star ratings come from a proprietary benchmark of systems coming from different industries and is calibrated yearly to provide a state-of-the-art insight into current market maintainability, alongside increasing the model's correctness and objectivity. More details on the model description can be found in [12]. Measuring software product properties allows the model to assess the quality characteristics of maintainability and its sub-characteristics. These software product properties consist of:

- **Volume:** Overall system source code size.
- **Duplication:** Degree of duplication in a system's source code.
- **Unit Size:** Source code size contained in system's units.
- **Unit Complexity:** Complexity of source code contained in system's units.
- **Unit Interfacing:** Size of the interfaces of units in terms of interface parameter declarations.
- **Module Coupling:** Degree of coupling between system's modules in terms of incoming dependencies. The definition of module is expressed as a grouping of related units in a system. In the context of Java development, a module is considered to be a `class`.
- **Component Balance:** The product of the rating for size distribution of system components and the rating for the amount of components in a system. Component is defined as group of source code modules.

[1]https://www.sig.eu/

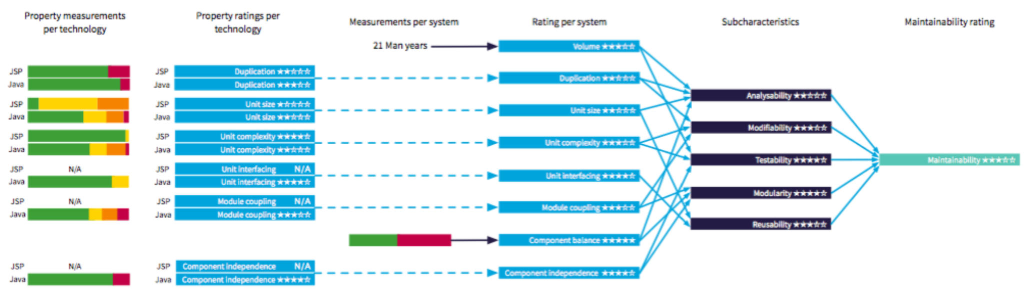


Figure 2.2: The process of measuring maintainability and its sub-characteristics of a software system and the aggregation of results. Adopted from the process presented in [18].

- **Component Independence:** Expressed as a rating of source code contained in modules that have no incoming dependencies from other modules.

The model measures these software product properties on the system's source code on four different levels: unit, module, component and overall system level. Furthermore, the model supports multiple programming languages and allows measurements to be done on the source code of a software system composed of multiple programming languages. For this research, we will only be focused on the overall system level metrics of Android applications under analysis. In the same way, as most of the Android applications are written in Java programming language, we will only take into account the measurements and maintainability ratings for this programming language and exclude other possible languages, such as *C* or*xml*. Furthermore, this study focuses on only one programming language in conducting the research in order to isolate the study with respect to possible sources of bias coming from considering multiple programming languages, such as inter-technology dependencies and underlying differences of multi-language programming.

2.2 Android Development Lifecycle

Android platform is an open-source, Linux-based software stack that allows the creation of software for a variety of devices, including smarthphones, wearables, cars and TVs. The features in the Android OS are made available to the developers through the use of Java APIs, and Java is considered as the main programming language used to build applications. As mentioned before, according to the study conducted by the EDC corporation[8] there are now over 5.9 million developers who chose Android as their first go-to development platform. In this section an overview into the details of Android mobile app development is given, with provided details into the Android framework specifics relatable to this research. Android developers can obtain necessary guidelines and specifics from the official Android developers website[1]. This resource includes specifics related to Material Design, a guideline into the UI design of Android applications, various trainings and samples for developers, as well as the official documentation reference to the specific code-based implementations of components. In this research Android application components are evaluated with respect to their maintainability issue evolution, therefore presented below are details about Android application components and the relation between Android application development and source code quality.

2.2.1 Android Application Components

All Android applications consist of essential building blocks, called *App Components*, and they represent the entry points through which users or other systems can interact with an application. There are four different App components, `Activities`, `Services`, `BroadcastReceivers` and `ContentProviders`, each serving a different purpose and following a different lifecycle. Figure 2.3 presents the App Components and their relation to the Android system. Below is a short introduction to each of the components.

The main entry point for interacting with the user, represented by a single screen containing user interface elements, is called an **Activity**. For example, a to-do list application might contain one Activity presenting the overview of the current items in the to-do list, and another one for composing new items for the to-do list. Activities can communicate with each other and exchange data, or even start other application's Activities, however they are independent from each other. Main interactions that an activity fosters between itself and the system include resource management, keeping track of user interactions, and providing a way for applications to implement user flow.

A **Service** represents an entry point for running applications in the background. Services do not provide user interfaces, rather they are used to perform long-running

[1]https://developer.android.com

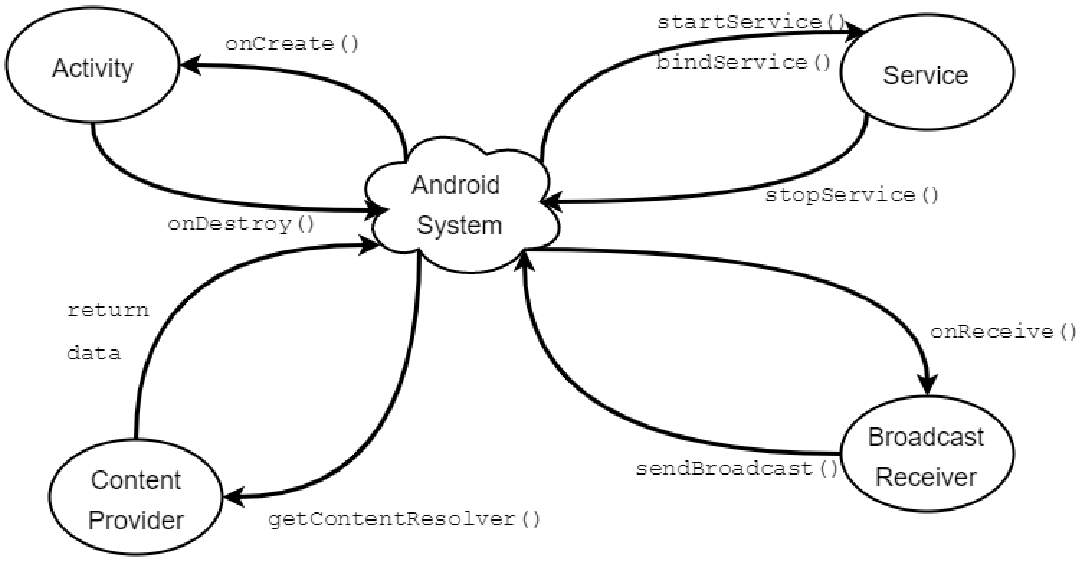


Figure 2.3: App components and their lifecycles in relation to Android System. Arrows note the method calls that interact with the components.

processes in the background. Services allow the uninterrupted user interactions while performing their tasks in the background, such as playing music or downloading data over the network. There are two main types of services, with respect to the user awareness of their execution. Should a service be considered important to be running uninterruptedly, such as playing music in the background, the system can prioritize resources to ensure this service is running until it finishes its process. Similarly, services that are not considered crucial can be interrupted in their execution by the system, to allocate resources to a more desirable component and then return to the execution at a later point. The flexibility of services allows for building numerous higher-level system concepts, such as notification listeners, screen savers or different input methods.

BroadcastReceivers enable the system delivery of events to an application outside of the regular user flow, therefore allowing the application to respond to system-wide broadcast announcements, even if the application is not running at the moment. For example, while scheduling alarms for a clock application, by the use of a BroadcastReceiver it is not necessary for the application to remain running until the alarm goes off. Broadcasts can originate directly from the system, such as low battery announcement broadcast, or a broadcast announcing that a picture has been captured and is readily available for other applications to use. Most commonly, BroadcastReceivers are used as gateways to other components and are intended to perform a minimal amount of work. They do not display a user interface, however they can create a status bar notification to alert the user when a broadcasting event has occurred.

Through the use of **ContentProviders**, applications can share a set of app data stored somewhere in the file system, a database or on the web. With this component, other applications can query or modify the shared data if the content provider allows it. One example of a content provider is the user's contact management content provider, provided by the Android system. Applications with the proper permissions can request this information and modify particular person entries in the contact list. Using an URI scheme, content providers define what resources can be accessed by different applications. Furthermore, content providers are also useful for reading and writing data that is private for the application and not shared. For example, the to-do list application can use a content provider to save to-do items.

The main communication between these different application components is achieved through the use of `Intents`. Furthermore, intents can be used to activate three out of four of the defined App components. In short, Intents are defined as objects carrying a message that activates a specific component or a specific type of component. Intents are not in the scope of this research, however they are extensively used in the Android application development and more information can be found on the Android developer guide website[1].

2.2.2 Android Source Code Quality

Apart from presenting the Android application components, it is worth mentioning the relation between Android development and the underlying source code quality. In this context, some research has been conducted and some freely-available tools exist to guide developers in achieveing higher source code quality for their applications. The related research done on source code quality is presented in the next chapter (See Section 3), and below is a short overview of the source code quality inspection tools available for Android developers.

Although the research conducted in [9] found no significance in their investigation of impact of source code quality on an application's market success, the impact of source code quality surely exists on the resources required for maintenance efforts once that

[1]https://developer.android.com/reference/android/content/Intent.html

application has been published. In order to minimize these costs, developers can track and improve their source code quality with the help of freely-available tools. One such tool is **Lint**[1], an Android Studio code scanning tool that helps developers identify and correct problems with the structural quality of the application's code. The tool reports on the description of the identified problem as well as its severity, allowing developers to prioritize different reported problems. The Lint tool provides support in finding potential bugs and optimization improvements with respect to security, performance, usability, correctness and internationalization.

Another open-source solution that utilizes static code analysis to find bugs in Java code is **FindBugs**. The analysis engine of FindBugs reports nearly 300 different bug patterns, and similarly to Lint, reported bugs are categorized and assigned a priority level. FindBugs has been used by Google to evaluate the bugs in their Java-based software. More information about the usage of FindBugs can be found at their official website[2].

PMD[3] is a popular source code analyzer, able to find common programming issues such as empty catch blocks, unused variables etc. PMD supports a variety of different languages, among which are Java and XML, making it suitable for analyzing source code of Android applications. Furthermore, PMD provides support for identificaiton of duplicated code in Java source files.

Another freely-available tool is **CheckStyle**[4]. Although it does not identify bugs or programming faults, CheckStyle allows Java developers to write code that adheres to a suitable coding standard, therefore increasing the code's readability. For example, developers can configure CheckStyle to follow Google Java Style guidelines that will make their Android application's source code follow a predefined coding standard.

SIG developed a handy tool allowing developers to continuously track the source code maintainability of their mobile apps, called **Better Code Hub**[5]. With this tool, the developers can easily connect their Github repositories and measure source code maintainability, receiving a report on the ten maintainability guidelines as a result,

[1]https://developer.android.com/studio/write/lint.html
[2]http://findbugs.sourceforge.net
[3]https://pmd.github.io
[4]http://checkstyle.sourceforge.net
[5]https://bettercodehub.com/

alongside suggestions for improvement or refactorings. The ten maintainability guidelines have been derived from the book by Visser et al. [37].

Although the presented tools provide some support to developers with respect to managing the source code of their mobile apps, this research has opted to use the Software Analysis Toolkit (SAT) provided to researchers by SIG (more details about the tooling in Section 4.3.2). The SAT tool allows for execution of static code analysis on different app snapshots, which proved to be of great help for this research. This tool allowed the automatic application of static code analysis on multiple apps, and provided the resulting metrics in a simple manner that allowed further statistical analysis. Furthermore, the tool follows the maintainability definitions from the ISO Quality model, and provides a comprehensive report on both the identified maintainability issues residing within the app's source code and the identified code smells, therefore providing a deep insight into the app's source code quality.

3

Related Work

This section discusses related work with respect to software evolution and Android source code quality, with the addition of papers that have used similar software repository mining techniques as this research.

Hecht et al.[17] approached the software quality of Android applications by introducing a tooled approach called PAPRIKA. The tool was designed to perform static code analysis on Android application's bytecode and detect software antipatterns, defined as poor design choices, appearing throughout the application's evolution. PAPRIKA produces a total of 34 different Object-Oriented (OO) and Android-specific metrics. In the paper, Hecht et al. analyzed the mobile application's quality evolution across 3,568 versions of 106 different Android applications obtained from the Google Play Store. They have been able to identify relationships between antipatterns and five different quality evolution trends. This paper ties into our research due to similar methodologies and focus on quality aspects and their evolution in the context of mobile (Android) applications. Similarly, in our work we use a tooled approach to measure the source code quality. Furthermore, we take evolution aspects into account, however we focus on maintainability related issues rather than software antipatterns, and our research scope involves analysis of over 2 thousand Android applications, compared to the 106 analyzed in this paper.

Di Penta et al. [30] analyzed the evolution trends of vulnerabilities detected in source code. For the detection of vulnerable source code lines, they have used 3 different static code analysis tools, namely Splint, Rats and Pixy. The paper uses different static code analysis tools to cover a wider range of detected vulnerability categories and programming language support. The 3 different networking systems were analyzed by means of executing the static code analysis tools on system's extracted snapshots, which is methodologically similar to our work, with our research having a larger frequency between each snapshot. Furthermore, the goal and research questions of this paper closely align with our goals of uncovering evolution trends, with a different focus on examined properties. However, the subjects and therefore, the outcomes of the research differ from this work, as we are using only one static code analysis tool to analyze the evolution and cover evolution of maintainability issues as opposed to vulnerabilities in source code.

Bad code smells are defined as symptoms of poor design choices and implementation, and their importance has been investigated in a study conducted by Tufano et al. [36]. In this study, the researchers set out to answer *when* and *why* bad code smells are introduced in in a software project. Their large scale empirical study involves investigating the circumstances and rationales behind bad code smells introduction, and is conducted on a change history of 200 open source projects. The change history was analyzed in the context of mining over 0.5M commits and an additional manual analysis of smell-inducing commits. Analysis of the results showed that most smell instances are introduced upon adding a software entity into the whole system, which is contrary to established common wisdom of smells being a result of maintenance activities. They have also found that smells are being introduced rather rapidly, with sudden increases in smell-uncovering metrics. Furthermore, smells manifest themselves while the developers are enhancing existing or implementing new features, and there is more smells introduced with developers who exhibit a higher workload. Our research contains a narrower scope than the wide variety of systems analyzed in this paper, however our research does contain more instances of analyzed applications. Furthermore, there is a similarity in the paper's research questions with our second research question.

Similarly to our research, Koch [25] set out to analyze the evolution of open-source software systems on a large scale. Utilizing the data for 8,621 projects coming from SourceForge, the evolutionary behaviour of the systems was characterized by applying both linear and quadratic models to the systems, where the quadratic model outperformed the linear one. Furthermore, the evolutionary behaviour has been modelled as a function of lines of code and time since first commit. Although this work focuses on general evolutionary traits of different software systems, compared to our focus on maintainability evolution, the methodology is largely similar. Both our research and Koch's paper focus on large-scale, open-source systems. However, we focus on Android applications, and have applied two additional regression models to examine the maintainability evolution behaviour, Cubic and Quartic.

4 Study Design

In this section an overview of the research questions in this thesis is given. First, a short introduction to research questions and their main objectives is given. Afterwards, a more detailed approach to answering each research question is presented, alongside the explanations of statistical methods. In this research, main focus is on answering questions related to evolutionary aspects of different application's properties and investigating whether source code properties or Android-specific properties have an impact on maintainability of these applications. Therefore, four main research questions have been defined:

RQ1 How does the maintainability issue density of Android applications evolve over time?

RQ2 How are maintainability issues distributed across different Android components?

RQ3 How does the size of the development team (number of contributors) impact the maintainability issue density of Android applications over time?

RQ4 How does the number of source code changes impact the maintainability issue density of an Android application over time?

For **RQ1**, the main objective is to investigate whether the evolution of maintainability issue densities across different Android applications exhibits identifiable trends. With this question, the research aims to provide an understanding into the evolutionary trends of the amount of maintainability issues that can be used as a guidance in maintenance and development planning for Android developers. This trend analysis is conducted by clustering applications with respect to their similarities in maintainability issue density evolutions. The identified clusters of similar applications are then used as a foundation for investigating the different relationships between applications belonging to the same or different clusters. In this research question a method for automatically clustering the applications based on their evolution of maintainability issue densities is given. This method involves calculating different regression models for each application and subsequently clustering these applications based on their similarities. This approach has been described in more detail later in this section. In this research, multiple different categories of maintainability issues have been identified and therefore the clustering technique is applied to each of the maintainability issue subcategories separately. The resulting clusters of applications are also reported on each of the identified subcategories.

In **RQ2**, the research aims at providing an insight into the relations between Android-specific components and the distributions of different maintainability issues across these components. Identifying meaningful relations between Android-specific components and the distributions of different maintainability issue in those would give rise to the most and least affected components, as well as provide guidance into the maintenance efforts related to one or more of these components. In this research, an Android-specific component is a collection of source code units belonging to one of the following four Android framework components: Activity, Service, BroadcastReceiver or Content-Provider. Later in this section a more detailed overview into the process of identification of Android-specific components for each application is given, and the resulting identified components are reported. Furthermore, the components are related to each of the identified maintainability issues, to allow further investigation into the relations between Android-specific components and the distributions of maintainability issues. The correlations between different Android components and issue categories will be statistically assessed.

For **RQ3**, the main objective is to test the effect of the development team size on the different maintainability issue densities. Specifically, interest here is to investigate whether variability in the size of development team impacts the amount of maintainability issues positively or negatively. In this research, the development team size is defined as the number of contributors that have made at least one commit to the Android application's source code, excluding commits that have been committed to the application's repository but do not relate to the application's source code (for example the application's server side source code). The approach involves mining the number of contributors for each application that is described in more detail later in this section, and applying the necessary statistical correlation tests to investigate the significance and impact of the variability of number of contributors on the variability of maintainability issue density. This research question provides the reader with an overview into the desired management of the development team size and the possible positive or negative outcomes that increasing or decreasing the team size could bring on the maintenance efforts involved in maintaining the application's source code.

Finally, **RQ4** follows a similar approach to RQ3, where the exploration is focused on the investigation of impact of source code changes made to each of the obtained application's snapshot on that application's maintainability issue density. Source code changes, or *Code Churn*[13], is defined by the static code analysis tool as the sum of different source code modifications, namely additions, modifications and removals of source code lines, committed to the appropriate application's source code. Therefore, the correlation between the number of source code changes and maintainability issue density at each snapshot of the application is investigated and reported. Source code changes metrics have been extracted from the static code analysis tools, and appropriate correlation statistical tests are applied to each of the before identified clusters. The results are reported for each cluster as well as for each of the identified maintainability issue subcategories. More details on static code analysis metrics can be found later in this section. Having a quantifiable relationship between the number of source code changes and its impact on maintainability issue density provides the reader with an insight into the current and possible future state of maintainability of the developed application.

4.1 Goal-Question-Metric Paradigm

In this section the relations between research questions and the metrics used to answer them is presented. The approach is based on the Goal Question Metrics Paradigm, presented by Basili[6]. GQM approach is commonly used in software engineering empirical research to define measurement programs. It provides an outline that guides researchers in defining goals, refining these goals into questions and finally specifying measurements and data that needs to be collected in order to answer those questions.

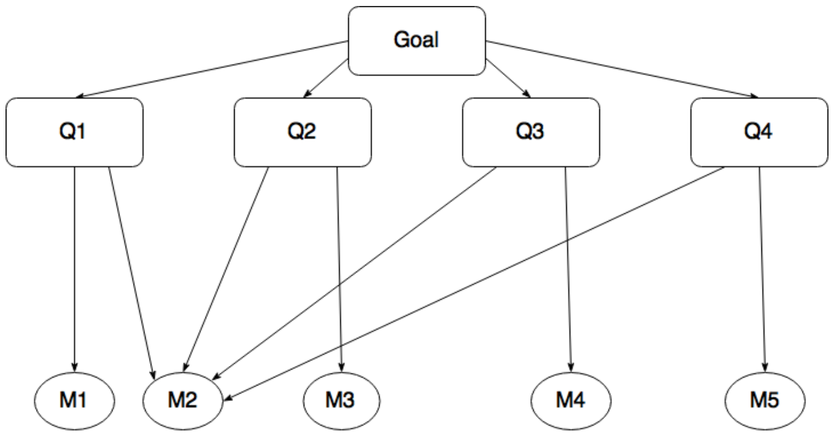


Figure 4.1: Overview of the GQM Tree with relations between questions and metrics.

Therefore, the starting point of the GQM paradigm is to define the ***research goal***, which is defined as:

To analyze	the source code of Android mobile applications
for the purpose of	characterizing and determining best practices
with respect to	its maintainability
from the point of view of	software developers and researchers
in the context of	open-source applications published in the Google Play Store.

As this research is heavily based on measurements of different maintainability issues and their evolution, questions derived from the research goal match the research questions posed earlier in this section 4, therefore for this step four different questions were defined that will answer the research goal. Once the questions have been derived, it is necessary to extract meaningful metrics and data that needs to be collected in order to answer these questions. The metrics are listed below.

	Metric
M1	# of Clusters
M2	Issue density
M3	Types of Android components
M4	Number of Contributors
M5	Code Churn per snapshot

Finally, the relationship between the Questions and Metrics is presented in Figure 4.1 as a *GQM Tree.*

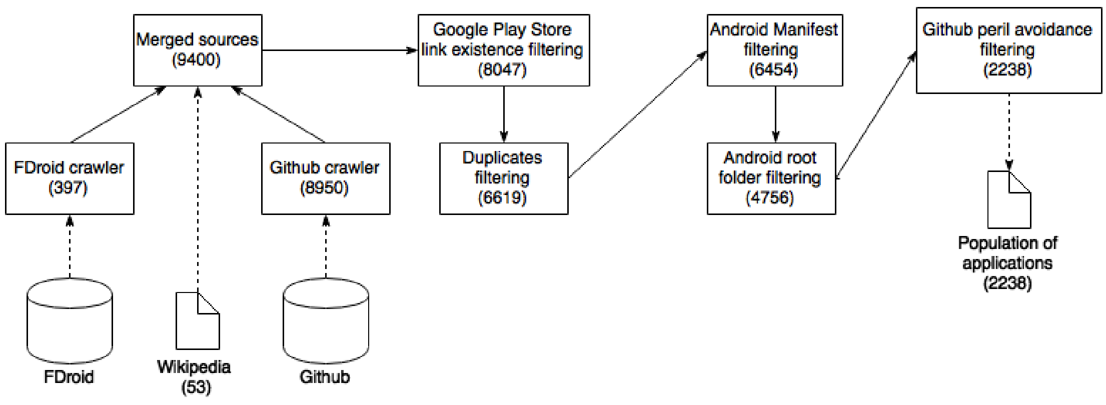

Merged sources (9400)
Google Play Store link existence filtering (8047)
Android Manifest filtering (6454)
Github peril avoidance filtering (2238)
FDroid crawler (397)
Github crawler (8950)
Duplicates filtering (6619)
Android root folder filtering (4756)
Population of applications (2238)
FDroid
Wikipedia (53)
Github

Figure 4.2: An overview of the data collection process.

4.2 Data Collection

In this section a detailed overview into the data collection and extraction process is given. Identifying the required target population of applications for this research requires applying several filtering strategies, which are documented alongside the respective numbers of applications in each filtering step. See Figure 4.2 for an overview of the process.

Initial collection of Android open-source application originates from three different sources. This approach is largely based on the data collection process described in the work done by Das et al.[10] and includes the same originating sources. Differences between the approaches regard the inclusion of additional filter in the filtering process, namely the Github peril avoidance filtering and the larger starting samples of apps originating from the initial sources, as to make it more suitable for this research. Github[1] is a well-known online version control management system where numerous developers store, manage and maintain the source code of their projects. From Github, a custom based search was performed that targeted all the repositories containing a link to Google Play Store application page in their readme files. As the study is focused on real-world examples of open-source applications, with this step Android repositories that contain unpublished applications have been excluded from the search. After the initial step of mining the Github repositories, **8,950** applications have been obtained. Second

[1] https://github.com

source for the dataset includes FDroid[1], a largely known online catalogue of free and open-source Android projects. From this catalogue, a search was applied that locates applications that contain: a) a link to the respective Github repository, and b) a link to the respective Google Play Store page for the application. Mining the FDroid repository resulted in **397** identified applications.

One small note: At the time of writing this paper (August 2017) the search functionality on FDroid appears to be broken or not working. Furthermore, the link[2] that was used in the mining script does not exist anymore.

Last source of the initial list of applications comes from the Wikipedia[3] list of free and open-source Android applications. From this list, a manual selection was performed to select the applications that, again, contain a link to the respective Github repository and are published on Google Play Store. Wikipedia provided a list of **53** applications and the final list of applications after the initial mining step resulted in a total of **9,400** applications.

After the initial collection from the three sources with respective filterings based on availability of Github repositories and Google Play Store links, next step in the process is to identify the Android application package and the actual Google Play Store page link from the respective Github readme files. This step was done for all applications and resulted in a list of **8,047** applications. Some application's, although containing a link to the Google Play Store page in their readme files, were not actually existing on the Google Play Store, and have therefore been excluded. This could happen if the developers decide to unpublish the mobile app from the store or if Google decides to take the app down for violating the necessary publishing policies. Next, with the help of scripts, the three different sources were merged together and the duplicate entries were removed, which leads to a list of **6,619** applications. The dataset at this point contains entries for applications that contain respective links to the Github repositories where the source code for each application is contained, their Google Play Store application page links and the source from where the applications originated (G for Github, F for FDroid and W for Wikipedia).

[1] https://f-droid.org

[2]https://f-droid.org/forums/search/

[3] https://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications

Proceeding with the filtering phase, it is necessary to identify which repositories contain actual Android application source code. This is done by means of identifying the availability of Android Manifest files in the application's repository. As mentioned on the official Android developers website[1], every application must contain an `AndroidManifest.xml` file located in the application's root directory. This manifest file contains all the essential application information supplied to the Android system, allowing it to run the actual application code. For example, Manifest file is in charge of naming the Java package for the application, which serves as an unique identifier of each application (e.g. com.bojan.application1). Furthermore, the Manifest file is of vital importance to the application as it supplies the Android system with vital application information. Information contained in the Manifest includes describing the application's Android components and running processes, declaring the required permissions that the user needs to grant to the application in order for functionalities to operate properly and specifying the minimum required API level for the application. Last but not least, the Manifest file describes the third-party libraries the application uses. Therefore, repositories that do not contain the appropriate AndroidManifest.xml file are not actual Android applications and should be excluded from the list. With the exclusion of repositories that do not contain an actual AndroidManifest.xml file in their directories, the final list now contains **6,454** applications. Penultimate filtering step involves managing the application's root folders containing the actual Android source code for each application. In this step, for each application, the Android Manifest file is located in the repository structure and the root folder that contains the file is extracted. In this research, combined with the Google Android website guidelines about Android Manifest files, the rationale is that the Manifest must be in the application's root folder, therefore the directory containing the Android Manifest file should also contain the complete source code for each application. The identification of the application's root source code folder is crucial for later stages of this research, as it is possible that the application's repository contains redundant code, such as source code created for the same application but for different mobile platforms, different server side code, associated application websites, etc. Therefore, with this filtering step, it is possible to specify the exact relevant directories used later in the research during static code analysis steps. The final dataset version now contains **4,756** mobile apps. At

[1] https://developer.android.com/guide/topics/manifest/manifest-intro.html

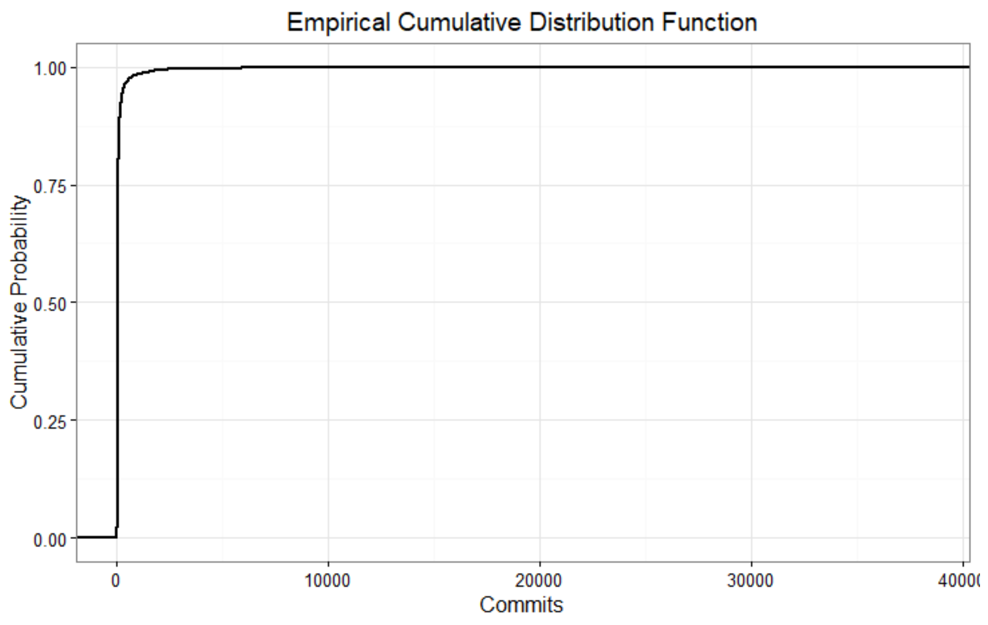


Figure 4.3: Empirical Cumulative Distribution of commits in the dataset.

Table 4.1: Demographic information on Commits and Lifetime (in Weeks) of applications in the initial dataset

	Min	1st Quartile	Median	Mean	3rd Quartile	Maximum
Commits	1	5	13	105.2	46	38340
Lifetime	0	0.0066	5.844	34.4	45.42	693

this stage, the dataset contains a large list of applications. However, additional steps need to be taken before the list is finalized. These steps include gathering additional metadata about each repository and therefore, each application. For each repository, application development lifetime has been extracted alongside the number of commits. The application development lifetime is defined as the range between the first commit contributed to the repository, and the last commit contributed to the repository at the time of running the data mining script (2 May 2017). Therefore, *Lifetime* of an application is measured between the two timestamps corresponding to each of the aforementioned commits. Having the number of commits and lifetime metrics already available allows for some short demographic analysis of the application dataset.

This demographic repository data closely follows the findings uncovered in the study conducted by Kalliamvakou et al.[22]. In this study, the researchers set out to investigate the quality and properties of the available Github data, with the purpose of guiding researchers in conducting software engineering studies based on mined Github repository data. They have been able to identify nine different perils researchers might come across while mining Github repositories, and provided peril avoidance strategies respectively. In this study, two of the identified perils are relatable to the research, namely:

(a) Most projects have very few commits, and

(b) Most projects are inactive.

Regarding peril a), the study shows that most of the projects in Github exhibit a low amount of commits, with the median being 6 and with more than 90% of projects having less than 50 commits. As a peril avoidance strategy, the study suggests that the researchers should consider the number of recent commits and pull requests. Furthermore, for peril b), the study found that 32% of the projects have been active for only one day. This information suggests that these repositories are being used to either archive source code or are being used for testing purposes. Therefore, a peril avoidance strategy proposed by the paper suggests reviewing the description and readme files in order to examine whether the repository fits the research needs. Following from the perils presented in the paper, it is to be expected that some repositories are not used for the development of the application, but rather for archiving source code or for testing purposes.

What can be concluded from Figure 4.3 and Table 4.1 is that a large amount of applications in the current dataset either have very few commits, with half of the application's having 13 or fewer commits, or exhibit short lifetimes, where ~53% of applications have a lifetime shorter than 8 weeks. Regarding the perils and avoidance strategies mentioned in the study and taking into the account the presented information, additional filtering of inactive repositories has been applied to the dataset. In order for a repository to be classified as active and viable for this research, it needs to exhibit the following characteristics:

(1) The repository must have 6 or more commits

(2) The repository lifetime span must be at least 8 weeks

The first characteristic corresponds to the median number of commits in the presented study, while for the second characteristics, 8 weeks have been chosen to represent two months of activity. Upon applying the specified additional filters, the repositories

Table 4.2: Summarized data collection process and filters

Filter ID	Filtering strategy	#Apps
1	Repository must contain links to Github and Google Play Store pages	9,400
2	Google Play Store page for the application must exist	8,047
3	Removing duplicates from 3 sources	6,619
4	Repository must contain appropriate AndroidManifest.xml file	6,454
5	Repository must contain a viable application root folder	4,756
6	Github mining peril avoidance strategies, >= 6 commits, >= 8 weeks lifetime	2,238

in the current dataset are now related to published open-source Android applications. The dataset now contains a list of **2,238** actively maintained repositories. Finally, Table 4.2 presents the summarized data collection approach with the applied filters and the respective number of applications for each step.

4.3 Data Extraction

Upon obtaining the dataset containing **2,238** Android open-source applications, next step in the research is to apply the static code analysis to each of the applications. For this purpose, a proprietary, industry-tested static code analysis tool called System Analysis Toolkit (SAT) is used, provided to the researchers by SIG. For each application, analysis is conducted on a series of snapshots of the application's source code. In this section, the steps in the data extraction process are outlined and reported. The extracted information forms the raw data that is used for the empirical analysis of software maintainability issue evolution.

4.3.1 Snapshots Extraction

Since the interest of this research is to perform a fine-grained analysis of maintainability issue evolution, it is necessary to examine different versions of source code files for each application under analysis. These different versions stem from creating a snapshot series for each application. The process of extracting snapshots has been the most time-consuming, due to some apps having large lifetimes and therefore numerous snapshots in their series. For example, the largest snapshot series contains 693 snapshots, which means creating 693 different sets of source code files. In total, the whole analysis process of extracting snapshots and applying static code analysis to each snapshot spanned a period of 2 months. Furthermore, a total of 800 million LOC has been processed, and 7,205.83 GB of source code is contained within the 123,261 snapshots.

Particularly, this research considers the evolution of maintainability issues of an Android application as a sequence of Snapshots $(S_1, S_2, ..., S_n)$. A time-windowing approach has been adopted and closely follows the approach presented in the work by Di Penta et al.[30]. Using this approach, it is possible to define a **Snapshot** as a *set of source code files of an application at a given point in time.* The sequence of snapshots for a given application constitutes the snapshot series. A snapshot series can be extracted from an application's Github repository history log. For this research, the time interval between two snapshots in a snapshot series is defined as 604,800 seconds, which evaluates to 1 week. Specifically, one week has been chosen as the desired time period based on the study conducted by [17], where it has been shown that mobile apps get updates once a week or less frequently. Furthermore, in introduction section it can be seen that most widely popular mobile apps have weekly updates. Therefore, choosing this time period as a baseline allows for some source code evolution to happen in between the two measurement points. Figure 4.4 gives a visual representation of a Snapshot Series.

Starting from the initial snapshot, defined at t_{start}, the total number of snapshots is $lifetime + 1$. Therefore, if an application has a development lifetime spanning 10 weeks, a snapshot series will contain 11 snapshots, each containing a set of source code files for that application in the given week $S_{series} = (S_{w0}, S_{w1}, ..., S_{w10})$.

Once the snapshot series has been obtained, each application's repository is cloned and subsequently checked out for each snapshot in the series. It is worth mentioning that the data mining snapshot extraction and analysis script allows for user specification of a different time window, measured in seconds, and can therefore allow for a more coarse-or-fine grained analysis of the applications' issue evolution. Once the snapshot series for each application has been obtained and the required sets of source code files

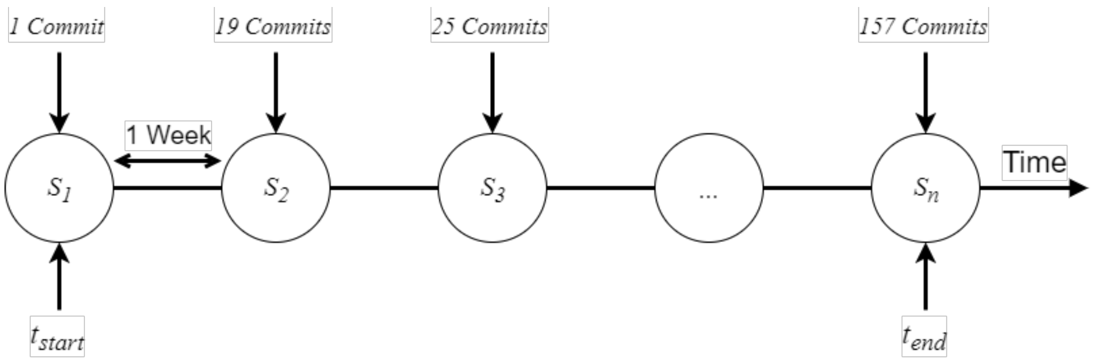


Figure 4.4: An overview of the snapshot extraction process. Each circle denotes a set of source code files for an application. Start and end points for snapshot extraction correspond to starting and ending timestamps, respectively. Notice that the number of Commits can vary between snapshots.

of each application have been stored, the applications are ready for static code analysis applied on each snapshot in the snapshot series.

4.3.2 Static Code Analysis with SAT

In this section a report on the usage of Software Analysis Toolkit (SAT), developed and made available to the researchers by Software Improvement Group (SIG) is given. In this research, SAT has been extensively used to process every snapshot for each application, producing numerous usable metrics related to maintainability of an application and providing insights into the maintainability issue evolution. Figure 4.5 presents an overview into data processing executed for each application in the dataset. The total static code analysis processing of snapshots took 12.45 days, with every snapshot taking 8.73 seconds on average to analyze.

Software Analysis Toolkit (SAT) is a tool that automatically analyzes software maintainability, based on SIG Maintainability Model (explained in Section 2.1.1). SAT has been used extensively by SIG to support consultancy services and management decision making, evaluating a broad variety of systems stemming from industries such as finance/insurance, government, logistics, IT and others. The tool produces source code metrics for the system under analysis and is constituted from two different parts, namely the Analysis tooling and the Software Monitoring service. In this research the focus is on historical analysis of the applications' source code, therefore the usage of

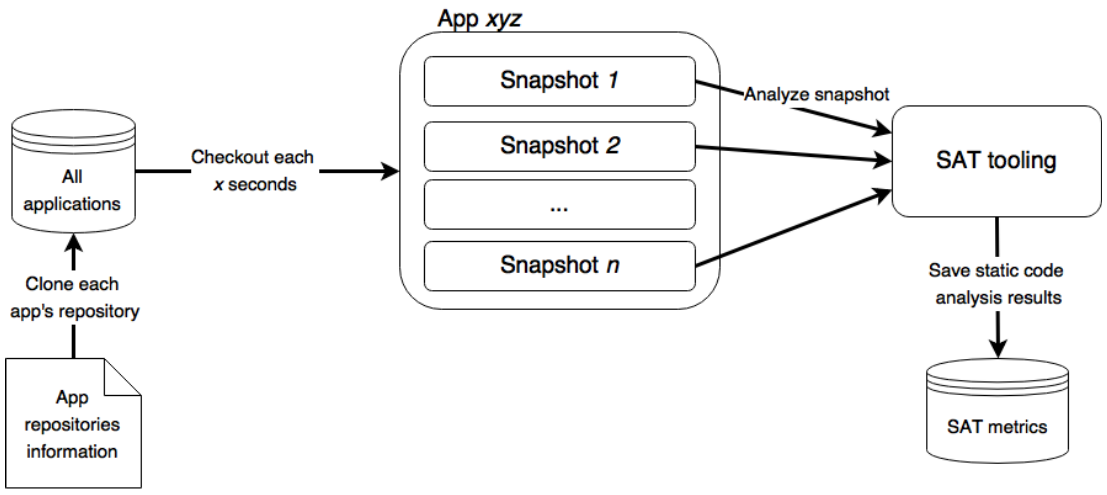

App *xyz*
Snapshot *1*
Snapshot *2*
...
Snapshot *n*
Analyze snapshot
SAT tooling
All applications
Checkout each *x* seconds
Clone each app's repository
App repositories information
Save static code analysis results
SAT metrics

Figure 4.5: Overview of the data processing steps. x represents user-specified time-window for checking out apps (604,800 seconds or 1 week used in this reseach), while n is the total number of snapshots for each app.

SAT is limited on the analysis tooling. The analysis tooling allows for a specification of a snapshot series for each application and therefore produces a set of metrics for each snapshot in the series.

As mentioned before, measurements in SAT are based on the use of SIG Maintainability Model, supported by the ISO/IEC quality model. These measurements are presented as a star rating scale, ranging from 1 (least maintainable) to 5 (most maintainable) rating. The measurement and evaluation process follows these steps:

(1) SAT calculates the software product quality source code based metrics, which include *Volume, Unit Size, Unit Complexity, Unit Interfacing, Module Coupling, Duplication, Component Balance and Component Independence.*

(2) From the software product quality metrics and by utilizing SIG's proprietary, yearly calibrated benchmark of systems, star ratings are derived for each of the Maintainability Model's subcharacteristics: *Analyzability, Modifiability, Testability, Modularity and Reusability.* For more detailed explanations of these subcharacteristics and their relations to software product quality metrics, refer to Section 2.1.

(3) Measures for each of the subcharacteristics are aggregated together to produce a unified star rating of a software system's maintainability.

The Maintainability Model as well as SAT tool have been used and accepted by numerous SIG's clinets, including ING[1], KLM[2], Waternet[3] and Apotti[4], and the approach used in measuring software maintainability allows for comparisons between different software systems. In this research the focus will not be on comparing the star rating characteristics, rather, the focus is on evaluating the evolution of the density of maintainability issues, for which the metrics will be derived from the outputs of the SAT. This design decision has been made in order to avoid the bias of not being able to replicate the study, due to proprietary tooling constraints. By design SAT has been developed to be technology independent and support multiple programming languages and a wide range of different software systems. With such a wide variety of possibilities and allowing for such a flexibility in usage of the tool, there are some steps that are semi-automatized or have to be done manually before conducting the SAT analysis. More detail about these processes can be read here[12].

As SAT's intended everyday usage does not generally include processing over two thousand applications, some limitations have been imposed, in the sense of evaluation each application's main architectural components and their interrelations. Therefore, in this research, metrics related to Component Balance and Component Independence have been excluded from the measurements, as it proved to be unfeasible to generate the correct component configurations for each application. The architectural components have to be predefined for each application before the static code analysis is applied, and due to the large variability in the mobile app development, it is not trivial to infer mobile app architecture based only on the structure and organization of the source code files. Therefore, due to time constraints for this project, it proved unfeasible to manually investigate each app's architectural components in order to be able to use them in static code analysis. Furthermore, the metrics could be misleading as the architecture of an app can easily change between snapshots, which would then require reconfiguring the component configurations for each app. In conclusion, SAT provides a practical, pragmatic and industry approved approach to measuring software product quality, and has provided this work with a means to evaluate maintainability issue evolution.

[1]https://www.ing.nl
[2]https://www.klm.com
[3]https://www.waternet.nl
[4]http://www.apotti.fi/

4.3.3 Maintainability Issue Density Identification

In this reseach, Maintainability issues are classified across different categories, corresponding to different sections of SIG Maintainability model. Additionally, in this research a notion of Code smells issue density is also used. Although this category does not fall directly under the maintainability issue subcategories presented in the model, it contains useful information about the quality of source code of applications and allows for a more fine-grained analysis of different source code bad practices occurring in applications. Therefore, in this research, there are seven different categories that have been used to investigate Maintainability issue evolutions. The definitions of each issue category are presented below. Furthermore, for each category, except Maintainability, a sample issue has been extracted from one of the repository source code files in the dataset. The source code listings can be found in the Appendix section 10.1.

- **Maintainability:** Aggregated issues of all other categories (except Code smells)
- **Unit Size:** Issues related to units that exhibit more LOC than recommended by the SIG Maintainability model and therefore pose a maintenance effort risk
- **Unit Complexity:** Similar to Unit Size, with respect to complexity of units. Complexity of units is measured with McCabe's cyclomatic complexity. Should an unit contain a high complexity, it will be harder to maintain it.
- **Unit Interfacing:** Similar to Unit Complexity, but interfacing revolves around the number of interface parameter declarations in an unit. The more parameters, the harder it is to perform maintenance changes to that unit.
- **Module Coupling:** Issues related to coupling between modules, whereas modules in this context are groupings of source code units.
- **Duplication:** Degree of duplicated/redundant code blocks in the application's source code. In this research a code block is considered redundant if it exhibits 6 or more identical source code lines.
- **Code smells:** An issue category revolving around specific source code bad practices or faults in code that could potentially cause failures in the production code. These issues are smaller with respect to their scope than their maintainability counterparts, and are usually traceable to a single line of code.

Once the issues have been identified, a unified and comparable measure of their amount is needed, in order to allow for objective comparisons between differently sized applications and between different issue categories. Therefore, for this research, a notion of *Issue Density* has been introduced. This approach closely follows the density definition presented in [30], and in this research the issue density is defined as

$$IssueDensity = \#ReportedIssues/NKLOC$$

where the number of reported issues is obtained for each application's snapshot and the $NKLOC$ is defined as the number of thousands of Lines of Code for each snapshot. For each of the aforementioned issue categories, appropriate densities have been calculated. Seeing as these density measures take into account the source code size of an application, measured in KLOC, two applications become comparable with respect to their amount of maintainability issues, regardless of their differences with respect to size and volume. Maintainability Issue Density entails the aggregated densities for maintainability subcategories, while Code smells Issue Density is calculated in isolation from the other issue categories. Therefore, when comparing two applications, the application with a higher issue density is of lower source code quality. Likewise, a lower issue density value translates to a better source code quality of an application. Apart from defining the proper objective density measures for issues, it is necessary to standardize the development lifetimes of all applications. Seeing as applications' development lifetime ranges from 8 to 693 weeks, it would be unfeasible to compare these applications on a same time scale by taking into account the data points in the development lifetime. Therefore, lifetime for each application has been standardized to represent the complete application development lifetime, and is defined as a range of values from 0–1. Weeks in an application's lifetime corresponding to data points are then standardized to match these values. For example, if an application has a lifetime of 10 weeks, the time will be standardized as a sequence of decimal values in the range of 0–1, therefore week 2 corresponds to a value of 0.2, week 6 to a value of 0.6, etc. Standardizing issue values and development lifetimes for all applications allows for a comparison between applications, and, more importantly, allows for the application of regression analysis techniques in order to examine evolution trends.

4.4 Trend Analysis with K-means Clustering on Regression Coefficients

This section provides an overview into the K-means clustering technique of all applications in the dataset. Rather than qualitatively analyzing the evolution of the density of maintainability issues for **123,261** snapshots in order to uncover underlying trends, this research will focus on evolution trend identification supported by the results obtained using regression models and clustering algorithms. The obtained clustered applications will be used further in the research to answer research questions. Clustering of all applications is performed in several steps: firstly, applications are normalized with regards to their variables, namely time and reported different granularity issue densities. Secondly, regression models of different polynomial orders are applied to each application in order to obtain regression coefficients for each application. Lastly, these regression coefficients are used as input features for the K-means algorithm.

4.4.1 Fitting regression models to applications

How does one approach identifying and extracting meaningful information from $123,261$ snapshots? With respect to the available time and resources for this thesis, inspecting each application individually proved to be unfeasible. Therefore, an automated approach is needed that alleviates the workload and provides guidance into the analysis of maintainability issue evolution trends.

Regression analysis has been widely used in estimating or predicting values based on the available data. In this research, regression analysis has been used to standardize the evolution trends of all applications, in order for them to be more comparable between each other. In its core, regression analysis serves a purpose of predicting a dependent variable Y based on one or multiple independent variables X_i. In the context of this research, dependent variables correspond to different issue densities, while the independent variables are represented by (standardized) application development lifetimes. Furthermore, regression analysis used in conjuction with clustering techniques is gaining popularity in the field of clustering functional data. For example, Tarpey[33] has used linear transformation of the functional data for 414 depressed subjects treated with Prozac for twelve weeks in combination with k-means clustering in order to extract representative curve shapes and mean cluster curves. In another work by Tarpey et al.[34], linear functional data has been used alongside k-means clustering in order to determine representative precipitation patterns.

In the context of this research, four different regression models have been applied to the different issue categories. These models are polynomial-level ordered functions, including Linear, Quadratic, Cubic and Quartic models. Applying a regression model to the data produces a fit that can be used to describe this data with a certain degree of correctness. An example of a linear regression fit produced for the data takes the form of

$$Y_i = \beta_0 + \beta_1 x_i$$

where Y_i is the independent variable, β_0 is the value of the intercept, and β_1 is the slope value of the fit. It is trivial to notice that the fit corresponds to a straight line. Intercept value determines the value of Y_i when $x_i = 0$. It is important to mention here that the values of β_0 and β_1 are called **Regression coefficients**.

4.4.2 Obtaining the regression coefficients

Once the regression models have been applied to the dataset, it is necessary to extract some values from these fitted models that help explain the underlying data. There are two possible choices that can guide researchers with this: either calculate a set of predicted values based on the regression fit formula, or obtain the regression fit coefficients that are used to represent the fit. Furthermore, numerous regression models fitted to the data could prove to be *overfitting* or *underfitting* the data. What this means is that these models do not represent the underlying data well and produce false predictions. Underfitting models will fail to explain significant variability in the data, while overfitted models will account for errors in the dataset and also produce false and misleading results. In order to avoid this, a researcher can investigate the goodnes of a regression fit either visually or with the help of regression metrics. Visual inspection of a goodness of fit comes in the terms of inspecting the *residuals vs fitted* plots alongside *QQ residual normality* plots. With these two visualization techniques, if a residuals vs fitted plot exhibits no strong pattern inbetween the data points, and if the QQ residual normality plot points to a normal distribution, the regression fit is considered to be good. However, in the context of this research, examining the goodness of fit visually for each regression model applied on each application would amount to carefully examining thousands of different plots, which proves to be unfeasible. Therefore, a more numerical approach has been adopted. An $R^2_{Adjusted}$[35] measure has been used to investigate or compare the goodness of fit between two models. Unlike normal R^2 measure, $R^2_{Adjusted}$ accounts for the amount of data points in the fitted regression model, providing a more robust value of inspecting the goodness of fit[29]. Formal definition of $R^2_{Adjusted}$ comes in the form of

$$R^2_{Adjusted} = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

where n is the number of data points in the sample and k is the number of independent variables in the model. $R^2_{Adjusted}$ is usually only slightly lower than standard R^2 value, but allows for more comparisons between models. In this research $R^2_{Adjusted}$ is used to compare regression models when ties are introduced later in the clustering section. Following from the presented information, below is a list of all the important regression values extracted for each regression model fitted to an application's data points:

Linear	Intercept, x, R^2, Adjusted R^2
Quadratic	Intercept, x, x^2, R^2, Adjusted R^2
Cubic	Intercept, x, x^2, x^3, R^2, Adjusted R^2
Quartic	Intercept, x, x^2, x^3, x^4, R^2, Adjusted R^2

With this information readily available, applications are now standardized with respect to their evolutions and density values, and are ready for k-means clustering process. The obtained regression coefficients describe the fitted regression model that explains the application's underlying data. The R^2 measures have been used to decide on the goodness of fit for models, in the case of ties with respect to choosing the representative model for clustering. The regression coefficients have been used as input features for the clustering of evolution trends. The Intercept coefficients describe the value of the dependent variable when all the other independent variables are 0, while the rest of the x^i coefficients help in defining the unique characteristics of each fitted model, for example slope, curvature or magnitude. As an example, two applications with similar values of coefficients from the Quadratic fitted models on Maintainability issue category have been extracted. Table 4.3 shows the values for the obtained coefficients, while Figure 4.6 shows the fitted regression models over the two apps. App lifetime

Table 4.3: Sample regression coefficients for Quadratic fitted model.

App	*Intercept*	x	x^2
Android-Port-Scanner	-4.631334	34.65279	-10.75717
MyCVApp	-4.563626	35.09066	-15.13787

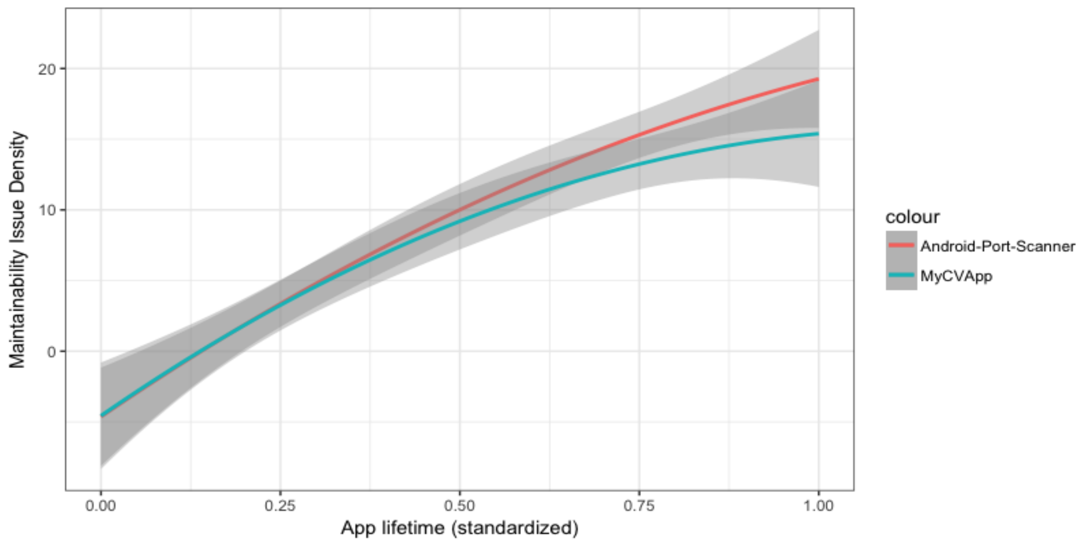


Figure 4.6: Example of fitted quadratic regression models on two apps with similar coefficients.

from the figure refers to the standardized time values for both apps, to allow for comparability between the models. What can be seen from the figure is that the coefficients following similar values construct similar regression curves that explain the underlying data. In this example, regression models for both apps follow similar evolution, with slight differences nearing the end of the sample lifetimes. Seeing as coefficients with similar values follow similar evolution, this serves as the basis for the clustering based on regression coefficients, which is what has been done for different issue categories and for different polynomial-order level regression models.

4.4.3 K-means clustering based on regression coefficients

In this section the clustering method for clustering applications based on their similarities in maintainability issue density evolution is presented. This method involves the use of k-means clustering algorithm[15] combined with the usage of regression coefficients calculated for each fitted model as input features for k-means. First, a short overview of the k-means algorithm is given, alongside a small algorithm explanation. Next, a more in-depth approach to the specifics of the modelled clustering approach for this research is presented. Finally, tools and techniques to support the clustering decisions, such as choosing the best number of clusters, are given at the end of this section.

One of the initial k-means algorithm versions is described in detail by John. A. Hartigan in the book *Clustering Algorithms*[15] in which the foundations for the algorithm have been laid out. Furthermore, a few years later, an improved version of the algorithm has been published that has been widely adopted and implemented in various statistical tools and programs. This research focuses on the improved version of the algorithm[16]. Furthermore, k-means clustering is a widely known method often used in research regarding clustering problems and often appears in relation to machine learning topics. There is existing research presenting improved versions of k-means algorithm in order to improve performance or increase capabilities [38], [23], [11], [20], applications of k-means in research results can be seen in [32], and k-means clustering of functional data has been investigated in [19], [33] and [34]. Regression clustering with the usage of k-means has also been explored in [31].

The aim of the k-means clustering is to partition n observations into k clusters such that the Within Cluster Sum of Squares (WSS) is minimized. More formally, the algorithm aims at minimizing the *objective function*, given as

$$J(V) = \sum_{i=1}^{c} \sum_{j=1}^{c_i} (||x_i - v_j||)^2$$

where,
$||(x_i - v_j)||$ is the Euclidean (L_2) distance between x_i and v_j,
c_i is the number of data points in i^{th} cluster,
c is the number of cluster centers. [28]

In the process of clustering, the k-means algorithm works in the following way:

1. Randomly assign initial cluster centroids
2. For each data point, compute the Euclidean distance to each of the cluster centroids
3. Assign the data point to the nearest cluster centroid

4. Once all data points have been assigned to their respective clusters, compute the new cluster centroids, positioned in the centers of identified clusters
5. Reiterate the process of assigning data points and calculating cluster centroids until convergence or until the user-defined number of iterations has been executed

It can be seen that k-means minimizes euclidean distance between points in clusters, and therefore groups data points based on their similarity in values. A visual example of k-means cluster assignment can be found in [24]. In this research, k-means clustering serves as a valuable tool in order to uncover maintainability issue density evolution trends. After computing the regression coefficients for each application, k-means clustering is applied to regression coefficients in order to cluster applications based on their similarities. K-means as a statistical clustering technique contains some assumptions and drawbacks. The first and biggest drawback is that the number of clusters k must be specified apriori the actual clustering. An explanation into the guide on choosing the optimal k for this research is given later in this section. Furthermore, the input features for k-means clustering need to be standardized in order for k-means to produce meaningful results. With the use of clustering on regression coefficients rather than clustering the raw data points, a few potential pitfalls have been avoided. With respect to the application's lifetime, numerous applications have differing lifetimes and therefore different amount of data points available. Computing regression fits for these application and obtaining regression coefficients produces standardized values that can be used for k-means. With the approach of clustering regression coefficients, the dimensionality of the dataset is reduced greatly, with the dimensions spanning from 2 for linear regression coefficients to 5 dimensions for quartic regression coefficients. As the regression coefficients are used to identify the function describing the regression fit applied to the underlying data, similar regression coefficients will explain similar regression predicted functions.

4.4.4 Choosing the optimal K number of clusters

In order to overcome the drawback of knowing the K number of cluster before the clustering is performed on the dataset, researchers have a number of tools and methods available for guidance. Researchers can look for an elbow point in a sum of squared error scree plot, rely on Silhouette plots or R packages such as `pamk` or `clusplot`. An example of an elbow point in a Sum of Squared Errors (SSE) scree plot is given in Figure 4.7.

In this research, R package `NbClust` has been used to guide the selection of the optimal K number of clusters. NbClust is a package that provides 26 indexes for determining the optimal number of clusters in a data set and offers the best clustering scheme to the researcher[27].

The NbClust package runs the k-means algorithm multiple times with a differing number of clusters and presents the researchers with the best option based on criteria ranked by the 26 criteria. An example of a NbClust output is given in Figure 4.8.

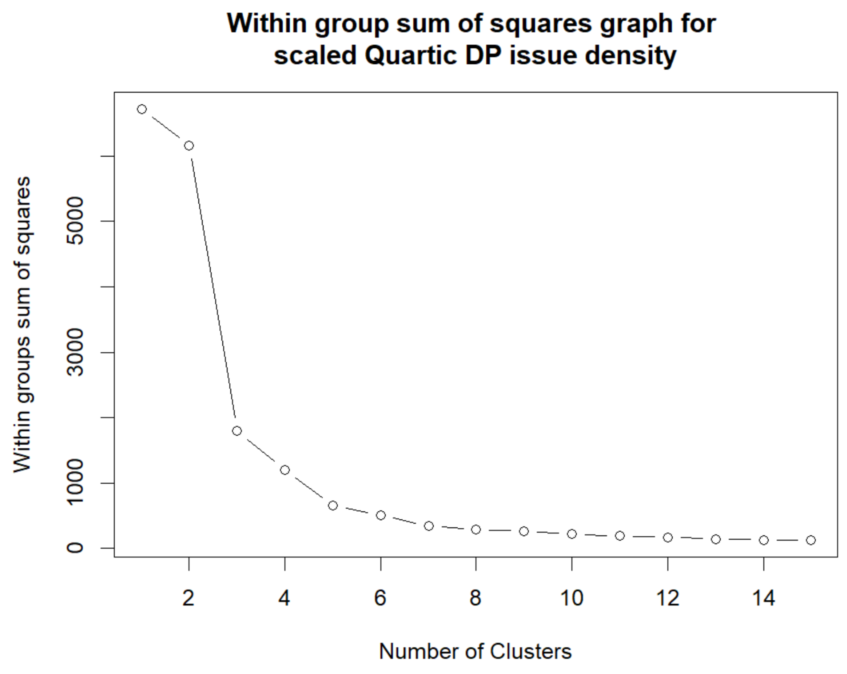


Figure 4.7: Example of an elbow point in a SSE scree plot, suggesting 3 as the best number of clusters.

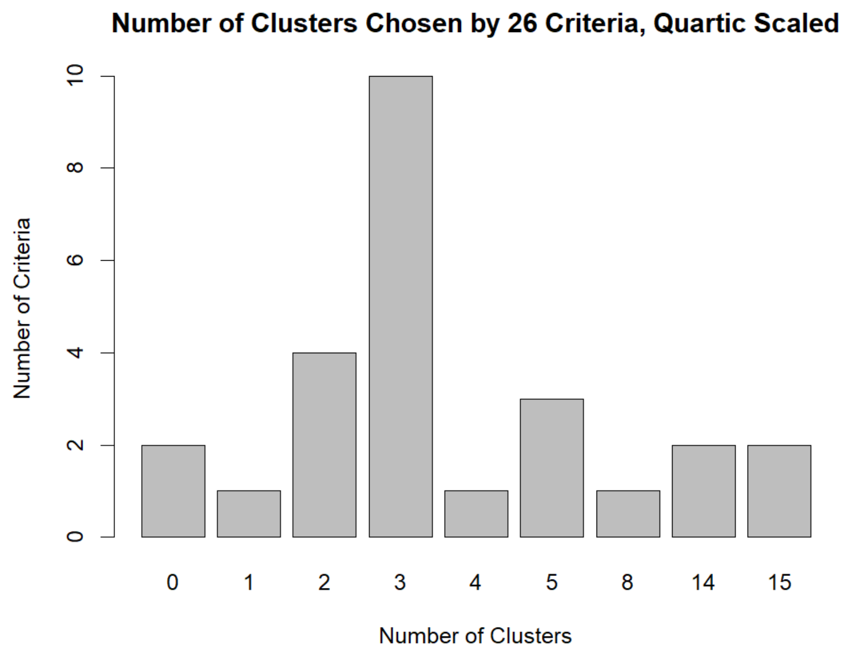


Figure 4.8: Overview of the NbClust result, providing the best number of clusters based on 26 criteria.

5

Results

In this section the results obtained in this research are presented. The results include summarized information about the conducted and performed statistical tests presented in the previous section. For each statistical test a significance level of 95% (p-value < 0.05) is assumed. Furthermore, the results are reported according to the identified research questions.

First, a high-level overview of the general source code quality of applications is presented with Table 5.1, expressed through SIG Maintainability star ratings. As a reminder, SIG Maintainability Model provides a five star rating. Expressing that in numbers provides a range of values between 0.5 to 5.5, corresponding to a star rating of 1 to 5 stars, where a lower amount of stars means a higher risk for the system and the increased maintenance efforts. Modularity has the lowest overall ratings, however this can be due to this research excluding metrics for Component Balance and Component

Table 5.1: SIG Maintainability Model Snapshot Ratings summary, ordered by median

	Min	1st Quartile	Median	Mean	3rd Quartile	Max
Testability	2.103	3.740	4.068	4.168	4.575	5.500
Modifiability	0.7979	2.788	3.303	3.430	4.030	5.500
Maintainability	1.527	2.732	3.054	3.126	3.465	4.750
Analyzability	1.306	2.760	3.052	3.059	3.360	4.250
Reusability	0.500	2.394	2.904	2.951	3.424	5.500
Modularity	0.500	1.439	2.018	2.025	2.918	3.000

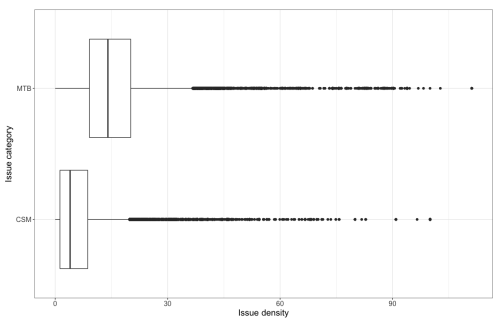


Figure 5.1: Maintainability and Code smells issue density distributions.

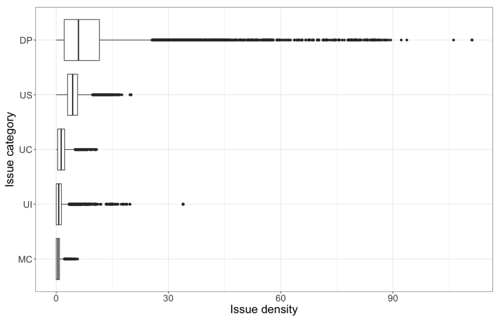


Figure 5.2: Maintainability subcategories issue density distribution, ordered by mean issue density.

Independence. In the SIG Maintainability model, higher component balance and independence positively affects the Modularity rating, while Module Coupling negatively affects the Modularity rating. Therefore it is not trivial to reason whether this really is the least maintainable subcharacteristic or not. However, the next lowest overall score (judging by mean values) falls under the Reusability category. This category is negatively impacted by Unit Size and Unit Interfacing. This result suggests that the unit-related metrics of Android applications suffer penalties with respect to maintainability and pose a risk for the maintenance efforts. Testability on the other hand exhibits highest overall ratings, which suggests that the Volume of applications is rather small and that the units exhibit lower complexity than the benchmark standard.

Figure 5.1 presents an overview into the issue density distribution across two main categories, Maintainability (MTB) and Code smells (CSM). It can be seen that the Maintainability issue density overall has a higher mean value than Code smells, suggesting that there are more issues related to maintainability contained in the snapshots than Code smells. Furthermore, comparing the medians of the two presented boxplots, Maintainability issue category exhibits a median of 14.06, while Code smells issue category exhibits a median of 3.97. This result suggests that there are 3.5 times more maintainability issues on average contained in mobile apps than code smells issues. As Maintainability issue category is composed of mutliple subcategories, the issue density distribution for these subcategories is presented in Figure 5.2. In this overview, Duplication (DP) issue category takes the lead, having both the highest median and mean issue densities compared to other maintainability issue subcategories, suggesting that mobile apps contain numerous redundant source code lines. Source code duplication in mobile apps results in higher LOC counts for unit sizes, which can be seen in the figure, as Unit Size (US) issue category exhibits the next highest mean issue density values. Middle rank of this ordered boxplot goes to Unit Complexity (UC) issue category, followed by Unit Interfacing (UI). It can be seen that Unit Interfacing issue category contains multiple outlier values surpassing the maximum values of Unit Size, suggesting that there is an unusual increase in UI issue density contained somewhere in the source code (more on this specific case later in 5.2). Module Coupling (MC) issue category exhibits overall lowest issue density.

5.1 RQ1: How does the density of maintainability issues of Android applications evolve over time?

Results of applying the `NbClust` technique for deciding the optimal number of clusters to each regression model and to each issue category (Maintainability, Code smells and each of the Maintainability issue subcategories) are reported in Table 5.2. Choosing the optimal number of clusters has been based on the values of the proposed Criteria for selection. The higher the value, the better fitting the clustering is. However, it can be noticed that there are some ties in certain issue categories with respect to the Criteria values (e.g. Maintainability). The ties are resolved by always choosing the higher k and the higher-order polynomial regression model. First, it can be noticed from the table that the proposed optimal k number of clusters is either 2 or 3. More specifically, $k = 3$ has been proposed 21 times in total, while $k = 2$ has been proposed 12 times in total. This result suggests that overall the $k = 3$ number of clusters is better fitting, and is therefore selected as the optimal number to resolve ties. Furthermore, higher-order polynomial regression models in this research exhibit a higher mean $R^2_{Adjusted}$ measure, which implies that these models explain more variability of the underlying data. Therefore, in the case of ties with respect to both proposed k number of clusters and satisfied Criteria, the higher-order model is chosen to be representative. In the case of Maintainability (MTB), it can be noticed that there are three ties with respect to the satisfied Criteria values, resulting in ties between Linear, Quadratic and Cubic fitted regression models. In this scenario, the ties are resolved using the $R^2_{Adjusted}$ measure. As the mean $R^2_{Adjusted}$ measure for Cubic fitted Maintainability models is highest (0.56), compared

Table 5.2: The proposed optimal k number of clusters, as given by criteria from `NbClust` algorithm. Values in bold represent the chosen k number of clusters for each issue category.

Category	k_{linear}	Criteria	$k_{quadratic}$	Criteria	k_{cubic}	Criteria	$k_{quartic}$	Criteria
MTB	3	8	3	8	**2**	8	2	7
US	2,3	6	2	9	**3**	11	2,3	8
UC	3	8	3	8	3	9	**3**	15
UI	2	9	3	10	**3**	13	3	11
MC	3	10	2,3	8	**2**	10	2	9
DP	**3**	9	3	7	2,3	8	2,3	7
CSM	2	8	3	6	3	14	**3**	14

to 0.46 for the Quadratic fitted models and 0.29 for Linear fitted models, Cubic fitted model has been chosen as a representative for Maintainability issue category. Similarly, Module Coupling issue category (MC) exhibits another tie with respect to the satisfied Criteria, in the case of Linear and Cubic fitted regression models. Following the same principle, Cubic fitted models have been chosen as representative. Following these rationales, it is trivial to identify the representative models for the rest of the issue categories. Linear fitted models with $k = 3$ have been chosen as representative for Module Coupling (MC) and Duplication (DP) issue categories, Quadratic fitted model with $k = 3$ has been chosen for Maintainability issue category, Cubic fitted models with $k = 3$ have been chosen both for Unit Size (US) and for Unit Interfacing (UI) and Quartic fitted models with $k = 3$ have been chosen for the remaining issue categories, namely Unit Complexity (UC) and Code smells (CSM) issue category. Presented table serves as a guidance into the clustering of evolution trends, where the K-means clustering has been applied to regression coefficients matching the selected representative model.

Following these results k-means clustering was performed on the representative models for each issue category separately. Table 5.3 describes the identified clusters for all issue categories and the corresponding amount of clustered applications. The total number of applications differs for each issue category due to limitations imposed by regression analysis, as was mentioned in 4. Furthermore, every issue category seems to exhibit a cluster that contains a significant amount of applications compared to the other clusters for the same category. The largest cluster in Code smells issue category contains more than 86% of all clustered applications, while the smallest one contains

Table 5.3: App distribution on identified clusters.

Category	# Clusters	Best Model	Total Apps	Apps c1	Apps c2	Apps c3
MTB	2	Cubic	1,504	221	1,283	-
US	3	Cubic	1,469	291	133	1,045
UC	3	Quartic	1,189	98	214	877
UI	3	Cubic	1,022	44	853	125
MC	2	Cubic	946	281	665	-
DP	3	Linear	1,336	144	1,108	84
CSM	3	Quartic	1,303	1,131	135	37

only 2.8%. Distribution of applications through identified clusters occurs more evenly in Unit Size issue category, where the largest cluster contains 71% of all clustered applications, the middle cluster contains almost 20%, and the smallest cluster 9% of all clustered applications. As the number of applications belonging to each identified cluster varies greatly, the *weight* of each cluster should be taken into account while drawing conclusions from the clusters. While comparing metrics exhibited in different clusters, the amount of apps contained within each observed cluster gives insight into the differences between these metrics. The consequence here is that differently sized clusters do not carry the same weight, in the sense that the trends cannot be compared equally, if one of them exhibits only 5% of apps, while the other contains the remaining 95%. Therefore, number of applications contained within each cluster has been presented in order to allow for objective reasoning about the metrics contained within them. In the next section a more detailed overview into the evolution trends of different issue categories is presented.

5.1.1 Maintainability Issue Density Evolution Trends

In this section identified evolution trends are presented and explained. With respect to maintainability issue density evolution, two different clusters have been identified, each exhibiting its own characteristics. Figure 5.3 presents the identified evolution trends for both clusters. In the figure, Relative Time represents the standardized app development lifetime for all applications. For more details, see Section 4.3.3.

In Maintainability issue category, two distinct evolution trends have been identified.

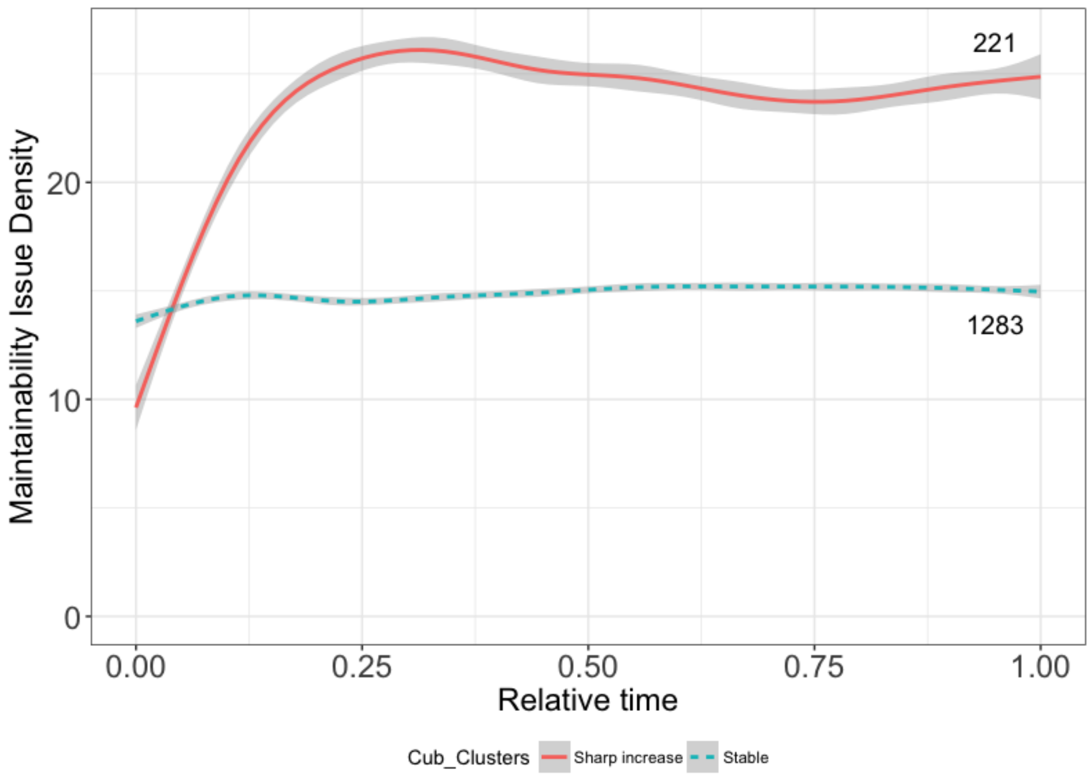


Figure 5.3: Maintainability Issue density evolution trends.

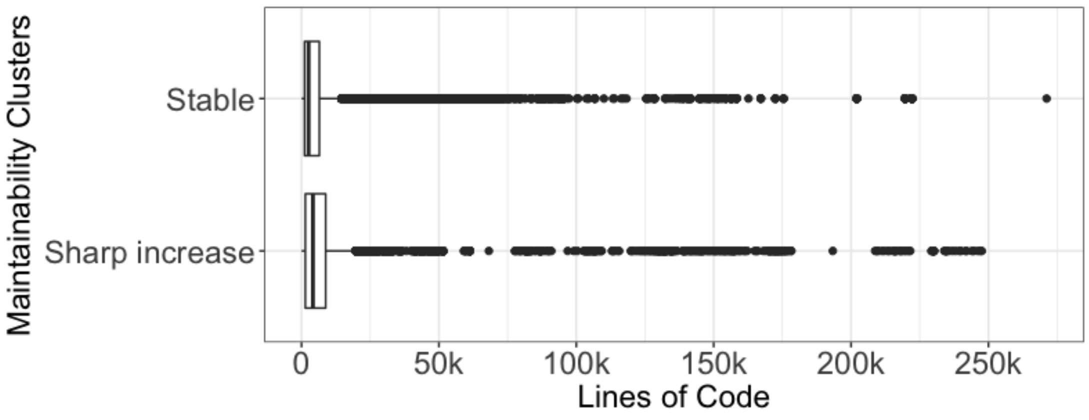


Figure 5.4: Distribution of LOC across maintainability evolution clusters.

In the cluster containing 1,283 mobile apps, there is a very slight increase at the beginning of the evolution, followed by a period of stability. Apart from exhibiting rather stable evolution, the mean density for all apps in the stable cluster is 14.88. Furthermore, the overall maintainability issue density seems to be rather low, with more than 80% of snapshots having a issue density of 20.21 or less. In the second cluster, containing 221 mobile apps, most noticeable is the *Sharp increase* evolution trend, followed by gradual decrease over time, and finally near the end of the evolution a slight increase can be seen again. It can be seen that the overall issue density values for this cluster are significantly higher than the stable cluster, where the mean value for Sharp increase cluster is 23.53, while the median is 22.14. For comparison, only 45% of snapshots have an issue density of 20.21 or less, compared to the 80% in the Stable cluster. Taking into account that the Sharp increase cluster contains 5.8 times less mobile apps than the Stable cluster, these numbers suggest that the mobile apps contained within Sharp increase cluster generally contain more maintainability issues. Furthermore, in the Sharp increase cluster, it can be noticed that Sharp increase is not followed by a Sharp decrease, therefore suggesting that the swift introduction of maintainability issues near the beginning of the evolution is not trivially reduced over time.

Figure 5.4 presents the distribution of LOC across the identified clusters.

What can be immediately noticed is that the second cluster, exhibiting a sharp increase in density, contains more LOC per snapshot than its Stable counterpart. Furthermore, the stable cluster has a mean LOC per snapshot of 6,567.5, while the sharp increase cluster exhibits a mean LOC per snapshot of 10,059.02. As the sharp increase cluster contains 5.8 times less mobile apps, this result is slightly surprising. However, this result also suggests that the apps in the sharp increase cluster are generally less maintainable than the apps in the stable cluster.

Each of the identified trends for the evolution of Maintainability issue density carries some implications for developers, project managers and researchers of mobile apps. It can be seen that the maintenance efforts for a mobile app that starts with a higher issue density remain substantial across the evolution of the app development, where it can be noticed that the resolution of maintainability issues is rather low. On the other hand, introduction of new maintainability issues can occur at a high magnitude, likely due to the introduction of new app functionalities, as the sharp increase trend is exhibited at early stages of app development. Therefore, while adding new app functionalities, developers should be cautious with respect to the amount of introduced duplication and the structure of source code units. Duplicated and complex code should be reduced with planned refactorings after the introduction of new functionalities. As the sharp increase trend is not followed by an equally sharp decrease trend, introducing new unmaintainable source code could require more maintenance efforts at later stages in the development. For researchers investigating Maintainability of mobile apps, these clusters can provide guidance into the different evolution trends and the composition of issues over time contained within different mobile apps. Furthermore, careful attention should be paid to the app characteristics as few apps can vastly affect the results of the research.

5.1.2 Unit Size and Unit Complexity Issue Evolution trends

Moving into Maintainability issue subcategories, first two categories are Unit Size and Unit Complexity. Unit Size issue density evolution has been clustered with Cubic fitted models, while Unit Complexity issue density evolution was clustered with Quartic fitted models. Both categories have three identified clusters. Examining each subcategory evolution trends in isolation allows for a more detailed overview into the evolution trends of specific components that compose Maintainability. Issue density evolution trends for Unit Size are presented in 5.5, while Figure 5.6 presents Unit Complexity issue density evolution trends. Regarding Unit Size, it can be seen that there are three distinct evolution trends.

First Unit Size cluster, containing 291 mobile apps (19.8%), exhibits a sharp increase in density early in the app development, followed by another, milder increase, before it starts slowly decreasing and lastly reaches a period of stability. Second cluster contains 133 mobile apps and exhibits an inverse situation. It starts of with a higher Unit Size issue density that sharply decreases, followed by a slight increase over time and finishing with a slight decrease. Lastly, the largest cluster, containing the remaining 71% apps, exhibits a slight increase at the beginning of the development, followed by a period of stability. As can be seen, even though sizes of units in mobile apps evolve differently, all three clusters converge to a similar density by the end of their observed evolution. This could occur due to planned refactorings in code that would eventually normalize sizes of units in the application's. What is interesting to note is that all three clusters do not converge towards having an issue density of 0, but rather towards an issue density of 4. This could suggest that the size of units in Android applications is in general above the recommended Unit Size coming from the SIG Maintainability Model[21].

There are numerous differences in the early stages of mobile apps development. Some applications start out with larger units that get reduced over time, while others start out with lower unit size that gradually exhibits growth. Project managers should understand that Unit Size acts in a more unpredictable manner, with oscillations and variations in issue density. Therefore, maintenance activities should be considered regularly to prevent unit size from constantly growing.

Moving onto Unit Complexity, issue density evolves similarly to Unit Size, hinting at the relationship between the two categories. The smallest cluster, containing 98 mobile apps, exhibits a sharp decrease trend followed by a slight increase. During the middle of the app development, issue density starts decreasing again, followed by a slight increase before the end of the observed lifetime. The most applications are contained in the cluster exhibiting a slight constant increase. Furthermore, this cluster has the lowest mean issue density value compared to other clusters. Furthermore, the scale of the y-axis suggests that Unit Complexity issue density is in general slightly lower than Unit Size.

It can be noticed that Unit Complexity exhibits a lower overall issue density magnitude, however it is more susceptible to growth than Unit Size. Even the cluster showing the most stability exhibits a slight constant growth in complexity of units. Furthermore,

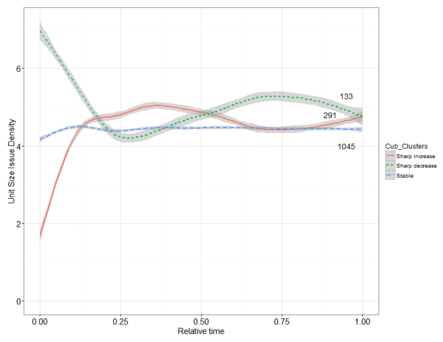


Figure 5.5: Unit Size Issue density evolution trends.

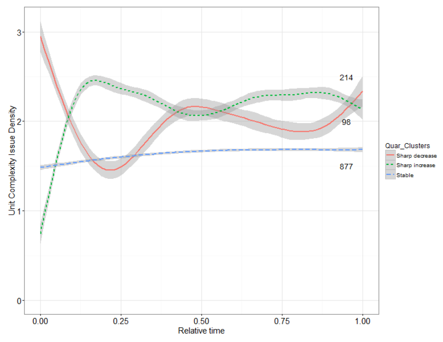


Figure 5.6: Unit Complexity Issue density evolution trends.

the resolution of complex units almost always results in a subsequent increase, suggesting swift introduction of Unit Complexity issues. Project managers should take this into account while planning development, preventing complexity of the app's codebase from quickly increasing. Mobile app developers should carefully observe the evolution of the complexity of units, especially during the beginning of the app's development, in order to maintain a manageable Unit Complexity issue density.

5.1.3 Unit Interfacing and Module Coupling Evolution trends

Unit Interfacing issue density relates to the number of incoming parameters to each source code unit, and exhibits three different evolution trends. The smallest cluster, containing only 44 apps, exhibits a sharp decrease trend at the beginning of the development, followed by a slight increase trend before reaching a period of stability. Second cluster exhibits a rather stable trend throughout the evolution, containing 853 apps (83.4%), and overall maintaining the lowest issue density. Final cluster, containing 125 apps, exhibits a sharp increase trend followed by a gradual decrease in density over time. Looking at the Figure 5.7, showing the distribution of LOC across different Unit Interfacing clusters, it can be seen that the cluster exhibiting the highest amount of apps also contains the most LOC out of the three clusters. This results differs from Maintainability issue category, where a cluster with a smaller amount of apps contained more LOC than the largest cluster.

With the cluster exhibiting stable evolution containing the most apps and the most LOC, certain assumptions can be made regarding Unit Interfacing issue density. It appears that Unit Interfacing is the most manageable unit-related metric, showing no

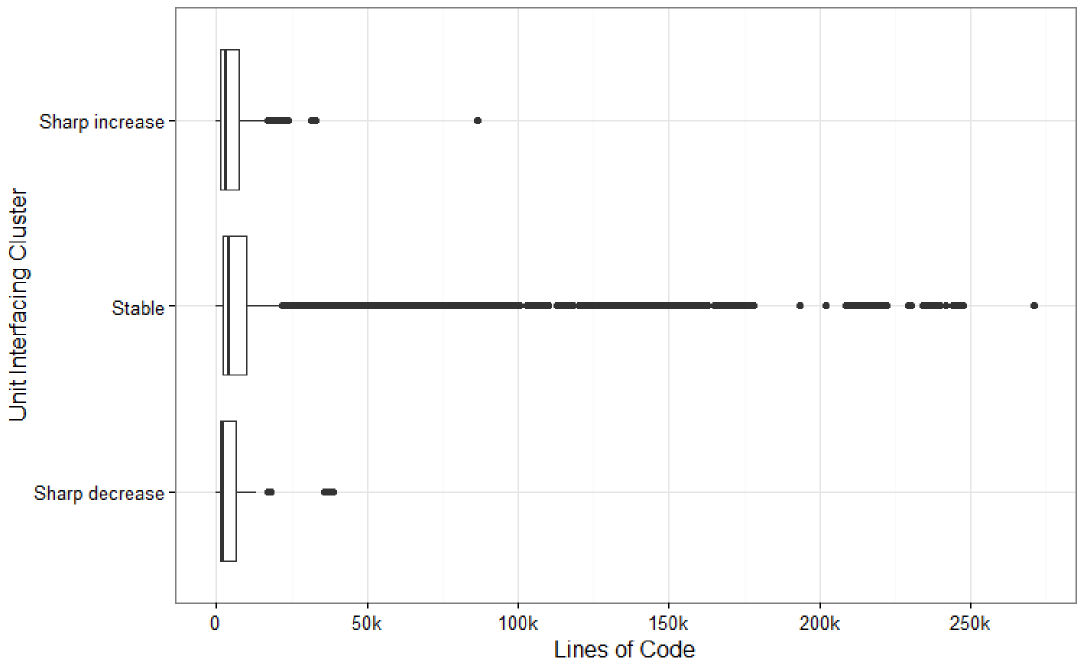


Figure 5.7: Distribution of LOC across Unit Interfacing evolution clusters.

reaction to the increased amount of LOC. Therefore, it seems that the developers of these apps are well acquainted with managing the number of incoming parameters in their units. Even apps that tend to start with a higher Unit Interfacing issue density tend to resolve these issue rather quickly, slowing down the issue density growth.

Module Coupling issue density relates to the architectural structures of each app and reports on incoming dependencies between app modules. In this category, two evolution trends have been identified. It can be immediately noticed that the two evolution trends closely follow the trends identified in the overall Maintainability issue category. The larger cluster, containing 665 mobile apps, exhibits both a slight increase and a slight decrease at the beginning of its evolution before reaching a period of stability. The other cluster, exhibiting 281 mobile apps, starts out with a lower module coupling issue density, but shortly afterwards exhibits a sharp increase trend. Similarly to overall maintainability evolution trends, the sharp increase is followed by a gradual decrease and lastly finishes with a slight increase in evolution. Looking at the y-axis of the evolution trends, it can be noticed that the density values are extremely low. Taking into account that Module Coupling issues are mainly related to architectural

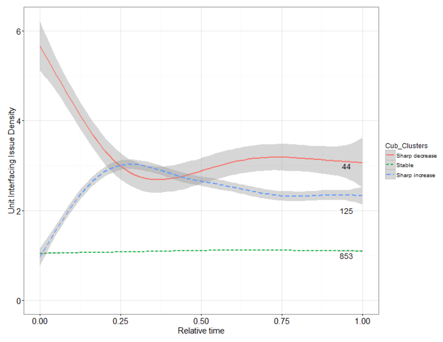


Figure 5.8: Unit Interfacing Issue density evolution trends.

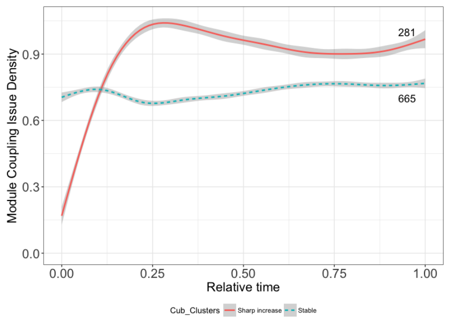


Figure 5.9: Module Coupling Issue density evolution trends.

source code metrics, such as Component Balance and Component Independence, which have been excluded from this research, there is a lack of strong evidence to support any claims made about the specifics of these clusters. However, developers and project managers can note that even with the exclusion of these metrics both evolution trends still exhibit overall growth, therefore hinting that the architectural coupling between modules could pose a maintenance risk at later stages in the app's development.

5.1.4 Duplication Evolution Trends

Before investigating the fine-grained Code smells issue density evolution trends, another important Maintainability issue category revolves around duplicated code. In this category, three clusters have been identified. Similarly to Module Coupling, the largest cluster contains the most apps (82.9%) and exhibits a slight increase in density at the beggining of its evolution, followed by a period of stability over time. The second largest cluster, containing 144 apps, also exhibits a slight increase in the evolution trend, this time followed by gradual decrease. The last trend, containing 84 apps (only 6.2%) exhibits a constant growth trend over time, where the first half of evolution exhibits a steeper increase, followed by a milder slope. Figure 5.10 presents the identified Duplication issue density evolution trends.

What can immediately be noticed from looking at the issue density magnitude is that Duplication appears to exhibit the highest issue density values compared to other Maintainability issue categories, suggesting that this is the most vulnerable issue category. Therefore, developers and project managers should pay special attention to the amount of duplicated code contained within their mobile apps. Furthermore, it can be

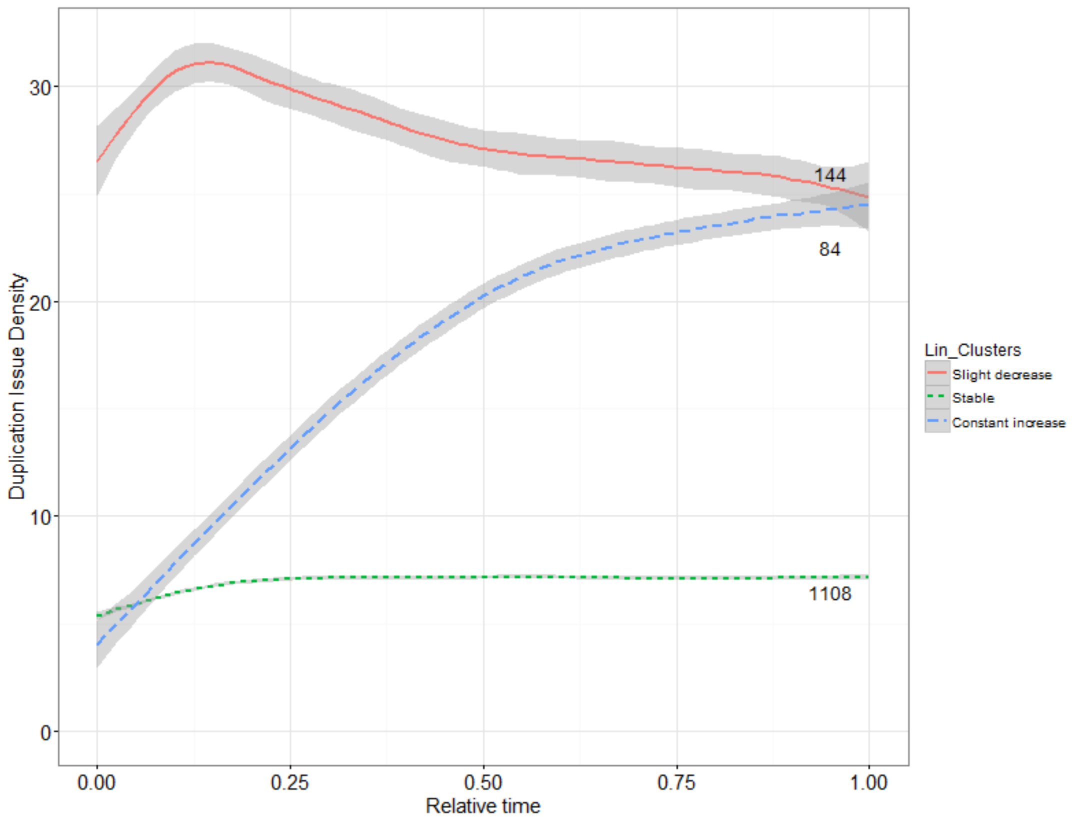


Figure 5.10: Duplication Issue density evolution trends.

noticed that the introduced duplicated issues tend to be removed rather slowly, compared to the rate of introduction of new Duplication issues. Looking at mean density values for each identified trend, the cluster exhibiting the most stability contains 7.17 duplication issues per 1,000 LOC, followed by the constant increase cluster with a mean issue density value of 17.83 and finally, the slight increase-gradual decrease cluster exhibits a mean issue density of 27.46. In order for project managers, developers and researchers to understand these numbers, it is worth mentioning the definition of Duplication from the SIG Maintainability Model again. For a code block to be considered duplicated, at least 6 LOC need to be identical. Therefore, having a issue density of 7.17 Duplication issues per 1,000 LOC means that in those 1,000 LOC at least 4.3% are redundant. The situation is worse for other clusters, where the cluster with the highest mean issue density contains at least 16.5% duplicated LOC. Therefore, a great deal of attention should be paid to the growing redundancies of an app's codebase, as it can severely impact its maintainability and therefore increase maintenance efforts.

5.1.5 Code smells Issue Evolution Trends

In this section an overview of the Code smells issue category evolution trends is presented. Code smells issue category differs from the rest of the maintainability categories as it provides a more fine-grained overview into specific development errors or faults. Furthermore, while Maintainability issues are usually traceable to a few LOC or a complete unit, Code smells issues can often be traced to a single LOC within the source code of an application. Therefore, meaningful information can be extracted by examining Code smells issue category in more detail, and in isolation from the rest of the Maintainability issue categories. Figure 5.11 shows the identified Code smells evolution trends. With respect to issue density evolution of code smells, three different clusters have been identified. The largest cluster contains 1,131 apps and exhibits a rather stable, linear evolution trend over time, with no noticeable oscillations with respect to issue density. The middle sized cluster, containing 135 apps, exhibits a slight increase in density at the beginning of the development, followed by a gradual decrease over time, before reaching a period of stability. The last cluster is the smallest one, containing only 37 apps (2.8%), but exhibits the most deviations with respect to issue density. Starting density for these apps appears to be quite high compared to other clusters, followed by

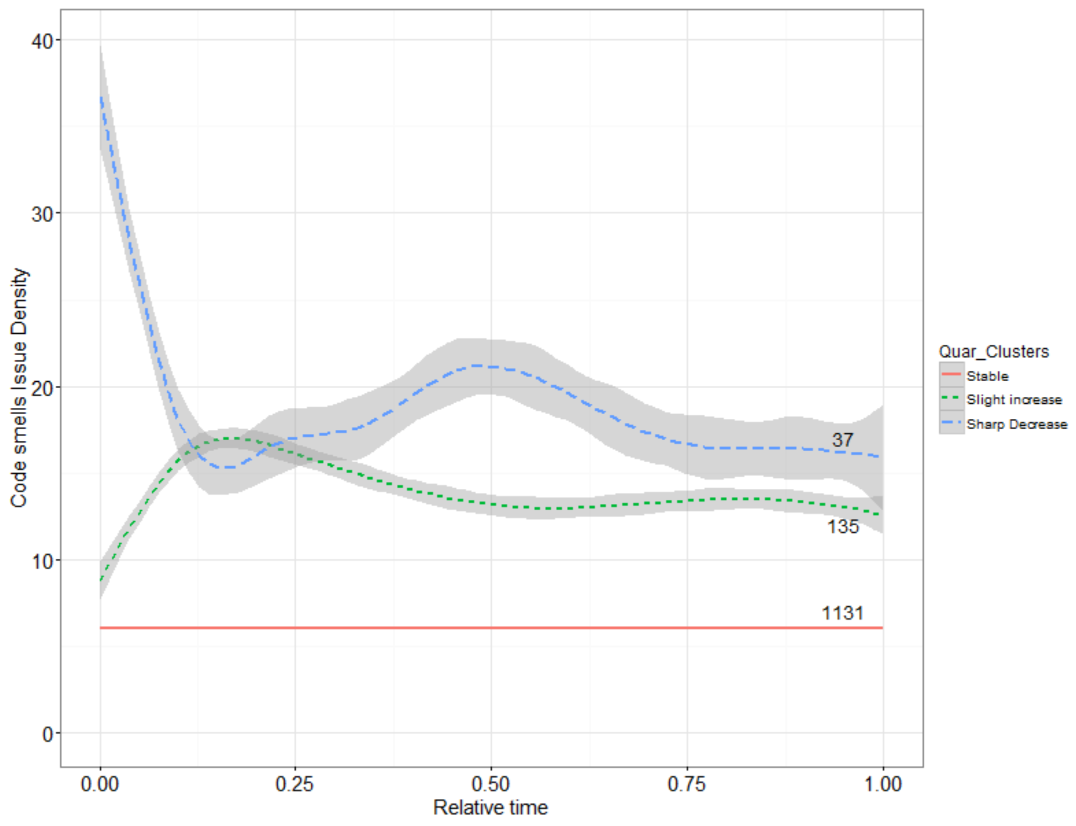


Figure 5.11: Code smells Issue density evolution trends

a sharp decrease in issue density. The sharp decrease is followed by a period of gradual increase, followed again by a period of slight decrease.

In order to draw some conclusions from the presented evolution trends, it is important to mention that developers should strive for the Code smells issue density to be 0, in order to reduce future maintenance efforts caused by buggy code. Furthermore, project managers should pay special attention to this issue category as some of these issues can produce production environment errors and therefore impact the mobile app functionalities. The resulting evolution trends hint at the importance of resolution of Code smell issues, as there is a general tendency of decreasing the issue density over time. Out of 107,548 snapshots analyzed in the Code smells issue category, a total of 7,462 (6.9%) exhibits a Code smell issue density of 0, which is the ideal value for this issue category and shows signs of clean code practice.

5.2 RQ2: How are maintainability issues distributed across different Android components?

With this question an investigation into the relationship and composition between Android-specific components and different issue categories has been performed. Out of 668,530 identified Android components in this dataset, over 77% of components are Activities. Services and BroadcastReceivers constitute similar portions of the dataset, at 9% and 11% respectively. Lastly, there is 12,554 identified ContentProviders (1%). Throughout the Android components a total of $3.9M$ issues has been identified. Figure 5.12 shows the distribution of identified maintainability issues across different Android components. It can be seen from the figure that Activities are predominant with respect to the amount of maintainability issues, with over $3.2M$ issues being contained inside Activities. Considering the ratio of issues found in Activities compared to all identified issues, Activities contain 82.6% issues. Services contain 12.7% of all identified issues, BroadcastReceivers 3% and finally, ContentProviders contain 1.7% of issues. From this data it can be concluded that BroadcastReceivers exhibit the lowest average amount of maintainability issues. Furthermore, looking at the average number of maintainability issues per Android component, Services exhibit the highest amount, 6.72 issues per one Service component, while BroadcastReceivers exhibit only 1.8 issues per component. This result suggests that Services are the least maintainable Android component in our study.

Now that some general issue distribution across Android components has been presented, a deeper insight into the distributions of different issue categories across components can be given. Relative distribution of different maintainability issue categories is shown in Figure 5.13. There are numerous insights that can be extracted from this figure. First, all components exhibit a large amount of maintainability issues related to Duplication. More than half of all issues contained in Activities and over a third of all issues contained in Services belong to this issue category. Activities and Services, on the other hand, contain the least amount of issues related to Unit Size (19%), followed by BroadcastReceivers and ContentProviders. Out of all the unit related issue categories, Unit Size is overall predominant with respect to its distribution across components, followed by Unit Complexity and Unit Interfacing issues. Furthermore, it appears that ContentProviders contain the largest units out of the four components.

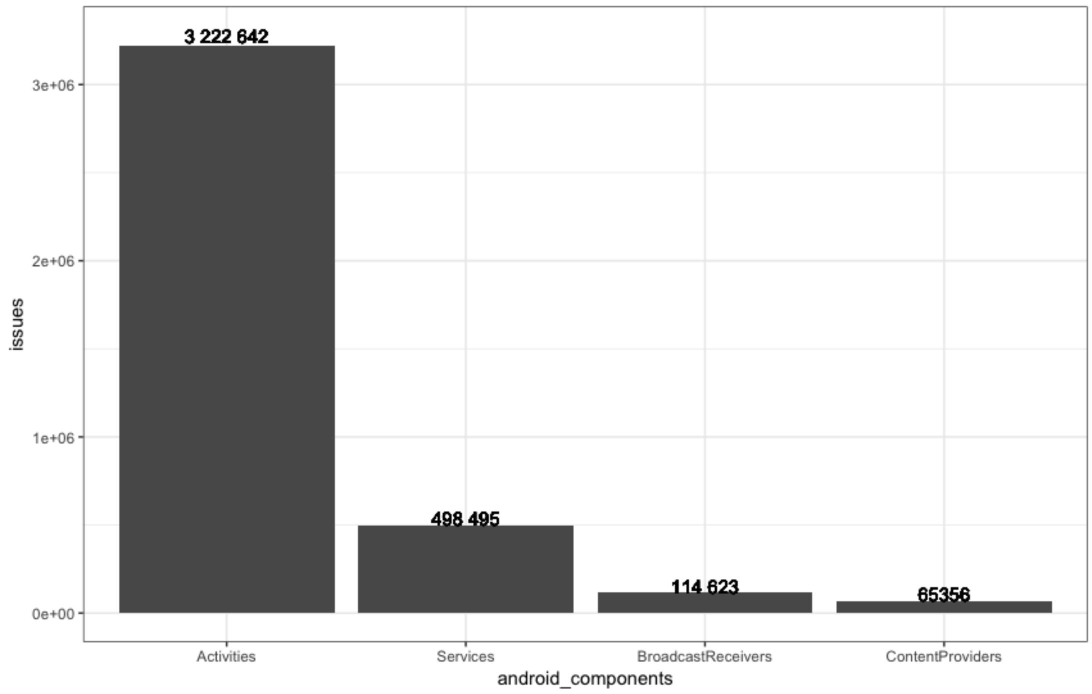


Figure 5.12: Distribution of issue categories across Android components.

Comparing Services and BroadcastReceivers, one can see that their distributions are largely similar, with Services exhibits slightly more Duplicated issues than BroadcastReceivers. In both of these components, Code smells issue category takes up more than a third of all the identified issues. Furthermore, issues related to Code smells issue category compose over a quarter of all identified issues (26.25%), and are the largest overall issue category after Duplication. As Code smells issue category revolves around bad and potentially dangerous coding practices, such as modifying control variables inside a loop, Services and BroadcastReceivers could potentially cause production environment execution issues and bugs. Looking at ContentProviders, it can immediately be noticed that a significantly larger portion of issues belongs to Unit Interfacing issue category, compared to the rest of the components. Almost a fifth of all the issues in ContentProviders are related to this category. As ContentProviders can provide an application with possibilitiies to manage access and sharing of its own data or data stored by other applications, it is expected there will be numerous interfaces involved in the development of ContentProviders. However, it seems that these interfaces contain large

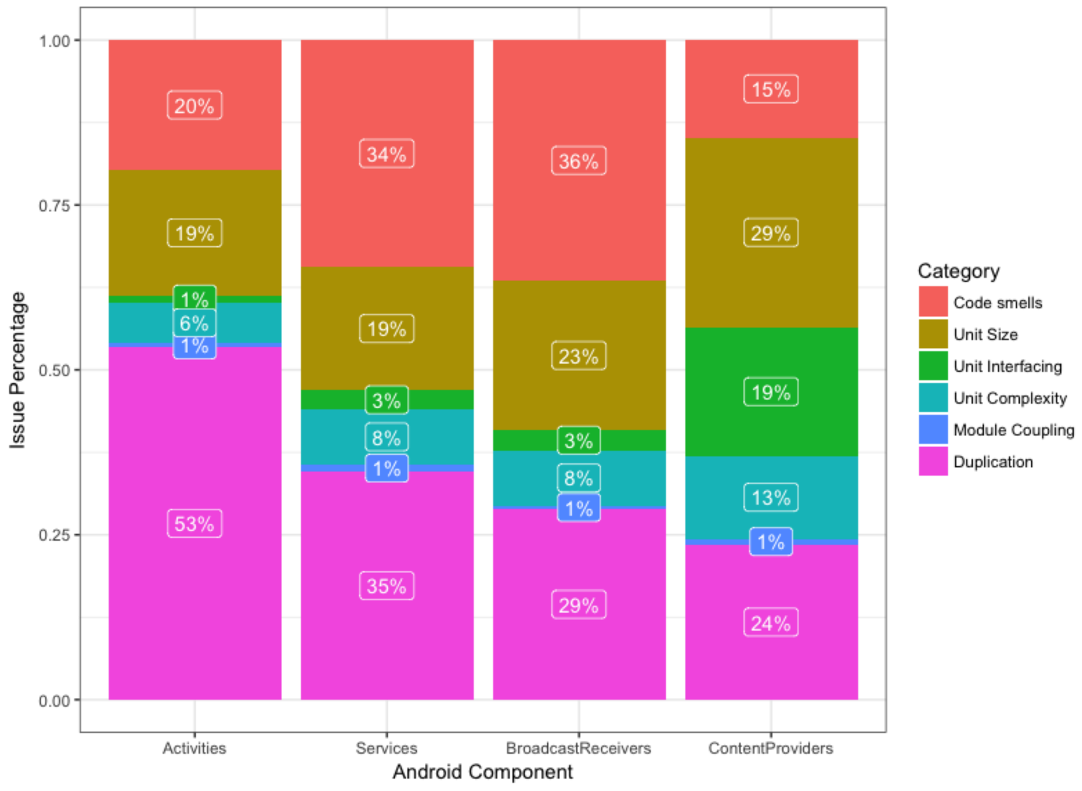


Figure 5.13: Distribution of maintainability issue categories across Android components.

amounts of input parameters, which could hinder maintenance efforts.

Regarding the complexity of development of ContentProviders, the results show that over 60% of all identified ContentProvider issues are related to unit-related issue categories, namely Unit Size, Unit Complexity and Unit Interfacing. As this percentage is almost twice as high compared to other Android components, it can be concluded that ContentProviders require the most attention from developers during development, as it can be seen that maintainability can suffer easily just from the unit-related issue categories. For example, the total amount of unit-related issues in ContentProviders is 39,748, compared to 39,087 issues identified in BroadcastReceivers. However, there are 3.24 times more BroadcastReceiver components, suggesting that the unit-related issue categories in ContentProviders exhibit more unit-related issues on average, therefore suggesting these components are more complex to develop.

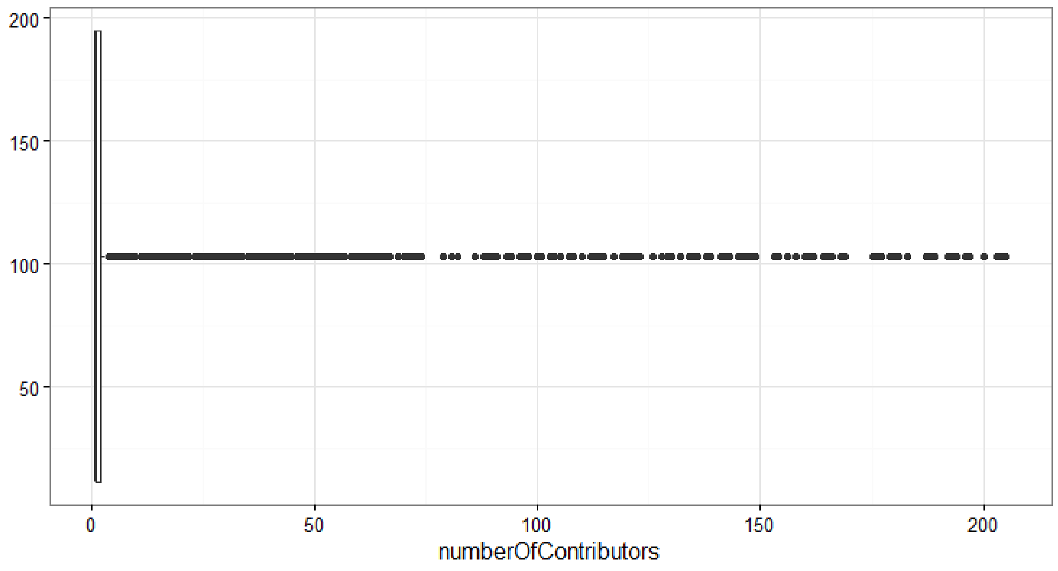


Figure 5.14: Boxplot of the number of contributors in snapshots.

5.3 RQ3: How does the size of the development team (number of contributors) impact the maintainability issue density of Android applications over time?

Moving from source code based metrics, it is interesting to investigate whether there exists a correlation of the number of contributors on maintainability issue density over time. It is worth mentioning that this research considers the number of contributors as the number of commiters to an application's repository source code directories containing the actual application source code, not accounting for commiters contributing to other sections of an application's repository (e.g. application's server side source code). Number of contributors has been extracted for each application's snapshot series and correlation tests were performed between the number of contributors and the different issue category densities. Figure 5.14 presents a boxplot of the number of contributors found in snapshots in the dataset.

It can be seen that the data is extremely right-skewed, with most of the snapshots exhibiting only a few contributors. The number of contributors influencing the development of Android applications is rather low, with more than 65% of snapshots having only 1 contributor. Furthermore, only 10% of snapshots contain 4 or more contributors.

Table 5.4: Correlation coefficients for the number of contributors on issue categories. Spearman rank-correlation coefficient is expressed with ρ, Kendall with τ.

Type	ρ	p_ρ	τ	p_τ
MT	0.04	<2.2e-16	0.03	<2.2e-16
US	0.06	2.716954e-15	0.05	2.160009e-15
UC	0.06	<2.2e-16	0.05	<2.2e-16
UI	0.08	<2.2e-16	0.06	<2.2e-16
MC	0.11	<2.2e-16	0.08	<2.2e-16
DP	0.03	<2.2e-16	0.01	<2.2e-16
CSM	0.06	1.092046e-14	0.03	2.109424e-14

With this information in mind, two correlation tests were performed and calculated, namely Spearman's rank correlation coefficient and Kendall's rank correlation coefficient. Spearman and Kendall rank correlation coefficients were chosen as both of these coefficients do not assume a normal distribution of data. As the number of contributors distribution in this dataset is quite skewed, Spearman's rank correlation coefficient has been chosen over Pearson. Furthermore, Kendall's Tau rank coefficient has been calculated for correlation data as it appears to be less sensitive to ties in the dataset or errors. As the statistical correlation tests have been applied multiple times to the variables, it is necessary to accordingly correct the $p - values$ in order to test for significance of results. *Holm* correction method has been chosen to correct the $p - values$. The results of the correlations and corrected $p - values$ for both coefficients are presented in Table 5.4. The coefficient's have been rounded down to two decimal spaces. The ρ value in the table presents Spearman's rank correlation coefficient, p_ρ are the associated Holm-corrected $p - values$ for Spearman's rank correlation coefficients, τ presents Kendall's rank correlation coefficient values, and finally p_τ are corresponding Holm-corrected $p - values$ for Kendall's rank correlation coefficients.

From the table, it can be noticed that all issue categories show significance with respect to the Holm-corrected $p - values$. Seeing as the correlations for all issue categories are very weak, it can be said with reasonable confidence that the number of contributors to an application's repository does not impact the different issue densities.

Investigation of the relationship between the number of contributors and different issue densities continues for each issue category, this time separated by the previously identified clusters. Therefore, for each issue category the correlation coefficients have been calculated for all applications in those clusters. The clusters used here follow the best chosen models presented at the beginning of this section. The situation differs in several instances from the general correlation coefficients.

The results of applying the Spearman correlation test on different clusters and the resulting Holm-corrected $p-values$ are shown in Table 5.5. Positive values in the table state that there exists a positive correlation between variables, i.e. with the increasing value in one variable, the other is expected to increase in value as well. Negative coefficients state that with an increase in value in one variable, the other variable's value will proportionally decrease. Therefore, the correlation coefficient's range is from -1–1 inclusive, -1 showing the strongest possible negative correlation while 1 shows the strongest possible positive correlation. From the table it can be noticed that two clusters do not show significance with respect to the $p - values$, the middle cluster of Unit Size and the middle cluster of Unit Interfacing issue categories. The rest of the correlations show significance. Although the correlations across clusters slightly differ from the general correlations, the resulting rank coefficients are still very weak.

5.4 RQ4: How does the amount of source code changes impact the maintainability issue density of an Android application over time?

It can be seen from the previous question that variability in the number of contributors to an Android application's development does not have a meaningful impact on the amount of maintainability issues and therefore, the issue density. However, what about the changes these contributors have made to the application's source code in between different snapshots? In this research this change is measured via a metric called `Code Churn`, corresponding to the absolute difference between added, changed and removed lines for each snapshot. Apart from code churn, for this question, Spearman's rank correlation coefficient has also been calculated for the composing metrics separately, namely amount of added LOC, deleted LOC and changed LOC. The resulting correlation coefficients are shown in Table 5.6. For all statistically assessed issue categories, the corrected $p - values$ have shown significance. The reported $p - values$ can be found in the Appendix section 10.2.

Table 5.5: Correlation between team size and different issue densities, calculated for each issue category clusters.

Type	Cluster 1	Cluster 2	Cluster 3
MT	-0.01	0.09	-
p_{MT}	<2.2e-16	<2.2e-16	-
US	0.09	-0.01	0.02
p_{US}	<2.2e-16	0.469	1.263339e-11
UC	0.05	0.04	0.02
p_{UC}	4.620731e-06	1.015995e-06	2.172125e-08
UI	0.19	0.01	0.04
p_{UI}	<2.2e-16	0.67	5.552502e-05
MC	0.02	0.07	-
p_{MC}	<2.2e-16	<2.2e-16	-
DP	-0.14	0.08	-0.04
p_{DP}	<2.2e-16	<2.2e-16	0.00112984
CSM	0.04	-0.04	-0.18
p_{CSM}	<2.2e-16	3.386791e-05	<2.2e-16

Table 5.6: Spearman's correlation coefficient for source code change metrics on issue categories.

Metric	MT	US	UC	UI	MC	DP	CSM
Code churn	0.06	-0.02	0.04	0.08	0.11	0.08	0.02
New LOC	0.06	-0.02	0.04	0.08	0.11	0.08	0.02
Modified LOC	0.07	-0.02	0.06	0.09	0.12	0.08	0.03
Deleted LOC	0.05	-0.02	0.04	0.08	0.11	0.07	0.02
Snapshot Size (LOC)	0.50	0.13	0.40	0.47	0.55	0.50	0.21

The results exhibit similar traits as the previously presented correlations between team size and different issue densities, where for all categories the correlation coefficient is rather low. Furthermore, all of the issue categories exhibit a slight positive correlation with the exception of Unit Size, exhibiting a slight negative correlation. The highest correlations can be identified between the actual Snapshot Size, measured in LOC, and the different issue categories. With the addition of LOC to snapshots, it can be seen that Module Coupling, Duplication and Maintainability issue densities are expected to grow the most. Apart from Snapshot Size, the rest of the source code changes metrics exhibits very weak correlations, therefore, it is reasonably safe to conclude that the amount of source code changes does not impact the density of maintainability issues or its subcategories.

After presenting the general results, a more detailed overview into the impact of code churn on different issue densities is presented through the clusters. As was done for the number of contributors, each issue category has been tested for correlation using Spearman's rank correlation coefficient on the best clustered model. Table 5.7 presents the resulting correlations alongside corrected $p - values$. Results that did not show significance with respect to the corrected $p - values$ appeared in the last Unit Complexity cluster and for two clusters in Code smells issue categories. Rest of the correlations showed significance. Similarly to the impact of the number of contributors on issue density, the correlations between code churn and different issue densities are very weak, therefore suggesting that code churn does not impact the issue density of Maintainability or its subcategories.

Table 5.7: Spearman's correlation coefficient and corrected $p - values$ for source code change metrics on different clusters.

Type	Cluster 1	Cluster 2	Cluster 3
MT	-0.01	-0.13	-
p_{MT}	<2.2e-16	4.458239e-06	-
US	-0.13	-0.05	-0.01
p_{US}	<2.2e-16	1.145016e-07	2.613755e-04
UC	0.03	-0.06	0.003
p_{UC}	0.001685644	1.066206e-15	0.3607928
UI	-0.08	0.008	-0.11
p_{UI}	9.508206e-06	0.01852649	<2.2e-16
MC	-0.10	0.05	-
p_{MC}	<2.2e-16	<2.2e-16	-
DP	-0.02	0.07	-0.05
p_{DP}	0.02077543	<2.2e-16	4.684793e-04
CSM	-0.02	-0.008	-0.03
p_{CSM}	<2.2e-16	0.3712976	0.2457036

6

Discussion

This section discusses the findings presented in the Results and their implications. Obtained clusters, distributions and correlations constitute a foundation for presenting the up-to-date overview of Maintainability of Android Open-source applications. Furthermore, comparing different Android components and investigating different relationships between different issue categories allows for creation of a set of best case practices to serve as a guidance material for new and existing Android developers and researchers.

Having obtained the SIG ratings for Maintainability and its sub-characteristics, it is possible to reason about the general quality of applications and compare their maintainability to the market average. It is visible that Testability of Android applications fares the best among these subcategories, seeing as more than 75% of all applications have a Testabilty score of 4 stars or higher. As Testability is affected by the Volume and Unit Complexity of the applications, one can conclude that the application's are rather small with respect to Volume and the units contained within the source code exhibit lower complexity than the rest of the market. This comes to no surprise, as numerous Android applications cannot reach the Volume or Unit Complexity levels of, for example, enterprise software systems. Regarding Maintainability in general, mean value of 3.126 corresponds to a medium rating of 3 stars, implying that the overall maintainability of Android applications follows the market average.

Clustering different maintainability issue categories brought rise to some meaningful insights. Firstly, **most of the applications follow a rather stable evolution trend** with slight oscillations with respect to their issue density. Furthermore, the most noticeable differences and unstability in evolution trends was found in general until the first quarter of the development of an application, suggesting that this period is the most unstable and brings the most source code changes. This is understandable, as application development evolves rapidly in the beginning, with numerous functionalities being added or removed throughout development, and at later stages when the application has been published or almost finished, a stable period with few changes to the source code appears. In the next paragraph an overview into the identified clusters is presented, alongside implications for each maintainability issue category.

- **Maintainability** A stable trend and a sharp increase trend
- **Unit Size** A sharp increase trend, a sharp decrease and a stable trend
- **Unit Complexity** A sharp decrease trend, a sharp increase and a stable trend
- **Unit Interfacing** A sharp decrease trend, a stable and a sharp increase trend
- **Module Coupling** A stable trend and a sharp increase trend
- **Duplication** A slight decrease trend, a stable trend and a constant increase trend
- **Code smells** Stable trend, a slight increase trend and a sharp decrease trend

What can be concluded from this list is that **unit related metrics follow similar evolution trends**, with slight changes in the magnitude of their respective issue densities. However, out of the three unit related issue categories, Unit Size and Unit Complexity exhibit similar trends both in evolution and in density. Unit Interfacing, on the other hand, exhibits more stability with respect to the evolution trends, containing numerous apps with stable low densities over time. However, **Unit Complexity exhibits the lowest overall density values**. SIG Maintainability Model penalizes maintainability subcategories for having a high unit complexity, namely Modifiability and Testability. Looking at the SIG ratings overview table, it can be easily noticed that these two subcategories exhibit the highest mean values out of all subcategories in the model. This finding suggests that the Android application's complexity of units is a) generally low for all applications, and b) lower than the market average unit complexity. With respect to the amount of duplicated code contained within the source code of Android apps, the resulting issue density is generally much higher than compared to the rest of the categories. This finding suggests that duplicated code poses numerous threats to maintainability of Android apps, as it seems to constitute most of the overall maintainability issue density. The Duplication evolution trends suggest that duplicated code that is introduced at the beginning of the application development becomes increasingly harder to identify and remove over time, as the decreasing trend noticed in the highest density cluster exhibits only a slight decrease. Furthermore, with the noticeable constant growth, whether sharp or steady, in the remaining clusters, it can be said that **duplicated code blocks in general are not trivially removed**, implying these redundant code blocks can cause significant maintenance efforts at later development stages. A separate section of issue categories is represented by Code smells issue category, exhibiting maintainability issues related to bad code practices and potentially vulnerable source code lines. The stable cluster in this category contains most of the applications and shows the lowest density values out of the three identified evolution trends, suggesting that **developers do take Code smells issues seriously and pay attention to them**. Comparing to Duplication issue evolution trends, where the issue density either grows sharply or decreases slowly, Code smells issue density has an overall tendency of decreasing. In the cluster exhibiting a sharp decline evolution trend, it can be seen that applications that start out with a high Code smells issue density swiftly decrease the density over time. The same decrease can be noticed once again after the small increase in the same cluster, suggesting that developers have more ease in removing these Code smells issues from their applications. Furthermore, compared to Duplication, Code smells issues could prove of more importance to developers than redundant code blocks. Planned bug fixes are common in the software development business, which could also explain the decreases in Code smells issue density over time. **As a best practice advice, developers should strive for minimizing the density of Code smells issues to prevent further maintenance efforts**. As mentioned before, tools that support identifying and preventing these issues are Lint, Findbugs, BCH, etc. Regarding issue distributions across different Android components, several meaningful findings were found. Firstly, **most of the issues revolve around duplicated code or code smells issues**, and this closely follows the distributions presented in Figure 5.2 at the beginning of the Results section. Although Activities contain overall the most identified issues, Services contain the most maintainability issues per component. This finding suggests that, according to this study, **Services are the most vulnerable Android component with respect to maintainability**, and that these components will require the most maintenance efforts to solve these issues. Apart from Services being the most vulnerable component according to our study, **BroadcastReceiver components exhibit the least average amount**

of maintainability issues per component. Seeing as BroadcastReceivers are generally used as a means of communication between different applications or between an application's component, a suggestion is to keep these components rather simple. Looking at the distributions, it can be seen that the complexity of these units is rather low. However, **most of the issues in BroadcastReceivers stem from Code smells issue category**, implying that these components exhibit the most source code bugs or potential maintainability risks. This finding is closely related to the concept of *Broadcast receiver's connector envy*, presented in the work done by Bagheri et al [5], where it is said that BroadcastReceivers are complex by design and that developers should stray away from introducing application logic into these components, as they decrease maintainability and increase complexity.

With respect to ContentProviders' issue distributions, the most noticeable deviation in distribution is with respect to Unit Interfacing issues. ContentProviders heavily rely on the use of interfaces to allow communication data sharing between application's, therefore it is expected that the number of issues related to interfacing components will be higher. However, ContentProviders also exhibit the highest amount of Unit Complexity and Unit Size issues compared to other components, and more than 60% of all identified ContentProviders issues contained within unit-related issue categories. **This finding suggests that ContentProviders contain the largest, unmaintainable units**, and are therefore complex with respect to readability and maintenance work. Presented below is a short summary of the most vulnerable sections for each of the Android components.

- **Activities**: Duplicated, redundant code blocks
- **Services**: Duplicated code blocks alongside bad code practices and bugs
- **BroadcastReceivers**: bad code practices and bugs
- **ContentProviders**: Units (size, complexity, number of incoming parameters)

Regarding correlations and the impact of changing development team size and the amount of source code changes on different issue densities, the main impression from the correlation tests is that both of the categories do not have a significant impact on the different issue densities. With a highest correlation coefficient of 0.11 between the number of contributors and Module Coupling issue category, there is a lack of significant impact on the amount of issues. Therefore, it is safe to say that **Number of contributors does not impact the density of maintainability issues.** However, observing applications from the viewpoint of clusters identified for each issue category, more detailed results can be obtained. The correlations now vary in some difference between the clusters in different issue categories, however the correlation coefficients are still very low for all performed correlations. With respect to the impact of code churn on different issue densities, the findings are similar to the impact of number of contributors, i.e. the correlation coefficients are very low for all issue categories. Furthermore, additional correlation tests were performed to test detailed differences of source code additions, deletions or modifications on the amount of maintainability issues. These results also showed weak correlations, therefore it can be said that the **amount of source code changes between snapshots does not impact the density of maintainability issues.** However, looking at the correlations between the total size of a snapshot and different issue categories, some categories exhibit stronger positive correlations. Most sensitive to overall changes in LOC seem to be Module Coupling, Duplication and Maintainability. This finding suggests that **increases in snapshot size usually introduce new duplication and coupling issues for applications**, which are reflected in the overall increase of maintainability issue counts. This correlation can be seen in the smallest Duplication issue category evolution cluster, exhibiting a constant growth over time in issue density. As time progresses, the application's codebase grows, and this trend follows in line with the growing issue density.

Following from the presents results and findings, a set of best practices has been created that provides guidelines to Android developers and researchers, with the most common pitfalls identified with respect to the amount of maintainability issues. This set of best practices can serve as a guide into reducing the overall application maintenance efforts, alongside improving overall application's source code quality.

- Developers should pay special attention to duplicated code when working on Android activities
- While developing Services, special attention should be paid to the overall Maintainability, as Services exhibit the most issues on average
- BroadcastReceiver components are very complex to develop and understand, therefore developers should avoid introducing complex logic and keep the BroadcastReceivers simple
- Keep units manageable with respect to their size, complexity and the amount of input parameters when developing ContentProviders

- If an application's source code changes often and in sizeable portions, coupling and duplication issues are more likely to be introduced
- As the beginning of an application's development exhibits the most unstability with respect to issue density evolution, it is recommended that the codebase is measured and tracked with respect to maintainability to adjust maintenance efforts and reduce the possibility of a high issue density at the beginning of the development

7 Threats to Validity

This section presents the threats to validity for this thesis project.

Construct validity threats consider the relationship between theoretical knowledge and actual observations. In particular, results in this research can be affected by the implementation and performance of the proprietary static code analysis tool, SAT. It is possible that the tool has detected some false positive issues in certain cases. However, this is mitigated with a few methods. Firstly, tool documentation was made available to the researchers explaining necessary configuration steps. Furthermore, an interview has been conducted with one of the tool developers to investigate the details related to the reporting of identified maintainability issues. Afterwards, a manual inspection of several reported issues across different issue categories was performed, to investigate the correctness of the reported issues. Lastly, this threat is mitigated with the fact that SAT is an industry tested tool used on a daily basis, and in previous research [26], [4], [3].

With respect to *Conclusion validity*, threats considering the relationship between treatment and outcome are investigated. In this research, statistical analysis assumptions have been accounted for, to minimize the possibility of false results being reported. For correlation tests performed in RQ3 and RQ4, non-parametric statistical correlation tests have been applied, accounting for the distribution of the data. Regression analysis assumptions and k-means limitations have also been accounted for. With regression, the analysis accounted for a minimum number of data points and for a minimum variability in each application's evolution, and applications that did not satisfy the requirements have been excluded. However, excluding apps that do not satisfy regression requirements resulted in issue categories having different total number of clustered apps. Therefore, this reduces the possibility to compare the clusters cross-issue. To avoid conclusion bias based on differing number of apps, the numbers have been reported and conclusions were not made based on cross-issue related data. For k-means cluster analysis, input features have been standardized and scaled accordingly, so that the k-means results are plausible and meaningful. Furthermore, supportive methods have been applied to guide the optimal number of clusters selection, resulting in executing k-means cluster analysis multiple times for each issue, improving the clustering results.

Threats that could have influenced the results of the research are considered as *Internal validity* threats. In this research, first internal validity threat considers the configuration of exclusion files for static code analysis. As repositories containing the application source code differ in structure, it is possible to obtain false results with the inclusion of non-application related source code (e.g. third party library code or source code developed for other application platforms). This threat has been mitigated with the specifications before and during the static code analysis. Firstly, for all applications a root application folder containing the Android source code has been identified, and subsequent repository metrics have been collected only for the source code contained within this directory. Furthermore, SAT tool allows for exclusions and inclusions of different files during static code analysis, and this was exploited in the sense that .jar files and library directories have been excluded from the analysis. Another threat relates to the exclusion of component related maintainability metrics from the measurements. In this research, considering the large sample size of the applications in the dataset, it was unfeasible to manually investigate and specify correct architectural components. However, these metrics are mostly concerning the results reported in the SIG Maintainability Model, while most of this research has been focused on the actual reported maintainability issues rather than the modeled ratings.

External validity threat consider the generalizability of findings. In this research a relatively large dataset has been considered, and the selected sample of applications comes from a real-world setting (i.e. applications are published on Google Play Store). Furthermore, only open-source applications have been considered in this research, therefore it would be interesting to extend the range of the research to other types of applications and settings.

Regarding reliability, with this thesis project a replication package is made available. The replication package consists of scripts allowing for software repository data mining and collection of non-proprietary metrics. Furthermore, the static code analysis scripts allow for inclusion of a different static code analysis tool, as the one used in this research is proprietary and is not freely available.

8

Future Work

Future work aims at extending the research to perform a qualitative analysis of snapshots in identified maintainability issue density evolution trends, with the goal of uncovering deeper insights into the different identified clusters of applications. With the use of mined commit messages from the application's repositories, more insights into rationales behind sudden changes in evolution trends could be identified. Furthermore, extending the research to different source code characteristics of Android applications would allow for the creation of a source code quality guidance tool that could guide developers through the creation of performant, maintainable applications. Extending the research with the inclusion of multiple static code analysis tools allows for a more objective comparison of different maintainability issues contained within the applications. It would be interesting to see how the maintainability issue evolution trends compare between the different static code analysis tools and whether they exhibit differences or similarities. Last but not least, questions related to the duration of maintainability issues should be answered. For example, an investigation into the average duration of maintainability issues in applications could be performed. When do these maintainability issues first appear in Android applications, and more importantly, how are these maintainability issues solved? Is it due to planned maintenance efforts or by chance? These are all the questions worth answering in order to advance the body of knowledge related to maintainability of Android applications.

9 Conclusion

In this master thesis project, a large scale empirical experiment aimed at uncovering maintainability issue evolution trends has been performed on 2,238 Android Open-source applications, originating from sources such as Github, FDroid and Wikipedia. A static code analysis with the help of proprietary SAT tool has been performed on equal historical intervals for each application. Following the static code analysis, regression analysis in combination with k-means clustering has been performed in order to extract maintainability issue evolution trends.

When analyzing maintainability issue evolution, different trends have been identified, such as periods of *Sharp increase* in issue density, *Sharp decrease*, *Slight increase* or *Slight decrease.* However, most of the applications exhibit a *Stable* evolution trend. This finding aligns with the findings uncovered in the work done by Penta et al. [30], where the researchers discovered that most of the application's exhibit a stable evolution trend with respect to vulnerability density. Related to different maintainability issue categories, in this research Duplication issue category is the most commonly identified maintainability issue, followed by fine-grained Code smells issues. Most of the identified maintainability issues reside in Activities, however Services exhibit the highest proportion of maintainability issues compared to other Android components. Furthermore, Services, alongside BroadcastReceivers, exhibit a high number of Code smells issues, suggesting these components contain the most bugs in their source code, and are more complex to develop and understand. However, it can be seen that Code smells issues tend to be removed swiftly, with respect to evolution trends exhibiting sharp decreases in Code smells issue densities. Further qualitative analysis of these issues is required to confirm whether these removals of Code smells issues are a result of planned maintenance work or bug fixes. Investigation of correlations between two influencing factors of maintainability, namely *Team size* and the amount of *Code churn* resulted in correlations of weak effect. The correlations exhibit the same weak effects when looked at from different cluster grouping, apart from Snapshot Size, where it appears that Module Coupling and Duplication issue densities are the most susceptible to a growing codebase (based on LOC).

The outcome of this research and the findings presented serve as an overview into the state of the art of maintainability issues in open-source android applications. Android developers can use the identified maintainability issue evolution trends and findings to support their application development. Findings in this research can be used as a guidance into managing the maintainability of Android applications, and with them developers can focus their attention on the most vulnerable Android components and the most common maintainability issues.

10

Appendix

10.1 Maintainability issue source code examples

The examples for the identified source code Maintainability issue examples have been taken extracted from the snapshots of apps in the dataset.

10.1.1 Unit Size

Listing 10.1: The size of this unit is too large

```java
private void logAuthorizationComplete(AuthorizationClient.Result.Code result, Map<String, String> resultExtras,
        Exception exception) {
    Bundle bundle = null;
    if (pendingAuthorizationRequest == null) {
        // We don't expect this to happen, but if it does, log an event for diagnostic purposes.
        bundle = AuthorizationClient.newAuthorizationLoggingBundle("");
        bundle.putString(AuthorizationClient.EVENT_PARAM_LOGIN_RESULT,
                AuthorizationClient.Result.Code.ERROR.getLoggingValue());
        bundle.putString(AuthorizationClient.EVENT_PARAM_ERROR_MESSAGE,
                "Unexpected␣call␣to␣logAuthorizationComplete␣with␣null␣pendingAuthorizationRequest.");
        } else {
            bundle = AuthorizationClient.newAuthorizationLoggingBundle(
                ↪ pendingAuthorizationRequest.getAuthId());
            if (result != null) {
                bundle.putString(AuthorizationClient.
                    ↪ EVENT_PARAM_LOGIN_RESULT, result.getLoggingValue())
                    ↪ ;
            }
            if (exception != null && exception.getMessage() != null) {
                bundle.putString(AuthorizationClient.
                    ↪ EVENT_PARAM_ERROR_MESSAGE, exception.getMessage());
            }

            // Combine extras from the request and from the result.
            JSONObject jsonObject = null;
            if (pendingAuthorizationRequest.loggingExtras.isEmpty() ==
                ↪ false) {
                jsonObject = new JSONObject(pendingAuthorizationRequest.
                    ↪ loggingExtras);
            }
            if (resultExtras != null) {
                if (jsonObject == null) {
                    jsonObject = new JSONObject();
                }
                try {
                    for (Map.Entry<String, String> entry : resultExtras.
                        ↪ entrySet()) {
                        jsonObject.put(entry.getKey(), entry.getValue());
                    }
                } catch (JSONException e) {
                }
            }
            if (jsonObject != null) {
                bundle.putString(AuthorizationClient.EVENT_PARAM_EXTRAS,
                    ↪ jsonObject.toString());
            }
        }
        bundle.putLong(AuthorizationClient.EVENT_PARAM_TIMESTAMP, System.
            ↪ currentTimeMillis());

        AppEventsLogger logger = getAppEventsLogger();
        logger.logSdkEvent(AuthorizationClient.EVENT_NAME_LOGIN_COMPLETE,
            ↪  null, bundle);
    }
```

10.1.2 Unit Complexity

Listing 10.2: Cyclomatic complexity of this unit is too high

```
    @Override
    public void showAds(Hashtable<String, String> adsInfo, int pos) {
        final Hashtable<String, String> curInfo = adsInfo;
        final int curPos = pos;
        PluginWrapper.runOnMainThread(new Runnable(){
            @Override
            public void run() {
                try
                {
                    String spaceID = curInfo.get("FlurryAdsID");
                    if (null == spaceID || TextUtils.isEmpty(spaceID))
                    {
                        LogD("Value␣of␣'FlurryAdsID'␣should␣not␣be␣empty");
                        return;
                    }

                    String strSize = curInfo.get("FlurryAdsSize");
                    int size = Integer.parseInt(strSize);
                    if (size != 1 && size != 2 && size != 3) {
                        LogD("Valur␣of␣'FlurryAdsSize'␣should␣be␣one␣of␣
                            ↪ '1',␣'2',␣'3'");
                        return;
                    }

                FlurryAdSize eSize = FlurryAdSize.BANNER_TOP;
                switch (size)
                {
                case 1:
                    eSize = FlurryAdSize.BANNER_TOP;
                    break;
                case 2:
                    eSize = FlurryAdSize.BANNER_BOTTOM;
                    break;
                case 3:
                    eSize = FlurryAdSize.FULLSCREEN;
                    break;
                default:
                    break;
                }

                if (null == mWm) {
                    mWm = (WindowManager) mContext.getSystemService("window");
                }
                if (null != mBannerView) {
                    mWm.removeView(mBannerView);
                    mBannerView = null;
                }
                mBannerView = new FrameLayout(mContext);
                AdsWrapper.addAdView(mWm, mBannerView, curPos);

                FlurryAds.fetchAd(mContext, spaceID, mBannerView, eSize);
            } catch (Exception e) {
                LogE("Error␣during␣showAds", e);
            }
        }
    });
}
```

10.1.3 Unit Interfacing

Listing 10.3: This method contains several input parameters

```
        public static void createTextBitmap(String string, final String
            ↪ fontName, final int fontSize, final int alignment, final
            ↪ int width, final int height) {

        //
        createTextBitmapShadowStroke( string, fontName, fontSize, 1.0f,
            ↪ 1.0f, 1.0f,  // text font and color
            alignment, width, height,       // alignment and size
            false, 0.0f, 0.0f, 0.0f, 0.0f,    // no shadow
            false, 1.0f, 1.0f, 1.0f, 1.0f);       // no stroke

        }
```

10.1.4 Module Coupling

Listing 10.4: The class (module) contains multiple method invocations

```
/**
 * Copyright 2010-present Facebook.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      ↪ implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

```

```
package com.facebook.internal;

import android.util.Log;
import com.facebook.LoggingBehavior;
import com.facebook.Settings;

import java.util.HashMap;
import java.util.Map;

/**
 * com.facebook.internal is solely for the use of other packages within
    ↪ the Facebook SDK for Android. Use of
 * any of the classes in this package is unsupported, and they may be
    ↪ modified or removed without warning at
 * any time.
 */
public class Logger {
    public static final String LOG_TAG_BASE = "FacebookSDK.";
    private static final HashMap<String, String> stringsToReplace = new
        ↪ HashMap<String, String>();

    private final LoggingBehavior behavior;
    private final String tag;
    private StringBuilder contents;
    private int priority = Log.DEBUG;

    // Note that the mapping of replaced strings is never emptied, so it
        ↪ should be used only for things that
    // are not expected to be too numerous, such as access tokens.
    public synchronized static void registerStringToReplace(String
        ↪ original, String replace) {
        stringsToReplace.put(original, replace);
    }

    public synchronized static void registerAccessToken(String
        ↪ accessToken) {
```

```java
        if (Settings.isLoggingBehaviorEnabled(LoggingBehavior.INCLUDE_ACCESS_TOKENS) == false) {
            registerStringToReplace(accessToken, "ACCESS_TOKEN_REMOVED");
        }
    }

    public static void log(LoggingBehavior behavior, String tag, String string) {
        log(behavior, Log.DEBUG, tag, string);
    }

    public static void log(LoggingBehavior behavior, String tag, String format, Object... args) {
        if (Settings.isLoggingBehaviorEnabled(behavior)) {
            String string = String.format(format, args);
            log(behavior, Log.DEBUG, tag, string);
        }
    }

    public static void log(LoggingBehavior behavior, int priority, String tag, String string) {
        if (Settings.isLoggingBehaviorEnabled(behavior)) {
            string = replaceStrings(string);
            if (tag.startsWith(LOG_TAG_BASE) == false) {
                tag = LOG_TAG_BASE + tag;
            }
            Log.println(priority, tag, string);

            // Developer errors warrant special treatment by printing out a stack trace, to make both more noticeable,
            // and let the source of the problem be more easily pinpointed.
            if (behavior == LoggingBehavior.DEVELOPER_ERRORS) {
                (new Exception()).printStackTrace();
            }
        }
    }

    private synchronized static String replaceStrings(String string) {
        for (Map.Entry<String, String> entry : stringsToReplace.entrySet
            ↪ ()) {
            string = string.replace(entry.getKey(), entry.getValue());
        }
        return string;
    }

    public Logger(LoggingBehavior behavior, String tag) {
        Validate.notNullOrEmpty(tag, "tag");

        this.behavior = behavior;
        this.tag = LOG_TAG_BASE + tag;
        this.contents = new StringBuilder();
    }

    public int getPriority() {
        return priority;
    }

    public void setPriority(int value) {
        Validate.oneOf(value, "value", Log.ASSERT, Log.DEBUG, Log.ERROR,
            ↪ Log.INFO, Log.VERBOSE, Log.WARN);

        priority = value;
    }

    public String getContents() {
        return replaceStrings(contents.toString());
    }

    // Writes the accumulated contents, then clears contents to start
        ↪ again.
    public void log() {
        logString(contents.toString());
        contents = new StringBuilder();
    }

    // Immediately logs a string, ignoring any accumulated contents,
        ↪ which are left unchanged.
    public void logString(String string) {
        log(behavior, priority, tag, string);
    }

    public void append(StringBuilder stringBuilder) {
        if (shouldLog()) {
            contents.append(stringBuilder);
        }
    }

    public void append(String string) {
        if (shouldLog()) {
            contents.append(string);
        }
    }

    public void append(String format, Object... args) {
        if (shouldLog()) {
            contents.append(String.format(format, args));
        }
    }

    public void appendKeyValue(String key, Object value) {
        append("␣␣%s:\t%s\n", key, value);
    }

    private boolean shouldLog() {
        return Settings.isLoggingBehaviorEnabled(behavior);
    }
}
```

10.1.5 Duplication

Listing 10.5: This code block is found in one of the source code files...

```
        mDrawerToggle.setDrawerIndicatorEnabled(false);
    }
}

@Override
protected void onResume() {
    super.onResume();

    // We want to track a pageView every time this activity gets the
        ↪ focus - but if the activity was
    // previously destroyed we could have lost our global data, so this
        ↪ is a bit of a hack to avoid a crash!
    if (GRTApplication.tracker == null) {
        Log.e(TAG, "null␣tracker!");
        startActivity(new Intent(this, FavstopsActivity.class));
    }
}
```

Listing 10.6: ...and is repeated in entirety in another source code file

```
        mDrawerToggle.setDrawerIndicatorEnabled(false);
    }
}

@Override
protected void onResume() {
    super.onResume();

    // We want to track a pageView every time this activity gets the
        ↪ focus - but if the activity was
    // previously destroyed we could have lost our global data, so this
        ↪ is a bit of a hack to avoid a crash!
    if (GRTApplication.tracker == null) {
        Log.e(TAG, "null␣tracker!");
        startActivity(new Intent(this, FavstopsActivity.class));
    }
}
```

10.1.6 Code Smells

Listing 10.7: Empty catch blocks are not allowed

```
        } catch (InterruptedException ie) {
            // so the screen might be slightly wrong; oh well.
        }
```

Listing 10.8: Catching basic Exception is also not allowed

```
} catch (final Exception e) {
    Log.e(TAG, "unknown␣exception␣exception");
    e.printStackTrace();
}
```

Listing 10.9: on line 9 control variable i has been modified

```
 1          for (int i = 1; i <= charLength; ++i) {
 2              tempWidth = (int) Math.ceil(paint.measureText(string, start,i)
                    ↪ );
 3              if (tempWidth >= maxWidth) {
 4                  final int lastIndexOfSpace = string.substring(0, i).
                        ↪ lastIndexOf("␣");
 5
 6                  if (lastIndexOfSpace != -1 && lastIndexOfSpace > start) {
 7                      /* Should wrap the word. */
 8                      strList.add(string.substring(start, lastIndexOfSpace))
                            ↪ ;
 9                      i = lastIndexOfSpace + 1; // skip space
10                  } else {
11                      /* Should not exceed the width. */
12                      if (tempWidth > maxWidth) {
13                          strList.add(string.substring(start, i - 1));
14                          /* Compute from previous char. */
```

```
                    --i;
                } else {
                    strList.add(string.substring(start, i));
                }
            }
            /* Remove spaces at the beginning of a new line. */
            while (i < charLength && string.charAt(i) == ' ') {
                ++i;
            }

            start = i;
        }
    }
```

10.2 Corrected p-values for Spearman's correlation with source code changes on issue densities

Table 10.1: Spearman's correlation coefficient for source code change metrics on issue categories.

category	Code churn	New LOC	Modified LOC	Deleted LOC	Snapshot Size
MT	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16
US	<2.2e-16	<2.2e-16	<2.2e-16	3.874475e-16	<2.2e-16
UC	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16
UI	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16
MC	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16
DP	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16
CSM	<2.2e-16	<2.2e-16	<2.2e-16	1.093757e-11	<2.2e-16

References

[1] ISO/IEC 14764:2006. 2006. Software Engineering - Software Life Cycle Processes - Maintenance. (2006). `https://www.iso.org/standard/39064.html` 7

[2] ISO/IEC 25010:2011. 2011. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. (2011). `https://www.iso.org/standard/35733.html` 8

[3] Tiago L. Alves. 2010. Determination of number of clusters in K-means clustering and application in colour segmentation. In *Proceedings of Simulation and EGSE facilities for Space Programmes (SESP2010)*. 75

[4] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering* 40, 11 (2014), 1100–1125. `https://doi.org/10.1109/TSE.2014.2342227` 75

[5] Hamid Bagheri, Joshua Garcia, Alireza Sadeghi, Sam Malek, and Nenad Medvidovic. 2016. Software architectural principles in contemporary mobile software: from conception to pratice. *The Journal of Systems and Software* 119, 1 (2016), 31–44. `https://doi.org/10.1016/j.jss.2016.05.039` 72

[6] Victor R. Basili. 1994. *Encyclopedia of Software Engineering, 2 Volume Set.* John Wiley & Sons, Inc. `https://www.cs.umd.edu/~basili/publications/technical/T89.pdf` 23

[7] Pierre Bourque and Richard E. Fairley. 2014. *Guide to the Software Engineering Body of Knowledge, Version 3.0.* IEEE Computer Society. 2

[8] Evans Data Corporation. 2016. Global Developer Population and Demographic Study 2017 Vol. 1. (2016). `https://evansdata.com/press/viewRelease.php?pressID=244` 1, 11

[9] Luis Corral and Ilenia Fronza. 2015. Better Code for Better Apps: A Study on Source Code Quality and Market Success of Android Applications. In *MOBILESoft '15 Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems.* ACM, 22–32. 13

[10] Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. 2016. A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps. In *ICSME '16 Proceedings of the 32nd International Conference on Software Maintenance and Evolution.* IEEE, 443–448. 25

[11] Chris Ding and Xiaofeng He. 2004. K-means Clustering via Principal Component Analysis. In *ICML '04 Proceedings of the twenty-first international conference on Machine learning.* ACM, 29–37. 41

[12] Till Döhmen, Magiel Bruntink, Davide Ceolin, and Joost Visser. 2016. Towards a Benchmark for the Maintainability Evolution of Industrial Software Systems. In *IWSM-MENSURA 2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement.* IEEE, 11–22. 9, 34

[13] Sebastian G. Elbaum and John C. Muson. 1998. Code Churn: A Measure for Estimating the Impact of Code Change. In *International Conference on Software Maintenance, ICSM '98.* IEEE, 24–31. 23

[14] Robert L. Glass. 2001. Frequently Forgotten Fundamental Facts about Software Engineering. (2001). `https://pdfs.semanticscholar.org/7eee/629b22cd3db63296cac13a0c37cb0a7235f6.pdf` 2

[15] John A. Hartigan. 1979. *Clustering Algorithms.* John Wiley & Sons, Inc. 40, 41

[16] John A. Hartigan and M. A. Wong. 1979. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society* 28, 1 (1979), 100–108. `https://doi.org/10.2307/2346830` 41

[17] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the Software Quality of Android Applications along their Evolution. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.* IEEE, 429–436. 17, 31

[18] Ilja Heitlager, Tobias Kuipers, and Visser. 2007. A Practical Model for Measuring Maintainability. In *QUATIC 2007. 6th International Conference on the Quality of Information and Communications Technology.* 137–143. `https://doi.org/10.1109/QUATIC.2007.8` vii, 10

[19] Julien Jacques and Christian Preda. 2014. Functional data clustering: a survey. *Advances in Data Analysis and Classification* 8, 3 (2014), 231–255. `https://doi.org/10.1007/s11634-013-0158-y` 41

[20] Anil K. Jain. 2010. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters* 31, 8 (2010), 654–666. `https://doi.org/10.1016/j.patrec.2009.09.011` 41

[21] Joost Visser. [n. d.]. SIG/TÜViT Evaluation Criteria Trusted Product Maintainability: Guidance for producers. ([n. d.]). `https://www.sig.eu/files/en/01_SIG-TUViT_Evaluation_Criteria_Trusted_Product_Maintainability-Guidance_for_producers.pdf` Accessed: 2017-08-23. 53

[22] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining Github. In *MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 92–101. 29

REFERENCES

[23] Tapas Kanungo, Nathan S. Netanyahu, and Angela Y. Wu. 2002. An Efficient k-Means Clustering Algorithm: Analysis and Implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 7 (2002), 881–892. https://doi.org/10.1109/TPAMI.2002.1017616 41

[24] Saurav Kaushik. [n. d.]. An Introduction to Clustering and different methods of clustering. ([n. d.]). https://www.analyticsvidhya.com/blog/2016/11/an-introduction-to-clustering-and-different-methods-of-clustering/ Accessed: 2017-08-17. 42

[25] Stefan Koch. 2007. Software evolution in open source projects - a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice* 19, 6 (2007), 361–382. https://doi.org/10.1002/smr.348 18

[26] Bart J.H. Luijten. 2009. *The Influence of Software Maintainability on Issue Handling.* Master's thesis. Faculty EEMCS, Delft University of Technology. 75

[27] Malika Charrad and Nadia Ghazzali and Veronique Boiteau and Azam Niknafs. [n. d.]. NbClust: Determining the Best Number of Clusters in a Data Set. ([n. d.]). https://CRAN.R-project.org/package=NbClust Accessed: 2017-08-21. 43

[28] Azad Naik. [n. d.]. k-means clustering algorithm. ([n. d.]). https://sites.google.com/site/dataclusteringalgorithms/k-means-clustering-algorithm Accessed: 2017-08-17. 41

[29] Robert F. Nau. [n. d.]. WhatâĂŹs a good value for R-squared? ([n. d.]). https://people.duke.edu/~rnau/rsquared.htm Accessed: 2017-08-17. 39

[30] Massimiliano Di Penta, Luigi Cerulo, and Lerina Aversano. 2009. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology* 51, 10 (2009), 1469 – 1484. 17, 31, 36, 81

[31] Guoqi Qian and Yuehua Wu. 2011. Estimation and Selection in Regression Clustering. *European Journal of Pure and Applied Mathematics* 4, 4 (2011), 455–466. 41

[32] Siddheswar Ray and Rose H. Turi. 1999. Determination of number of clusters in K-means clustering and application in colour segmentation. In *The 4th International Conference on Advances in Pattern Recognition and Digital Techniques.* 137–143. 41

[33] Thaddeus Tarpey. 2012. Linear Transformations and the k-Means Clustering Algorithm. *The American Statistician* 61, 1 (2012), 34–40. https://doi.org/10.1198/000313007X171016 37, 41

[34] Thaddeus Tarpey and Kimberly K. J. Kinateder. 2003. Clustering Functional Data. *Journal of Classification* 20, 1 (2003), 093–114. https://doi.org/10.1007/s00357-003-0007-3 38, 41

[35] Henri Theil. 1961. *Economic Forecasts and Policy.* North-Holland Pub. Co., 1961. 39

[36] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *ICSE '15 Proceedings of the 37th International Conference on Software Engineering*, IEEE Press Piscataway (Ed.). IEEE, 403–414. 18

[37] Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. 2016. *Building Maintainable Software, Java Edition.* O'Reilly Media, Inc. 15

[38] Kiri Wagstaff and Seth Rogers. 2001. Constrained K-means Clustering with Background Knowledge. In *Proceedings of the Eighteenth International Conference on Machine Learning.* ACM, 577–584. 41