Doctoral Thesis

Investigating Performance Issues in Mobile Apps

PhD Program in Computer Science: XXX cycle

Supervisor

Prof Massimiliano Di Penta
dipenta@unisannio.it

Author:

Teerath Das
teerath.das@gssi.it

Co-supervisor

Ivano Malavolta
i.malavolta@vu.nl

February 2020

GSSI Gran Sasso Science Institute
Viale Francesco Crispi, 7 - 67100 L'Aquila - Italy

Abstract

The world is moving at a dynamic pace, and this has led to the technological advancement of mobile applications. This rise in the advancement of the mobile application comes with critical concerns to end-users in terms of the performance, especially when implementing high intensive features. Moreover, enjoyable user experience in terms of performance is often considered as the main parameter to measure the success of any app. Poor implementation of source code, lack of developers knowledge, and time constraints on resolving performance issues are few of the major potential performance drawbacks in Android applications.

To overcome these performance issues, in this dissertation, we focus on investigating the performance-related issues in open-source Android application (mainly apps from GitHub). Our thesis can be divided into four key research objectives: (i) initially we investigate on the extent to which developers consider performance issues in their commits (while maintaining their apps) and how they document it, (ii) to complement this study, we conduct an experiment to study the evolution of Android specific performance issues detected by Android Lint, and based on the obtained results, (iii) we introduce an Eclipse plugin that can be used to automatically resolve seven types of performance-related issues detected by Android Lint; in addition to this, we performed a survey-based study to analyze the self assessed performance refactoring code of the proposed tool from the developers' perspective; and (iv) we design and conduct a measurement-based study to examine the impact of performance violations at run-time.

The key contributions of this thesis are (i) a taxonomy of developers' concerns about performance, obtained by applying card sorting technique on a dataset of commit messages extracted from GitHub, (ii) the empirical research considering seven types of performance issues identified by Lint tool resulted: (a) a taxonomy for different kinds of evolution patterns of Android performance issues emerged by tracing the history of Android apps, (b) a catalog of documented performance issues resolved by Android developers, (iii) an automatic refactoring tool to address the seven types of performance-related issues of Android Lint, (iv) developers perspective related to refactoring and non-refactoring code in the form of survey responses, and (v) a measurement-based study to analyze run-time performance of Android apps. These results provide developers a base to take the next leap in solving performance-related issues in the mobile apps of the future.

Acknowledgements

With utmost respect, I would like to thanks *Gran Sasso Science Institute (GSSI)* and *IMT Lucca* for providing me an opportuinity to purse my doctoral studies.

I would like to express all my gratitude to my *PhD* Supervisors, Prof. Massimiliano Di Penta, from *University of Sannio, Benevento (Italy)* and Ivano Malavolta from *Vrije Universiteit Amsterdam (The Netherlands)*. Their continuous guidance, support and advices throughtout the course of my research made my thesis work possible. I am deeply thankful to them.

I am highly indebted to Prof. Luca Aceto and Prof. Michele Flammini, the coordinators of the *PhD* program in Computer Science at GSSI for their kindness and willingness to help me at every stage, I needed. I am also very grateful to Prof. Rocco De Nicola for helping me in the initial days of my PhD.

Further, I would like to extend my thanks to the reviewers, who gave their precious time to review the thesis. I am also grateful to dissertation committee members, for devoting their time for my dissertation defense.

I am very much thankful to Prof. Mika Mantayla, *M3S Research Unit, University of Oulu (Finland)* for providing me work space and all the necessary help during my research visit. I would also like to thanks Prof. Patricia Lago from *Vrije Universiteit Amsterdam (The Netherlands)* for providing me working place during my research stay.

I am extermly grateful to my brother Tanesh Kumar (and his wife), who helped me finanicially and morally throughout the extension period of my PhD.

During the time of *PhD*, I feel bless to have friends who had been very supportive, specially Atith Sagar, Sagar Kumar, Rajinder Kumar, Syed M. Yasoob Bukhari and many more. I am also thankful to all the participants (developers), who took their time to participated in the survey.

Last but not least, I am grateful to my family specially my wife and parents, without their love, support, and patience, this journey will not be easy. I would like to dedicate this thesis work to my wife, family and all the sath sangat.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Mobile Applications and Performance

The current society is mainly driven by smart gadgets such as mobile phones, Personal digital assistant (PDAs), laptops and tablets to get digital services in various domains of life. It is expected that this dynamic trend is going to increase rapidly in the coming years due to technological advancements and digitalization with the number of connected devices increasing from 4.3 billion in 2019 to 7.2 billion in 2023 worldwide [68]. The widespread diffusion of smartphones is directly proportional with faster growth of mobile applications (apps)[1] to ease the human life in various key applications such as online shopping, social networking, and net banking etc. The underline apps economy is forecasted to rise from 62 billion USD in 2016 to 139 billion USD in 2021 [12], especially when the gaming app industry is projected to generate a revenue from 11.3 billion to 33.8 billion. Moreover, the time spent on surfing various mobile apps is 3.2 hours on an average for the age group between 18-24 [78] (Fig. 1.1 reports for other age groups as well). The Android operating system is playing the lead role in this phenomenal apps revolution, as it is popular and the most used among all the age segments in countries like the US and UK [78]. As of 2017 [13], nearly 3.5 million Android apps are published over the Google play store (The official online store for Android apps) and downloaded a billion times annually (82 billion downloaded in 2016).

The huge diffusion of mobile phones and apps stimulates end users to perform various tasks like managing a bank account, health-monitoring information, and buying online products etc. in a single touch. However, the continuous increase in users demand for new features (especially apps like games and video streaming) can trigger performance issues in mobile apps. Performance is a crucial aspect in any app because it can provide

[1] In the rest of the dissertation, we use apps as an acronym for applications

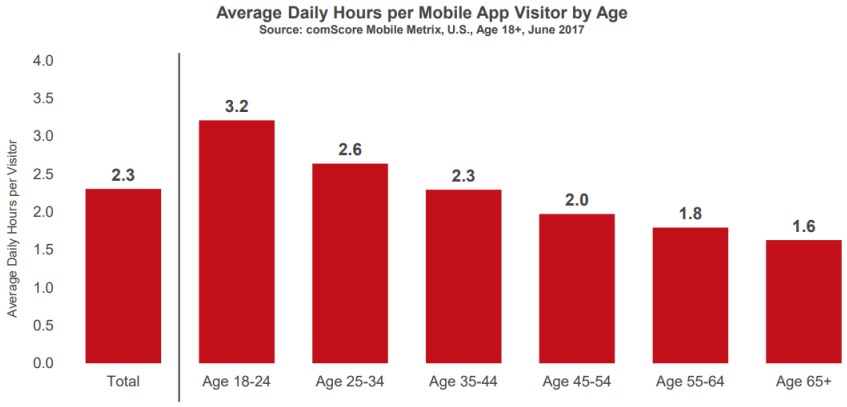


FIGURE 1.1: Number of hours spent by different age groups on mobile apps [78].

various critical mobile services in the cases where the margin of error is thin. A good case example can be monitoring accuracy and delay related to critical services for the aeronautical and military industry. Nevertheless, an enriched user experience increases good reviews and rating that ultimately result in the success of any app. Whereas apps which exhibit bad performance not only impede users to fully use some features but also may not survive long in the market competition.

Liu *et al.* [9, 83] randomly crawled 60,000 Android apps from Google Play Store and by analyzing the user reviews and release logs, they found that more than 11,108 apps were suffering or have suffered from at least one performance issue. Moreover, 75% of apps tend to have been uninstalled in the first three months [28] and poor performance is ranked as the second potential reasons for the app removal [27]. Therefore, it is very crucial to consider performance as a top-priority aspect for app developers during app development (*i.e.,* need to give more attention).

1.2 Challenges

In recent years, researchers are more concerned about performance issues for mobile apps and studied them from different dimensions. Liu *et al.* [83] conducted a study by identifying and characterizing 70 real-world performance bugs in 8 android application and came to a consensus by boiling them down to three main categories as GUI lagging (53/70), energy leaks (10/70), and memory bloat (8/70). Among these three categories, GUI lagging (53/70) is said to be more recurrent than energy leaks (10/70) and memory bloat (8/70). On the other hand, few studies are dedicated to analyzing different types of code smells in mobile apps [55][101], such as resource leaks, member ignoring methods (MIM), getters and setters etc. However, there is still more work that is required to

deeper understand performance issues (and its categories) during the development and maintenance of real-time Android apps.

Static code analysis tools have been largely used for identification of various code smells, including performance-related smells. Such tools use keywords based pattern matching to highlight the problematic statements, while others perform a more detailed and accurate analysis that includes data flow analysis. Regarding the static analysis tools *i.e., FindBugs*[2], and *PMD*[3] scan Java projects to fulfill some common goals, such as ensuring the overall correctness of the source code. However, there are few tools like Paprika [63], *PerfChecker* [83], *aDoctor* [92], and *Android Lint* [5] that are more specific to Android applications. However, there is still a wide scope to investigate the evolution of performance issues over the whole span of projects. These details may allow developers to detect some types of performance issues at early stage of development. In addition to this, further research is needed to analyze how the Android-specific performance-related issues are resolved and documented by developers in open source apps available on GitHub.

As discussed, the main purpose of many static code analyzers is to highlight the performance issues and provide some useful suggestions to manually resolve it. However, there may be few hardships to manually resolve performance issues, *e.g.,* (i) lack of knowledge to resolve them, (ii) time and effort to resolve them *e.g., HandlerLeak*, (iii) issues that do not affect the functionality of app, but can degrade the performance of app, effort has been already highlighted as a drawback of the current process to fix performance issues. A potential solution to these issues is an auto-refactoring tool, that can automatically resolve the performance issues. Building an automated refactoring tool to address performance issues is still an open problem.

1.3 Research Questions

The goal of this dissertation is to *analyze* the performance issues of Android apps with the *purpose* of investigating various aspects with *respect* to identification, evolution, and resolution of performance issues from the *viewpoint* of developers and researchers in the *context* of open-source Android apps.

To achieve the above goal in the context of free, open-source Android apps, we formulated the following three main (high-level) research questions in this dissertation.

[2]FindBugs - Find Bugs in Java Programs. `http://findbugs.sourceforge.net/`
[3]PMD - an extensible cross-language static code analyzer. `https://pmd.github.io/`

RQ1 *Which are the most recurrent types of performance-related issues observed in the developers commits for Android apps?*

Performance issues have been analyzed from various aspects (*i.e.,* from apps bug reports) but not well-explored through the commit messages. This research question focuses on analyzing the performance-related commits with the aim to identify different types of performance problems in Android apps. This research question will be answered in Chapter 4.

RQ2 *How do the performance issues identified by Android Lint evolve in Android apps?*

This research question emphasizes on the evolution of performance issues throughout the lifespan of Android apps by inquiring to what extent these types of issues are resolved and documented by developers. Chapter 5 shows the results for this research question.

RQ3 *Is it possible to automatically resolve statically-detectable performance issues in Android apps?*

This research question investigates the information obtained from RQ2 to automatically resolve such issues. Chapter 6 describes the proposed solution for this research question.

1.4 Research approach and method

In order to address the different flavor of research questions, we adopted various methods and approaches. Few of them are briefly explained below.

- **Mining Software Repositories (MSR)** is a software engineering field used to extract useful pattern and phenomenon from the data of software projects [24] such as issue tracking systems, communication lists (*e.g.,* email), bug tracking systems (*e.g.,* Bugzilla), version control repositories [16] etc. and also analyze the relationship between two variables. Various data mining techniques are usually anticipated for the data analysis purpose. The MSR field allows researchers and software practitioners to empirically evaluate the different artifacts of a software project, which will help them to understand various parameters of software evolution, and changes over time [70].

- **Card Sorting** is a qualitative method to organize and give a meaning to some unstructured or scattered information. Participants use cards to arrange different topic into categories and then label the each category accordingly. Generally, *Card Sorting* is applied in software engineering to evaluate some useful categories from

available data. There are two types of *Card Sorting i.e., Open Card Sorting* and *Closed Card Sorting*. In *Open Card Sorting,* participants are advised to analyze the data (content) and label it according to the best-described content. This technique is generally adopted when there are no such predefined groups available, whereas, in *Closed Card Sorting*, predefined set of categories are available. Participant analyzes the content and group them into given categories. [8].

- **Experiments and Quasi-experiments** The *experimental* study is done with the focus to reveal the causal relationship between different variables. For instance, to study the behavior of action X, we consider the impact of other factors or variables as constants. Generally, four factors should be formulated before conducting an experiment *i.e.,* (i) *Independent and Dependent Variables*, (ii) *Treatment and Hypothesis*, (iii) *Causality*, and (iv) *Matching and Randomization*. Using randomization, various assignments of treatments (or subjects) are analyzed while setting other variables as constant [6]. While in *Quasi-experiment*, researchers may not have control over all these four factors, and thus try to analyze the causal relationship. In simple words, the assignment of treatment of subjects is not based on randomization, rather evolve from the characteristics of subjects itself [119].

1.5 Contributions

The main contributions of this dissertation are provided below.

- A quantitative and qualitative study has been carried out with the aim of analyzing the performance-related commits in 180 Android apps. The main contributions of this study are twofold: the quantification of **"how much" developers are concerned about performance issues, and the taxonomy of concerns**, which can be used as a checklist by developers
- The extensive empirical study to investigate **how Android performance issues evolve during the life cycle of 316 apps detected by Android Lint**. The findings of this study presented five different evolution patterns by tracing the evolution history of seven types of performance issues. Furthermore, we manually analyzed the previously resolved documented commits, and we provided a catalog of solutions for each type of performance issue.
- To complement the above contribution, we **introduced an auto-refactoring Eclipse plugin with the aim of automatically resolving seven types of performance issues reported by Android Lint**. The proposed Eclipse plugin

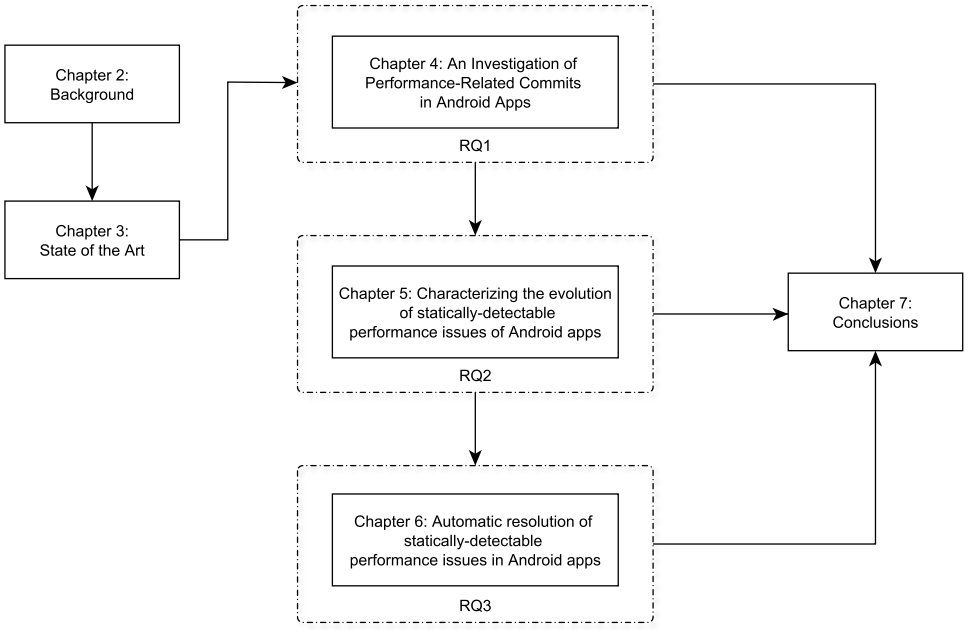


FIGURE 1.2: Structure of this dissertation.

is extended from *Leafactor* tool through implementing three new issues (rules) and slightly modifying two existing rules.

- To evaluate the proposed tool, **a survey-based study was performed to investigate the self-accessed auto-refactoring code from the developer's point of view**. The survey involves 21 Android developers and features two versions of code (*i.e.,* before and after refactoring) for the seven types of Android-specific performance issues. Developers' responses with respect to code execution and comprehensibility of refactored vs. non-refactored code versions.

- **A measurement-based study is dedicated to analyze the run-time performance of Android apps**. This preliminarily study analyse CPU utilization and memory consumption for the apps before and after the removal of the performance issues.

1.6 Structure of this dissertation

Fig. 1.2 summaries the overall structure of the thesis, which is organized as below.

- *Chapter 2* introduces the background, consisting of concepts and definitions used in this dissertation work such as Mobile apps, Android Programming Model, and static tools.

- *Chapter 3* presents the recent state-of-art work carried out in different areas related to the scope of this dissertation.

- *Chapter 4* provides the quantitative and qualitative study to analyze the performance problems from GitHub commit messages. The main types of performance problems are grouped from performance-related commits by applying the card sorting technique.

  Parts of this chapter are published in:

  Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. 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, 2016.

- *Chapter 5* reports an empirical study aimed at investigating the evolution of performance issues in 316 out of 724 analyzed Android apps i.e., when an issue is injected and resolved in the apps. A total of seven types of performance issues are considered in this study which is identified through static code analyzer Android Lint. Moreover, this chapter also provides the snippet of examples, where developers have resolved the performance issues and documented them.

  Parts of this chapter are published in:

  Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. Characterizing the evolution of statically-detectable performance issues of Android apps. Empirical Software Engineering (Journal), 2018.

- *Chapter 6* proposes an auto-refactoring Eclipse plugin, developed to resolve the performance-related issues of Android Lint. Moreover, this chapter also listed a survey-based study which is conducted to analyze the usefulness of proposed auto-refactoring plugin from the developers' perspective. In the last part of the chapter, a preliminarily measurement-based experimental study is discussed to measure the run-time performance impact (*i.e.,* CPU utilization and Memory consumption) in the context of Android apps.

- *Chapter 7* summarizes and concludes this dissertation. It also briefly discusses possible future research directions.

Chapter 2

Background

This chapter describes an overview of the key concepts and definitions related to the scope of research problems discussed in this dissertation. In section 2.1, we began with a brief explanation of the mobile app and its different types. Furthermore, the Android Programming Model is discussed in section 2.2. Finally, we conduct a recap of static analysis and refactoring tools for Android apps in section 2.3.

2.1 Mobile Applications (apps)

A mobile application (acronym used as mobile app) is a compact program developed to be deployed and executed on mobile phones and tablets. Mobile apps were initially designed in the past for very professional oriented tasks such as weather prediction, email, stock market etc. However, later on, as they started to attract a more diverse audience (kids, young and older people) in almost every domain of daily life such as entertainment, health care, banking, online shopping, social networking and many more. In a recent report [87], experts mentioned that the annual revenue generated from such apps is nearly 10 billion euros. The incremental growth in apps provided a positive impact on the job market by providing around 1.8 million jobs in at least 28 EU countries.

The apps are programmed to run on a particular mobile OS. (i.e. Android, Windows phone, iOS, Symbian, Blackberry) and the APK file (pre-installed version) for each mobile apps are published over their digital distribution platform *i.e.,* app market, which is an online store for the distribution of apps. Currently, there are mainly three app stores available which are popular and widely adopted among users; Google play store owned by Google, App Store operated by Apple and Microsoft Store (previously known as Windows Store) sponsored by Microsoft. As of 2017, Google play store[1] has published

[1]Google play. `https://en.wikipedia.org/wiki/GooglePlay`

over 3.5 million apps and cumulatively more than 82 billion of different copies of apps were downloaded until 2016. On the other hand, App store[2] has more than 2.1 million apps and altogether more than 130 billion of various copies have been downloaded until January 2017. In Microsoft Store[3], nearly 0.669 million apps were present until 2015. Generally, APK file of apps can be directly downloaded and installed from these digital distributed platforms into mobile phones. The two key terms review and rating are associated with the mobile apps, indicate the success of any app. In this dissertation, our research work is only focused on Android apps.

2.1.1 Types of mobile apps

The mobile apps can be mainly categorize into three types.

- **Native apps** are implemented to run on a specific mobile operating system. In other words, an app developed on one particular OS platform can not run on another operating system *e.g.,* Android native app will not work on iOS; instead, it will only work for its Android operating system. The key benefits of developing native apps are advanced graphics, pleasant user experience, and high performance. Moreover, developers use native device UI and have access to a larger variety of API's that does not constraint the app utilization. The problem with native apps is platform fragmentation. Since code written in one platform is not compatible with other platforms which leads to higher maintenance and testing costs, and consume more implementation time [26].

- **Hybrid apps** are developed via multi web-based technologies disguised in a native wrapper. Hybrid apps are comparatively more straightforward and quicker to implement. Since there is one code written for all platforms, which results in low maintenance cost and easier updating process. The major demerits of hybrid apps are; (i) slow in execution, (ii) less performance and optimization concerning native apps, and (iii) potential design problems may occur because an app coded for one platform does not need to look absolutely similar in another platform [26].

- **Web apps** are designed using web technologies and need a web browser to execute. Usually, they are programmed in CSS, HTML5, and JavaScript.The data related to web apps are stored in the database server. Therefore it consumes relatively less memory as compared with storage in smartphones. Moreover, to observe an enjoyable and user-friendly experience, it is mandatory to have a good Internet

[2]App store (ios). `https://en.wikipedia.org/wiki/AppStore(iOS)#Mostdownloadedapps`
[3]Microsoft store (digital). `https://en.wikipedia.org/wiki/MicrosoftStore(digital)`

connection because the data will be sent and retrieved from the server. The main disadvantage of web apps is that it would not be possible to use all the API's [26].

2.2 The Android Programming Model

Android is a Linux-based open-source operating system introduced by Google. Currently, it is among one of the most famous and highly used mobile app platform[17][4]. Mobile apps running on the Android platform are mostly developed using the Java programming language and are built via the Android Studio[4]. However, in some cases, developers use the Android Native Development Kit[5] (NDK) for implementing parts of their apps through native code, mostly in C and C++. NDK is often used when developers need to reuse libraries written in C or C++ (e.g., the OpenCV vision library[6]) or for processor-intensive tasks. An Android app is always built into a so-called Android PacKage (APK) which contains the compiled code, the used libraries, an XML-based manifest file providing essential metadata about the app (e.g., used permissions, unique identifier, supported Android APIs, etc.), and the local data and resource files that are required by the app at run-time. The APK of an Android app can be published in the official Google Play store to make it available and directly installable to Android-powered devices.

Android apps are composed of mainly four types of **components**: *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers* [1]. *Activities* can be performed on the single screen of the user's app interface and can be able to do; (i) reaction to user events (i.e., a touch on the screen) and (ii) updation to the user interface for providing information. An Android app can usually perform multiple activities to provide a cohesive user experience. For example, the app's screen showing the user to fill the personal details for reservation of flight ticket could be managed by a dedicated activity, whereas another activity may be in charge of managing the screen for showing the list of available flights for a given destination. *Services* are the components which run in the background to perform long-running operations *e.g.,* a service might play music in the background independently and may not dependent on the current tasks being executed in the app by the user. *Broadcast receivers* allow apps to react to events asynchronously. Broadcast receivers can receive events coming either from other components of the app or from other apps, sensors, or system services. As an example, if the user receives a message containing the geographical position of one of its contacts, then the messaging app can

[4] `https://developer.android.com/studio`
[5] `https://developer.android.com/ndk`
[6] `https://opencv.org/platforms/android/`

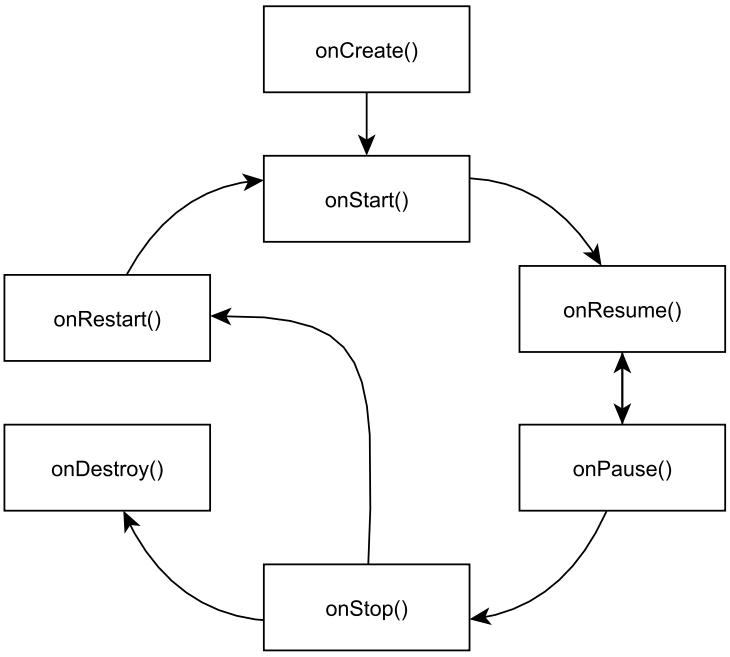


FIGURE 2.1: Life-cycle of an Activity.

open the corresponding mapping app on the user's device by launching an intent targeting the broadcast receiver of the mapping app. Finally, *Content providers* provide an abstraction layer for the data managed by the particular app. They can be queried and accessed by other components of the app (*e.g.,* performing CRUD operations on the data), provide standard data validation policies, and can make the data accessible to other apps (if needed) etc. *Content providers* offer a common API for accessing data independently by its internal persistence strategy. The data delivered by a content provider can be internally stored in a dedicated local database, file system, or in a server. For instance, in a messaging app, all the messages exchanged between the app users and its contents can be stored in a dedicated content provider and made available to all the other components of the app.

The communication among the various components within the Android application is carried out through the *Intents*. An *Intent* contains the short details of the operations or tasks that are going to be executed, and further, it can also be used to plea and invoke any specific actions from the other components of the app.

Furthermore, every application essentially follows some well-defined states which are called *lifecycle of an activity* [7]. Fig. 2.1 describes how an activity is created, utilized, and in the end, destroyed. Generally, the lifecycle of any activity is composed of seven different states, however, developers can have their logic to interpret the callbacks of each state. It can be shown in the Fig. 2.1, when an application is launched to create an

activity *oncreate()* state is called, whereas *onStart()* handler will be invoked to visible the activity to user, this is the time when application is prepared to enter in the foreground state. Activity will go to the foreground or Running State (the state where the user interacts with activity) when the *onResume()* is called. However, if any other activity comes to the foreground state, then the *onPause()* is called. Moreover *Onstop()* state is called when the activity is no more visible and *onRestart()* state is anticipate to make activity visible again from *Onstop()*. There may be cases when stopped, and paused activities can be forcefully killed due to immediate releasing of memory by the system to accommodate higher priority apps. In the last *onDestroy()* will call if the activity is finished or destroyed/killed by the system.

2.3 Static analysis and refactoring tools for Android apps

This section is composed of two sub-sections. Section 2.3.1 describes some of the available static analysis tools which are prominently used to detect performance issues. Whereas, section 2.3.2 is dedicated to explore various auto-refactoring tools which is utilized to auto-resolve various issues, including performance-related issues.

2.3.1 Static analysis tools

The fundamental purpose of static analysis tools is to ensure the quality of the software system by inspecting the code, without actually running it. The static analysis tools can be executed on different granularity level *i.e.,* on APK files, byte code, source code or any other artifacts to search for bugs. There are various reasons [69, 81] behind the massive popularity of static analysis tools in both industry and academia researchers. Moreover, many static analysis tools are conceived for Android apps and work on the different principals (techniques) including; keyword-based searching, control flow to data flow analysis, interval analysis and much more [81]. Among the available static analysis tools for Android platforms, many allow developers to highlight performance-related bugs. Some of them scan the source code of the project, some would analyze the metadata of the APK file, and while others would consider byte code as input with the aim to identify performance-related issues. In the following, we will recap some of the primarily adopted tools to find the performance-related issues in Android apps. Moreover, we classified them into two types (i) first, those which are used commonly in Java-related projects *i.e.,* general-purpose tools and (ii) others that are more precise to Android applications.

FindBugs[7] takes Java byte code as input, by analyzing, it can detect various code defects in Java projects. As it is an open-source static analysis tool, the configuration of *FindBugs* in the programming platforms is rather simpler such as in Eclipse IDE, and Netbean through ANT or maven settings. Moreover, it is smoothly executable from its official graphical user interface as well as from the command line. Since *FindBugs* is a general-purpose tool, so it can cover a wide range of code defects and also have separate categories for them such as correctness, maintainability, and performance issues etc. Each category is further divided into subcategories with a low, medium, or high severity.

PerfChecker is conceived to detect two kinds of performance-related anti-patterns in Android apps, *i.e.,* (i) lengthy operations in the main thread of a program, and (ii) violations of the viewholder pattern [83]. *PerfChecker* analyzes the Java byte code through the Soot framework[8], and throws a warning to the developer in case of such types of violations are found.

PMD[9] scans the Java projects in order to check for syntactic mistakes in the source code. Moreover, it is an open-source analyzer that can find bad programming habits and deficient code, which collectively lead to poor performance [20].

Paprika tool is capable to find the performance smells in open-source Android apps [63]. It receives APK file and its parsed metadata (such as app name and package) as input, generates a set of raw metrics related to Android programming model (also related to object-oriented (OO)). Initially, along with raw data (such as metrics, entities, and properties), the model is computed and then further, converted into a graphical model (stored in a database Neo4j). Then *Paprika* allows developers to detect different types of anti-patterns (*e.g.,* bob class, MIM [63] etc.) by applying respective queries to the database. The working of *Paprika* tool is dependent on the Soot[10], a popular framework for Java code optimization.

aDoctor is completely an automatic linter with good ability to identify 15 types of Android code defects [92]. Among various code smells detected, 7 are about the performance related issues such as *Inefficient Data Structure (IDS), Internal Getter and Setter (IGS) and Leaking Inner Class (LIC)* etc. Palomba *et al.* [92] transforms various rules into a tool (aDoctor), which are exactly described in the catalog proposed by Reimann *et al.* [101]. Furthermore, *aDoctor* utilizes the abstract syntax tree of the program and trigger the bug based on the rules similar to the one defined in [101].

[7] FindBugs - Find Bugs in Java Programs. `http://findbugs.sourceforge.net/`

[8] Soot - a framework for analyzing and transforming Java and android applications. `http://www.sable.mcgill.ca/soot/.`

[9] PMD - an extensible cross-language static code analyzer. `https://pmd.github.io/`

[10] Soot - a framework for analyzing and transforming Java and Android applications. `http://www.sable.mcgill.ca/soot/.`

Android Lint[11] is one of the most widely used among the Java and Android developers. It takes Java code file or XML file (or whole Java project) as input and highlights several types of issues[12]. *Android Lint* can be operated via command line as well as through IDE plugin (as it can be easily integrated with Eclipse and by default present in Android Studio). Like *FindBugs*, *Android Lint* has a dedicated category for each types of issues such as *accessibility, performance, security, and usability* etc.. Furthermore, each issue type has its subtypes, such as *DrawAllocation* is a sub-issue type of performance category deals with avoiding the allocation of objects during the draw operation inside *onDraw* method as described in Listing 2.1

```
DrawAllocation
--------------
Summary: Memory allocations within drawing code

You should avoid allocating objects during a drawing or layout operation.
These are called frequently, so a smooth UI can be interrupted by garbage collection
    pauses caused by the object allocations.

The way this is generally handled is to allocate the needed objects up front and to
    reuse them for each drawing operation.

Some methods allocate memory on your behalf (such as Bitmap.create) and these should be
    handled in the same way.
```

Listing 2.1: DrawAllocation description by Android Lint Tool

In this dissertation, we select Android Lint for multiple reasons; (i) it considers comparatively a higher number of frequent Android-specific performance issues, (ii) it has a separate category for performance-related issues in Android[13], (iii) it is integrated within Android Studio (*i.e.,* an official development platform for Android apps), being the standard de facto tool for Android apps, and (iv) it can smoothly run from the command line, effortlessly incorporated with other environments, and used into a broader set of software pipelines, similarly to those we implemented in chapter 5 (see Section 5.1.3).

2.3.2 Automated Refactoring Tools

There are few tools as well as plugins available to perform auto-refactoring. A summary of which is reported below for Java and Android apps.

[11] Android studio project site. `http://tools.android.com/tips/lint`
[12] Android lint checks. `http://tools.android.com/tips/lint-checks`
[13] At the time of writing only *FindBugs* also provides a dedicated category for performance-related issues, but it is not specific to Android apps.

Walkmod[14] is an open-source tool used to strengthen the code by automatically resolving the Java coding style issues of *CheckStyle*, *PMD* and *SonarQube*.

Facebook pf[15] consists of different APIs and tools to examine various aspects related to the source code. These aspects may include; code navigation, code visualization, static code analysis, or code-to-code transformation (*i.e.,* refactoring of source code). Pfff can provide support to various languages such as *C, PHP, Javascript, and Java*. It is also preliminary compatible with a few other languages such as *C++, C#, CSS, Rust, Haskell, Erlang, SQL, Python, Lisa, Html, and OPA*.

Kadabra tool[16] is used for transferring and instrumenting java-to-java source code. It works on the spoon which is an open-source library. This project is still under development phase.

AutoRefactor[17] is one of the popular open-source Eclipse plugins for automatically resolving the code smells in Java projects. The *AutoRefactor* allows developers to perform refactoring in many ways ranging from cleaning up the unnecessary code to replacing the piece of code with the more efficient version of the code etc. Furthermore, these set of rules can be applied individually (on a file) as well as collectively on a given input project.

Leafactor is an eclipse plugin, working on the principles of *AutoRefactor* tool. It can be conceived with an intention to auto-resolve some of the performance-related issues listed by Android Lint. *Leafactor* covers five rules for Java code base and one rule for XML resource files [45]. These rules includes *DrawAllocation*, *Recycle*, *ViewHolder*, *WakeLock* and *ObsoleteLayoutParam* (rule for XML). *Leafactor* tool takes a file or project as input then it looks for the above described five performance issues and automatically refactor them by transforming the performance smell code with a more optimized version of code.

In chapter 6, we extend the *Leafactor* plugin to provide an automatic solution for performance-related issues in Android apps.

[14] WalkMod. `http://walkmod.com/`

[15] Facebook pf. `https://github.com/facebookarchive/pfff`

[16] Kadabra. `http://specs.fe.up.pt/tools/kadabra/`

[17] AutoRefactor - Eclipse plugin to automatically refactor Java codebases. `https://github.com/JnRouvignac/AutoRefactor`

Chapter 3

State of the Art

In this chapter, we will discuss state of art studies on topics that are close to those analyzed. The literature studies related to performance issues in mobile apps is explained in section 3.1, evolution of statically-detectable issues are addressed in section 3.2, while studies using linters are discussed in section 3.3.

3.1 Performance issues in mobile apps

Many recent studies focused on the identification and characterization of performance issues. Liu *et al.* [83] performed an empirical study to investigate performance-related bugs in 8 well-known Android apps. They manually classified the 70 performance bugs into three main categories *i.e.,* GUI lagging, energy leak, and memory bloat. According to their findings, GUI lagging is the most frequent category with 75.7% bugs followed by energy leak with 14.3% and memory bloat with 11.4%. In the later part of their research, a static analysis tool—*PerfChecker*— was proposed to detect the two types of performance-related anti-patterns in Android apps.

Few efforts are also done to identify the performance issues from the developers point of view. Linares-Vsquez *et al.* [113] investigated the performance bottlenecks and the best practices to tackle them by interviewing 485 developers. The findings of their study show that developers mainly rely on static analysis tools for performance profiling as well as for debugging the Android apps. Regarding the best practices adopted, developers usually use multi-threading to resolve and improve performance bottlenecks.

One of the recent study conducted by Moura *et al.* [89], used the same approach as we did in Chapter 4. However, the context of their study is to analyze the energy-aware

commits whereas our Chapter 4 is more tilt towards analyzing the performance-related commits.

Over the years, various studies emphasis on identifying the performance issues via automated tools. For instance Nistor *et al.* [91] introduced a technique—*SunCat*—which can help developers to understand the performance issues by applying small common input and predict the potential issues for larger inputs. Whenever developer executes the application with common input then *SunCat* tool builds a list of recurrent patterns with some extra information to help developers in the future for larger inputs. To analyze the usefulness of their approach, Nistor *et al.* applied *SunCat* on the 29 different scenarios, which are taken from 5 Windows phone apps. Their results indicated that *SunCat* were able to find 9 performance issues.

Gomez *et al.* [54] proposed a context-aware approach named DUNE, intending to highlight UI performance regressions in previous releases of Android apps [54]. DUNE operated on two principles; (i) initially, it generates an ensemble model from the UI performance metrics obtained via previous test runs and (ii) Further, in next new test run, it will pinpoint the possible performance defects with deviations. To validate their approach, they conducted empirical study which involved three Android apps.

In many previous studies, performance smells are investigated in the context of energy aspects, such as bugs related to networking, sensors, non-sleep, and general [84, 95, 96, 116, 121]. All these studies are either conducted on app repositories or some forums. Where as Linarez *et al.* [112] investigate the API calls to detect the energy related code smells in 55 open-source Android apps by measuring through hardware power monitor. The findings of their study indicate that some design and implementation patterns can heavily impact the energy consumption in apps such as the usage of Model-View Controller, information hiding, or else development of the persistence layer by using a relational database. In chapter 4 of the thesis, we provided a quantitative and qualitative study on performance commits that resulted in different performance categories such as Local database, Networking, and load time etc.

Also, Cruz *et al.* [44] analyzed the impact of Lint based performance issues in the context of energy efficiency on six Android apps. The most important observation of their research work was that by resolving the few types of Android Lint based performance issues can save battery usage by one hour. The different types of performance issues associated with battery life improvements are; *DrawAllocation*, *ObsoleteLayoutParam*, *Recycle*, *ViewHolder*, and *WakeLock*. Cruz [44] introduced a tool named *Leafactor* aimed to auto-resolve various performance issues (mentioned above) to improve the energy consumption in Android apps.

Similarly, in another extensive study, Cruz *et al.* [46] presented a catalog highlighting 22 different design patterns that reduce the energy efficiency of apps. The study is based on the analysis of commits, issues, and pull requests of 1027 Android and 756 iOS apps. Results of their study highlight that the iOS developers would have less awareness regarding improvements of energy efficiency with respect to Android app developers.

Palomba *et al.* [93] conducted an extensive empirical study with the aim to analyze the impact of code smells on energy consumption of mobile apps. In particular, investigating on (i) how to identify energy-related code smells in the source code of mobile apps, and (ii) whether refactoring activities such as removing these smells can improve the energy efficiency. The study was carried out on nine method-level code smells specify for 60 Android apps. Results of their study showed that the four types of issues *i.e.,* Internal Setter, Leaking Thread, Member Ignoring Method, and Slow Loop methods consume 87 times more energy than the other types of code smells. Furthermore, refactoring of these code smells (*i.e.,* removing the code smell) can significantly improve the energy efficiency of Android apps.

Guo *et al.* [55] identified and characterized the resource leaks in Android apps. The presence of resource leaks can lead to app crash or performance degradation. Authors of this study proposed a lightweight tool named *Relda*, aimed to automatically detect the resource leaks by focusing on the application resource operations and help the developers by identifying the root causes of missing release operations.

Many past studies have been focused on adopting the profiling approaches to measure the performance optimization [98, 99] for mobile apps. For example, in [98], Qian *et al.* look into the interaction between the resource management layer and the application layer to detect inefficiencies in the usage of those resources that are mainly accountable for performance degradation in mobile apps.

Moreover, Habchi *et al.* [56] conducted an extensive interview-based study in order to understand the different aspects (*e.g.,* bottlenecks, limitation and best practices) of performance issues identified through Android Lint. Their study involved a total of 14 Android developers and adopted qualitative research methodology using Grounded Theory [31]. Along with other findings, their study shows that Android Lint can allow developers to take advantages in the form of learning the Android framework, quickly detecting performance issues and foresee the possible performance bottlenecks. Moreover, the study further revealed that how Android Lint is used *e.g.,* in some cases, it is used to fulfill the need of an individual development, in some cases, it is significantly used on a team level, and while in other cases it is purely exploited for the performance optimizations purposes. The study also reported some key obstacles that restrict the use of Android Lint such as the perspective that linters is not beneficial in terms of

performance, some organization follow reactive approach to deal with the performance aspect, and outcomes of analysis does not well presented to developers.

Hecht *et al.* [62] analyzed the real-time performance impact of three Android code smells on two open-source Android apps. They exploit *PAPRIKA* tool to detect the three Android code smells in their study *i.e.,* Getter/Setter, Member Ignoring Method, and HashMap Usage. Hecht *et al.* analyzed the four versions of each app independently as well as all of them. And then measure the UI and memory performance using a case scenario in order to analyze the four metrics *i.e., frame time, number of garbage collection calls, number of delayed frames, and memory usage.* Their findings show that by resolving these three Android code smells can improve the UI and memory performance. More precisely, they observed an up to 12.4% enhancement on UI matrices while resolving the Member Ignoring Method (MIM) smell and about 3.6% related to memory matrices while addressing all the three code smells.

To understand code smells in mobile apps, Palomba *et al.* [92] proposed a fully automated tool named aDoctor, identifies 15 Android-specific code smells, that are already described in the catalog presented by Reimann *et al.*. Further, to analyze the usefulness of the proposed tool, they conducted an empirical study which involves 18 Android applications. Result of their research indicated that aDoctor is promising in identifying the different code smells with nearly 98% of precision and 98% of recall.

Habchi *et al.* [58] conducted a large empirical study to examine survival life span of Android code smells. The study considered 8 different types of code smells in 255k commits of 324 Android apps along with history of 180k instances of code smells. The findings of their research suggest that about 75% of code smells are eliminated in the 34 commits on the host file. Moreover, such smells are resolved faster for the projects that are larger in terms of classes, developers and commits. Another interesting observation can be drawn that the code smells identified through Android Lint tend to remove faster than other types of issues, and thus have very little chances of survival. Furthermore, the granularity of code smells has greater impact on survival chances *i.e.,* code smell hosted by class are likely to survive higher than the inner classes and methods.

3.2 Evolution of statically-detectable issues

There are various studies that have been done in the past, which emphasize on the evolution of code smells identified by various tools. For example, *Paprika* has been used [61] for investigating the evolution of quality metrics of Android apps over time. The study involved seven different anti-patterns (4 were related to Android and 3 were

Object-Oriented) of PAPRIKA tools. The Android specific smells include; Member Ignoring Method, Leaking Inner Class, and UI Overdraw. These identified anti-patterns are exploit to observe the quality of the apps. The evaluation of their study is based on running the Paprika on different versions of 106 apps, gathered from Google Play Store. The baseline of software quality is measured through the complete set of considered apps. Further, in order to measure the quality of each versions of the app, they calculated the deviation with respect to baseline of the software system. The results of the study projected the relationship between various anti-patterns (*e.g.,* the complex class and blob anti-patterns tend to evolve together) and finally, they provided five different quality evolution patterns. Similar to this, in Chapter 5, we also present five different evolution patterns emerged from the investigating the lifetime history of 316 apps.

An empirical study is carried out by Di Penta *et al.* [97] focuses on the evolution of different vulnerabilities in the source code. In their study, three different tools namely Splint, Rats and Pixy were executed on three popular large scale networking apps (*i.e.,* Squid, Samba and Horde) with aim to analyze the evolution of various issues over time. In the Chapter 5 of our thesis, we performed similar kind of evolution study; however our context is related to performance issues (from Android Lint), and the object of the study is relatively much larger *i.e.,* 724 apps. Whereas, Di Penta *et al.* ran the tools on the lifetime of three projects and their study was mainly emphasized on the security-related vulnerabilities.

Another study investigating the evolution of four code smells over the whole span of two large scale open source applications (*i.e.,* JFlex and JFreeChart) is performed by Chatzigeorgiou *et al.* [40]. The issues considered in their study were Feature Envy problems, Long Methods, God class, and State Checking smells. The key outcomes of the study reflects that code smells tend to remain up to the latest releases of the app. Furthermore, survival analysis indicated that whenever the issues are injected, they survived (remain alive) quite long in the source code repository. And most noticeable result was that few resolutions were analyzed and the purpose of resolution was not intentional (*e.g.,* refactoring) but rather unintentional (*e.g.,* issue resolved while doing some other activities like implementing a feature or performing a change in some part of code.).

Tufano *et al.* [110] conducted an extensive study focusing on the survival analysis of bad code smells. Their study considered all the commits of 200 open source software systems extracted from the GitHub. The results reports in their study indicated that a majority of analyzed smells tend to remain alive in the system (*i.e.,* 80%)and only 9% of the issues were resolved with intention to perform the refactoring tasks. The main

difference with Chatzigeorgiou *et al.* [40] and Tufano *et al.* [110] work and what we did in Chapter 5 is we consider the performance-related issues instead of bad code smells.

A preliminary study targeted at investigating the time to fix a bug from its introduction was conducted by Kim, et al. [74]. Their primary focused was on the distribution times of bug fixing of two projects named as PostgreSQL and ArgoUML. Moreover, their results also reported the top bug fixing times.

Few past studies used survival model to perform the survival analysis. For example, Wedel *et al.* [117] used cox proportional hazard model to investigate the occurrence of bugs in Eclipse. Whereas Canfora *et al.* [39] also used same survival model but their context of the study is on the survival time of bug rather than occurrence of bug fixes. Various work have been done to analyze the distribution of occurrence of bugs. The results of studies [67],[74] indicated that main trend occurrence of bugs was related to system and they follow exponential and Weibull distributions. Whereas in the Chapter 5 of this thesis, we apply six different distributions on our identified issues with aim to investigate if any types of performance issue follow some specific type of distribution.

Some of such studies also focused on the automated detection of anti-patterns and their decays in continuous integration (CI). For instance, Vassallo *et al.* [115], initially surveyed 124 developers regarding CI anti-patterns. Then, they have proposed a tool named CI-ODOR, which detects the occurrence of four types of anti-patterns by analyzing the information of build logs and repositories. Moreover, they exploit the tool on 18,474 build logs of 36 widely used Java projects. The output indicated that there were 3,823 high-level warnings, which were distributed across all the considered projects. They validate their study by giving CI-reports to 13 original survey developers and through general feedback, 42 of them acknowledged the relevance of their reports.

To observe the refactoring activities in open-source projects, Vassallo *et al.* [114] performed a large-scale empirical study considering the change history of commits in 200 projects to investigate (i) developers' refactoring activities and their diffusion, (ii) the temporal context in which the refactoring operations are carried out, and (iii) the critical developer-related factors that can lead to the refactorings. The findings of their study showed that developers do not perform refactoring very frequently, but whenever they do perform, the most commons methods are; Rename Method, Rename Class, and Move Field. Moreover, the refactoring activities are generally carried out usually after at least one year from the start of the project or when the new version is about to release. Furthermore, enhancing features and bug fixing are the common developer-oriented factors, which leads to refactoring. And in the last developers tend to improve the design of the project when their workload is less than medium.

Mazuera-Rozo *et al.* [88] conducted a recent study aiming to investigate performance bugs in mobile apps. The study involves 500 commits extracted from 47 Android apps and 31 iOS apps (*i.e.,* 250 commits per each operating system). The main contributions of their study are (i) taxonomy of different types of performance bugs occurring in Android and iOS mobile apps, and (ii) survivability of such identified performance bugs *i.e.,* time taken (in days) from bug introduction to its resolution. The findings of their study showed that resource leak performance bugs are quite recurrent in Android apps (96 instances) followed by performance bad practices (47 instances). The resource leak performance bugs are broadly categorized into memory leak (61 instances) and suboptimal CPU usage (32 instances). Regarding the potential performance bugs affecting iOS apps are also related to resource leak (120 instances) with the majority of them related to memory leaks (110 instances). Moreover, on average, bug survives for 98 days, and it may increase to 178 days for the random-effects model, and up to 342 days for the max estimated survivability. Furthermore, performance bugs of type "resource leak" tend to survive longer with respect to other types of considered issues. Our study is different as we analyzed the performance issues in only Android apps with a much higher dataset of 457 commits distributed over 180 apps (Chapter 4). Also, we analyzed the survivability of 1314 performance issues of 316 Android apps (Chapter 5), which is much larger than their study. Whereas, their study presented a nicer taxonomy of performance bugs and survivability analysis of such issues in both Android and iOS mobile apps.

Various studies on API migration have been done in the past. For instance, Halrubaye *et al.* [33] discussed the effect of library API migration on software quality. They computed commonly used values of software quality both for before and after migration of 9 widely used APIs, which were occurred in the corpus of 57,447 open-source Java projects. The findings of their study revealed that the library API migration tend to improve the different aspects of software quality, such as increased cohesion, reduced coupling, and improved code readability. Moreover, they introduced an online portal for software developers, which can be used to understand the preliminarily effect of library migration on software quality and to provide the best design and implementation-related API migration examples that can improve the quality of software. Furthermore, they also offer a large dataset to the software engineering community with the aim to stimulate the library API migration research.

Lamothe *et al.* [76] investigated on the automatic API migrating techniques in practices. Their study analyzed the practical experience of the use of API migration in (Android) apps mined from the FDroid repository. For analysis, this study considered different documentation and historical code changes. The findings of this experience-based study showed that there are various challenges in the migration through historical code changes, and API documentation is significantly underestimated. The majority

of migration from depreciated API's to add a new API can be recommended through making a simple search in the documentation. Moreover, their practical experience suggested that API migration problems lie far away from the migration suggestions, *e.g.,* coping with parameter type varies in the latest Android APIs.

3.3 Studies using Linters

Studies using linters have been done in different contexts, for example, to investigate the usefulness of linters in dynamic programming, a study is conducted by Tmasdttir *et al.* [109]. Their study involved 15 developers to know the reasons behind why JavaScript developers practice ESLint [11] linter in OSS. The main findings of research shows that (i) lint is very helpful for enhancing the test suite, (ii) it motivates the newcomers to contribute, and (iii) it spares from wasting the time, which is spent onto talking about the code styles.

To investigate the importance of static code analysis, a survey and interview-based empirical study has been performed by Christakis *et al.* [41]. Among various important findings, they showed that performance-related issues are the second top potential concerns that demand frequent attention (from developers) to address. Moreover, performance issues are among the top four priority that developers looks for. Whereas, Johnson *et al.* [66] performed a study to analyze why developer do not consider static analysis tool for bug detection purpose. Their study reports the results obtained from interviewing 20 participants, which showed that a majority of participants were not denying the advantages of static tool analysis, but the only obstacles which restricted them not to adopt was representation of false positives and warnings.

Habchi *et al.* [56] presented a study aimed to understand the advantages and restrictions of detecting performance issues in Android apps using linters (Android Lint). This study was based on the interviews conducted from 14 experience developers. Their observations opened up different future research directions for developers, researchers, and tool creator communities. One of the benefits for developers to use linter was that it helps to tackle the performance issues, which were usually a tedious job for developers in terms of recognizing and resolving it. Also, linters can increase or promote the culture of performance within the team level. Further, their findings confirmed that the researcher community widely utilized a reactive approach to resolve performance-related issues [113]. They suggest it as a new future outlook direction for real-time comparison between reactive and proactive approaches. Regarding the findings for tool creator, linters should be more comprehensive and explicitly focused to different checks

of various categories. Furthermore, some additional details are required for developers, which may help them to resolve the issues such as nature and impact of identified issues.

Liu *et al.* [83] proposed a linter named *PerfChecker*, which is able to detect two types of performance issues in Android apps. Such code smells include (i) lengthy operations in the main thread of a program, and (ii) violations of the view holder pattern. By using the Soot framework, *PerfChecker* seeks byte code as input and generate a warning if any of above two violations is observed. Authors of the tool found 126 instance of these violations patterns when they evaluate this linter on 29 popular Android apps. Out of 126 violations, 68 were confirmed by developers as bugs (which were previously not known), and developers resolved further 20 violations.

One popular study focus on identifying the anti-patterns through a linter is conducted by Hecht *et al.* [63]. They introduce *Paprika*, which is capable to find various anti-patterns, including performance code smells in Android apps. Currently, *Paprika* can detect 3 Object-Oriented antipatterns and 4 Android-specific antipatterns. As discussed in section 3.2, in order to evaluate, *Paprika* has been ran on various versions of 106 Android apps, accumulated from the Google Play Store. Then, (i) for the baseline of software is calculated from the quality metrics of the considered apps, and (ii) for all the versions of each considered apps, the quality has been estimated using deviation with respect to the baseline. The outcome of their study shows the presence of co-relation between various anti-patterns (*e.g.,* the blob and complex class anti-patterns emerged together).And further, they proposed five main different evolution patterns, similar to what we identified in Section 5.3.

Another linter named *aDoctor* is introduced by Palomba *et al.* [92], which covers a wide range of Android-specific code smells (*i.e.,* 15 types). While writing this thesis, we observed 7 out 15 code smells are related to performance issues *i.e.,* Member Ignoring Method (MIM), Inefficient Data Structure (IDS), and Internal Getter and Setter (IGS) etc. The definition of the rules implemented in *aDoctor* is defined in a catalog by Reimann *et al.* [101]. Further, Palomba *et al.* conducted an empirical study involving the source code of 18 Android applications. The result shows that *aDoctor* can detect antipatterns with 98% of precision and recall.

FindBugs is one of the publicly available linter with dedicate performance bugs category and can be incorporated both using GUI as well as standalone (*i.e.,* from command line). Some past studies are done using *FindBugs* linter, for instance the study by Khalid *et al.* [72] investigate the relationship between *FindBugs* warnings and end-user ratings. They investigate the rating and associated review comments of 10,000 free Android apps from Google Play Store and analyze the corresponding warnings of apps obtained after executing *FindBugs* linter on source code. One of the outcome of their study suggest that

some categories of *FindBugs* warnings such as the Bad Practice, Internationalization, and Performance are quite significant in numbers.

There are many linters available, which is emphasis on auto resolving different types of code smells, including the performance-related issues [25] [19] [15]. Also, few studies on the linter has been done with aim to auto refactoring code smells including the performance issues. One of the recent study done by Cruz *et al.* [44], in which authors analyze the impact of performance-related warnings detected by *Android Lint* in the context of energy consumption in six Android apps. Cruz *et al.* introduce a tool named *Leafactor* (an extension of the *AutoRefactor* tool), which is able to auto-resolve five Lint based performance warnings *i.e., ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParam*, and *Recycle*. These performance issues are implemented with purpose of battery saving in the Android apps. The main findings of their study shows that by resolving a set of performance issues can save up to one hour of battery life of the mobile phones.

Chapter 4

An Investigation of Performance-Related Commits in Android Apps

This chapter reports a preliminarily study aimed at conducting a quantitative and qualitative characterization of performance-related commits for Android apps. First—and similarly to what previously done in a work on energy-related commits [89]—we identify, using regular expressions, commits explicitly referring to performance-related issues. In other words, instead of using static source code analysis—as done by Liu *et al.* [83], or dynamic analysis—which would require appropriate execution profiles and it is not practicable on large-scale—we rely on documented performance related changes, as previously done by Ray *et al.* [100] for the analysis of bug categories on GitHub.

We report the distribution of such commits, also analyzing how do they vary across app categories. Then, using the card sorting approach, we produce a taxonomy of performance related concerns, and qualitatively describe some examples of commits belonging to these concerns. With respect to work such as the one of Liu *et al.* [83], our study has been conducted in the large, featuring the analysis of commits from 2,443 open source Android apps. Of such commits, 457 of them, belonging to 180 apps, turned out to be documented, performance-related commits. They generally affected any kinds of apps, although categories in which user experience was very important—*e.g.,* health and fitness, or photography—were slightly more affected than others. The card sorting categorization revealed how the most frequent kinds of performance-related commits were about GUI-related changes, (performance-affecting) code smells removal, and network or memory-related problems.

In summary, the main **contributions** of this chapter can be summarized as: (i) an investigation on performance-related commits in 180 open source Android apps; (ii) a taxonomy of the main kinds of performance-related problems, obtained by applying card sorting [107]; and (iii) a replication package, featuring a dataset of categorized performance-related commits[1].

4.1 Study Design

The *goal* of this study is to investigate performance-related commits in Android apps, with the *purpose* of understanding their nature and their relationship with projects' characteristics, such as project domain or size. The study *context* consists of 2,443 open-source apps and their evolution history. The study aims at addressing the first high-level research question (RQ1) of this dissertation (described in section 1.3): To address this, we formulate the following sub-research questions.

- RQ1.1: To what *extent* developers consider performance issues of Android apps?
- RQ1.2: What are the *concerns* that developers have when dealing with performance issues of Android apps?

More specifically, RQ1.1 aims at assessing the frequency in which app developers consider performance issues of the app, whereas RQ1.2 aims at classifying the specific concerns that developers have when considering the performance issues of the app being developed (*e.g.,* fast access to file system, reactivity of the user interface, etc.)

The **context** of our study consists of a set of open-source Android apps distributed in the Google Play store. We decided to analyze mobile apps in the Google Play Store because of its large market share in terms of both distributed apps and sold smartphones with respect to other platforms such as Apple iOS, Windows Phone, BlackBerry [47, 79]. Since we are targeting mobile apps that have been designed and developed as real projects with real users and we also aim at accessing the performance concerns managed by their developers, the **objects** of our study are Android apps that (i) are freely distributed in the Google Play Store; and (ii) have their versioning history hosted on GitHub.

Fig. 4.1 presents the process we followed for identifying our target population, together with the number of apps considered at each step. Basically, (i) we mined the well-known FDroid open source apps repository for extracting all those Android apps in which the description page contains both a link to a GitHub repository and a link to a Google Play

[1] `https://github.com/teerath91/ReplicationPackageICSME2016`

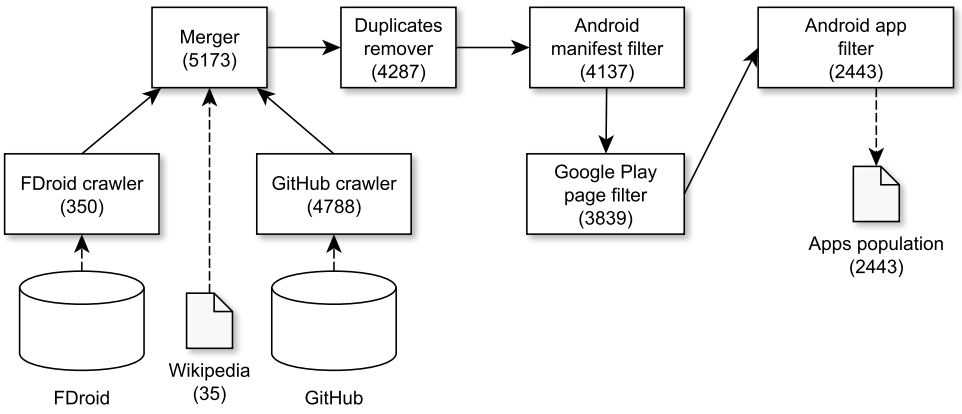


FIGURE 4.1: Apps identification process.

page; (ii) we performed a custom search on GitHub by targeting all the repositories in which the `readme.md` file contains a link to a Google Play page; and (iii) we collected all the apps enlisted in the community-maintained list of free and open-source Android apps on Wikipedia[2]. After a merging and duplicates removal activities we obtained 4,287 mobile apps. At this point we filtered out (i) all those apps whose GitHub repository does not contain an Android manifest file as they clearly do not refer to real Android apps, (ii) all those apps for which the corresponding Google Play page is not existing anymore (*i.e.,* they have been removed from the store for some reason), and (iii) all those apps in which the Android file is not in the root directory of an Android app (those cases happen when the manifest file actually refers to an Android library, to the binaries of some other app, etc.). The final population resulting from this process is a set of 2,443 mobile apps, each of them represented by its GitHub and Google Play identifiers.

The variables considered to address RQ1.1 are the (i) *pCommits*, the number of performance -related commits in the GitHub repository of the app, as compared to the overall number of *commits*, and (ii) the app *category* on Google Play. As for RQ1.2, we considered the different kinds of performance *concerns* being dealt in performance-related commits.

We extracted the *commits* and *pCommits* using a script that considers only the folder containing the source code and resources of the mobile app, excluding backend, documentation, test, and mockups. In each repository we identified the folder of the app by the presence of an Android manifest file. The mining script identifies a commit as performance-related if it matches at least one of the following keywords: *wait, slow, fast, lag, tim, minor, stuck, instant, respons, react, speed, latenc, perform, throughput, hang, memory, leak*. Those keywords have been identified by considering, analyzing,

[2] `http://wikipedia.org/wiki/List_of_free_and_open-source_Android_applications`

and combining mining strategies from previous empirical studies on software performance existing in literature (both mobile and not mobile-specific) [65, 85, 104, 120]. The mining script considers all the possible combinations of both lower and upper cases of each keyword. By applying pattern matching, we identified a set of 535 candidate performance-related commits. Such commits were manually analyzed, and 78 of them were identified as false positives. This produced a final set of 457 performance-related commits.

We extracted the *category* variable by mining the web page of the Google Play store of each app. Then, we identified the *concerns* by applying the open card sorting technique [107] to categorize performance-related commits into relevant groups; we performed card sorting in two phases: in the first phase we tag each commit with its representative keywords (*e.g., read from file system*, *swipe lag*) while in the second phase we group commits into meaningful groups with a descriptive title (*e.g., UI issues*, *file system issues*). To minimize bias, this activity has been performed by two researchers and the results have been checked by a third researcher; this activity resulted in a set of 10 categories (see Section 4.2).

4.2 Results

In this section we answer each research question by presenting the results of the study described in Section 4.1

4.2.1 RQ1.1 - To what extent developers consider performance issues of Android apps?

To answer this question, we count the frequency of performance-related commits (see the *pCommits* variable) with respect to all the commits of our dataset. Firstly, we can observe that 7.5% of the apps in our dataset have at least one performance-related commit (180 distinct apps over a total of 2,443). Among them, we identified 457 performance-related commits for this study.

Table 4.1 shows the distribution apps, and performance-related commits across categories (percentage represents the ratio of *pCommits* over *Commits* in each app category). Performance-related commits are more frequent in some categories [50]; for example, the category with the highest percentage of performance-related commits is *Comics* (2.31%), that primarily contains apps with an immersive user experience and long usage sessions, followed by the *Customization* (1.58%) and the *Weather* (1.48%)

TABLE 4.1: Distribution of performance-related commits across categories

Category	**#Apps**	**#pCommits**
Comics	4	5 (2.31%)
Customization	85	18 (1.58%)
Weather	21	15 (1.48%)
Health and Fitness	69	14 (1.21%)
Photography	36	67 (1.21%)
Tools	573	97 (0.85%)
News & Magazines	43	36 (0.83%)
Communication	91	21 (0.80%)
Productivity	216	28 (0.71%)
Games	350	39 (0.68%)
Shopping	13	3 (0.66%)
Libraries & Demo	70	4 (0.63%)
Travel & Local	71	16 (0.55%)
Media & Video	49	7 (0.52%)
Music and audio	62	6 (0.49%)
Social	69	37 (0.47%)
Medicine	8	3 (0.43%)
Finance	57	4 (0.36%)
Business	35	4 (0.34%)
Education	200	18 (0.33%)
Entertainment	128	8 (0.26%)
Transportation	70	4 (0.21%)
Books & Reference	41	2 (0.12%)
Lifestyle	82	1 (0.07%)

TABLE 4.2: Apps with the highest number of performance-related commits

Google Play ID (GitHub)	**Category**	**#pCommits**
com.almalence.opencam (almalence/OpenCamera)	Photography	31 (8.7%)
com.gopro.smarty (M66B/XPrivacy)	Photography	24 (2.5%)
com.newsblur (samuelclay/NewsBlur)	News	18 (13.56%)
ca.cumulonimbus.barometernetwork (Cbsoftware/pressureNET)	Weather	14 (7.27%)
org.wordpress.android (wordpress-mobile/WordPress-Android)	Social	13 (4.75%)
net.usikkert.kouchat.android (blurpy/kouchat-android)	Communication	11 (4.72%)
com.pacoapp.paco (google/paco)	Health	10 (13.20%)
com.eleybourn.bookcatalogue (eleybourn/Book-Catalogue)	Productivity	10 (1.9%)
org.qii.weiciyuan (makings/mst)	Social	9 (4.57%)
org.quantumbadger.redreader (bamptonm/RedReader)	News	9 (8.87%)

categories, that primarily contain utility, task-based apps with very short usage sessions, etc. This observation may be an indication that performance issues are somehow orthogonal across apps, independently of their specific application context and user experience requirements.

Nevertheless, it is interesting to note that in our dataset there are some apps with a relatively high number of performance-related commits, the top-10 being listed in Table 4.2. Note that we considered the absolute number of performance-related commits

instead of percentages (also reported), as we want to focus on the absolute number of performance-related tasks. As one can notice, these apps are not tied to some very specific categories. We manually analyzed the commit messages of the app with the highest number of performance-related commits (*i.e.,* com.almalence.opencam), it has 17 commits concerning memory consumption, 9 commits about user interface responsiveness, 4 commits about images optimization, and other mixed types of commits; being it a photography app, the nature of those commits is aligned with the features provided by the app. We also manually analyzed all the other top-10 apps and we found that the types of their performance-related commits vary without exhibiting any specific pattern.

4.2.2 RQ1.2 - What are the concerns that developers have when dealing with performance issues of Android apps?

Table 4.3 shows the 10 categories resulting from card sorting, together with an example of representative commit messages for each category (we randomly took it from our dataset), the frequency and the percentage of commits belonging to each category. It is important to note that the sum of all frequencies (586) is higher than the total number of identified commits because each commit message can belong to more than one category. More specifically, there are 422 commits with 1 category, 62 commits with 2 categories, and 2 commits with 3 categories. In general, *Android developers primarily focus on addressing one performance-related concern at a time (422 times over 486), rather than addressing more than one in combination (64 times over 486).* In the following we discuss each identified category of concerns. All together, those categories give an indication about what are the main concerns that developers perceive, consider, and address when dealing with mobile apps performance.

The most frequent concern that developers have when dealing with performance issues of Android apps is the responsiveness of the *user interface*, *e.g.,* in terms of swipe lags, screen layout drawing, lists scrolling responsiveness (27.35% of commits). This is an indication of the fact that developers know that end users perception of app performance is of paramount importance and that end users just expect mobile apps to properly work (*e.g.,* without delays, with few bugs, with a natural user experience), independently of the implemented technical solutions, used tools or libraries [50]. Examples of UI concerns include: use of Android's recycler views instead of plain list views[3], render images in slices, prefer Android's asynchronous tasks.

Another recurrent target of app developers is to fix existing performance-related *code smells* (22.53% of commits), *i.e.,* symptoms of poor design and implementation choices,

[3]https://developer.android.com/training/material/lists-cards.html

TABLE 4.3: Categories of identified concerns.

Category	Representative commit message	#pCommits
User interface	▲ *FIX layouts for better rendering performance*	125 (27.35%)
Code smells	▲ *fixed String concatenation performance issue*	103 (22.53%)
Generic concerns	▲ *Performance and error handling improvements*	99 (21.66%)
Networking	▲ *Use a socket connection when scanning for hosts instead of isReachable. Set performance options to prefer fast connection. Enable TCP_NODELAY*	59 (12.91%)
Memory	▲ *Fixed major memory leak; should improve responsiveness on older devices*	50 (10.94%)
Loading time	▲ *Made initial load WAY faster*	34 (7.43%)
Images	▲ *Draw all static objects on one image in android to optimize performance*	20 (4.37%)
Local database	▲ *Added indexes for posts.postid and posts.blogID to improve performance of several lookups*	12 (2.62%)
File system	▲ *Separate file loading and vault initialization to enhance performance*	10 (2.18%)
Sensors	▲ *Performance fix: Closing GPS service as soon as lat/long has been determined.*	5 (1.09%)

mainly due to time constraints of the project [52]. Examples of fixed code smells include inefficient usage of regular expressions, recurrent computations of constant data, usage of deprecated decryption algorithms.

In a relatively large number of commits (21.66%) developers mention only *generic concerns* about app performance, without detailing what the problem is and how it has been potentially fixed. This is a clear behavioral anti-pattern because generic commit messages make very difficult to know the nature of a performed change (*e.g.,* it may fix a bug, implement a new feature, improve code quality, etc.), its effects, to find when a bug has been introduced, etc.

Networking is one of the area in which performance-related commits occur a lot in mobile apps. For example, making HTTP requests is one of the most energy consuming operations in Android [80]. Our analysis shows that app developers take special care of networking operations (12.91% of commits) in their apps and refine their code in order to mitigate the impact of networking operations on the overall performance of the app. Examples of implemented solutions include: reduce as much as possible the frequency of calls when doing long polling, avoid making multiple requests in parallel, check the status of the connection before making a request to a server.

Keeping small the *memory* footprint of a mobile app is one of the key solutions for

improving its performance[4], specially for low-end devices. Developers are aware of this interaction between memory usage and performance (10.94% of commits) and apply solutions like stopping auxiliary services when available memory gets low, avoiding to load potentially unused data, etc.

Loading time of app screens is of paramount importance for its success, specially when considering the startup screen of the app. Developers are focusing on this key aspect of their apps (7.43% of commits) and apply solutions like reducing the information to be parsed when starting an activity, caching methods requiring a restart, etc.

Concerns with a frequency lower than 5% are: *images* management (4.37%), interaction with *local databases* (2.62%), *file system* access (2.18%), and interaction with device *sensors* (1.09%). Those concerns have been less targeted by app developers; nevertheless, they are representative examples of potentially relevant aspects to verify when considering the performance of an Android app. Examples of implemented solutions include:

- Images: load images directly in the required size, render images one at a time;
- Local databases: perform queries in an asynchronous task, add indexes to specific fields;
- File system: separate file loading from other activities, use buffered streams for file decryption;
- Sensors: filter sensor data, limit the use of the GPS service.

4.3 Threats to Validity

Threats to *construct validity* are mainly related to the use of pattern matching to identify performance-related commits. Under this perspective, we are assuming that if a commit message contains specific keywords is describing a change in the source code of the app related to its performance. Therefore, we are aware that such approach may miss undocumented performance-related commits. False positives have been instead avoided by performing a manual analysis of the identified commits.

Reliability validity threats concern the possibility of replicating this study. We mitigated this possible threat by making the replication package with all extracted data, mining, and analysis scripts available to interested researchers.

[4]https://developer.android.com/training/best-performance.html

Threats to *external validity* mainly concern the generalization of our results that relate to the representativeness of the apps considered in this work. We reduced this threat by considering a relatively large data set (*i.e.,* 2,443 apps) and by selecting apps that have been developed in the context of real projects (*i.e.,* all selected apps have been distributed in the Google Play store and available to the public). Another potential threat to external validity is that fact that we consider only freely available apps; this is an acceptable bias because free apps represent more than 75% of all Google Play store apps and they are downloaded more often [53]. Also, we analyzed only commit messages written in the English language; this potential bias can be considered as acceptable as English is the dominant language used by Android developers.

4.4 Conclusion

This chapter reported a preliminary study aimed at identifying documented performance-related commits in Android apps. The proposed investigation is carried out to answer the first high-level research question of this dissertation (mentioned in section 1.3), *i.e.,*

RQ1 - *Which are the most recurrent types of performance-related issues observed in the developers commits for Android apps?*

By analyzing commits of 2,443 apps, we discovered a total of 457 performance-related commits spread across 180 apps. We performed a qualitative analysis of such commits using card sorting, and identified a total of 10 commit categories. Overall, performance commits mainly related to issues found in the app user interfaces. Other than that, we also found frequent commits aimed at removing some code bad smells or at improving some part of the app logic and, finally, dealing with lags in the networking connection. Last, but not least, we found improvements related to I/O towards file system or databases, and related to the access to sensors.

As a preliminary result, the categories we identified can be used as a checklist by developers in order to see if they are considering all major performance-related aspects of their mobile app.

Chapter 5

Characterizing the evolution of statically-detectable performance issues of Android apps

The goal of this chapter is to empirically investigate the evolution of (potential) performance issues reported by static analysis tools and the extent to which such issues are actually resolved by developers. More specifically, the study analyzes the occurrence and resolution of seven kinds of performance issues identified by Android Lint in 316 open source Android apps hosted on GitHub (among 724 apps we analyzed in total). We have chosen Android Lint as static analyzer, because it is integrated in Android Studio and also available as an Eclipse plugin, therefore it is likely to be used by many Android developers.

First, we identify and report the occurrence of likely performance issues across the analyzed apps. Then, we trace the detected performance issues across the apps' evolution history, and determine whether they have been detected and to what extent they survive in the app. The study has the purpose of determining whether there are some kinds of issues that tend to be resolved quickly whereas others are more likely to be ignored by developers. Finally, we analyze the extent to which the resolution of the detected performance issues is also acknowledged by developers and documented in commit messages.

So far, performance issues have been investigated in Web applications [32], heterogeneous environments [51], or large-scale applications [86]. Also, Zaman *et al.* conducted a qualitative study of performance bugs [120]. To the best of our knowledge, the most recent work concerning performance bugs of mobile apps is the one by Liu *et al.* [83]

who analyses 70 real bugs, in which GUI lagging (53/70) found to be more recurrent followed by energy leaks (10/70), and memory bloat (8/70).

5.1 Study Design

This study has been carried out by following the guidelines for designing, conducting, and reporting empirical experiments in software engineering [105, 119]. In this section we focus on the design of our study, specifically we present its goal and research questions (Section 5.1.1), context selection (Section 5.1.2) and data extraction (Section 5.1.3).

A complete *replication package* is publicly available[1] to allow researchers to independently replicate and verify our study. The replication package includes the Python and shell scripts for identifying the targeted Android apps, the list of all considered GitHub repositories, the mined Google Play metadata, the raw data extracted from each GitHub repository, the Python and Shell scripts for extracting the raw data, and the R scripts we developed for data exploration, analysis, and visualization.

5.1.1 Goal and Research Questions

We formulate the goal of this study by using the Goal-Question-Metric perspectives [37]. Table 5.1 shows the result of our goal formulation.

TABLE 5.1: Goal of this study

Analyze	the change history of Android mobile applications
for the purpose of	characterizing their evolution
with respect to	statically detectable performance issues
from the viewpoint of	developers and researchers
in the context of	open source Android applications

In the following, we present and discuss the research questions we derived from the above mentioned overall goal in order to answer the second high-level (main) research question (RQ2.2) of this dissertation (as described in section 1.3).

> RQ2.0 – To what extent does Android Lint identify performance issues in the analyzed apps?

[1] `https://github.com/S2-group/AndroidPerformanceIssues`

As already introduced, in this study we leverage Android Lint for identifying performance-related issues of Android apps. This research question is exploratory in nature and aims at characterizing the number, frequency, and distribution of performance issues identified by Android Lint across the versioning history of all apps. By answering RQ2.0, we assess whether the context of our study (*i.e.,* the apps dataset we built – see Section 5.1.2) and Android Lint provide enough data points for answering the remaining research questions. Moreover, by answering RQ2.0 we identify the most recurrent statically-detectable performance issues during the evolution of Android apps, providing empirical evidence to developers and researchers for getting a better understanding of Android-specific performance issues.

RQ2.1 – How does the number of statically-detectable performance issues of Android apps vary over time?

The main objective of RQ2.1 is to investigate whether the evolution of statically-detectable performance issues across different Android apps exhibits identifiable patterns. The identified patterns can be used by researchers as a foundation for investigating the relationships between apps exhibiting the same or different patterns. Also, the emerging patterns can guide developers in identifying potentially dangerous patterns in their own apps, *e.g.,* a sudden increase of performance issues without any subsequent decrease.

RQ2.2 – Which types of statically-detectable performance issues tend to remain in Android apps across their lifetime?

By answering RQ2.2, we provide insights about how each type of Android performance issues tend to remain in Android apps over time. A long survival time of a performance issue can have two completely different explanations. On the one hand, it can indicate that the issue does not seriously affect the app's performance, and therefore it has been ignored by developers. On the contrary, if it is a harmful issue, remaining in the app for a long time would mean potentially affecting multiple releases.

RQ2.3 – What is the lifetime of statically-detectable performance issues of Android apps?

With the term lifetime we mean the interval between the introduction and the resolution of a performance issue along the versioning history of the app. The underlying intuition behind RQ2.3 is that different types of statically-detectable performance issues have significantly different life spans. The results of RQ2.3 can highlight whether particular kinds of performance issues tend to be resolved quicker than others, either because they are easier to spot, or because they are deemed to be more dangerous. Furthermore, we also characterize whether the lifetimes of different types of performance issues follow known probability distributions. This can help developers in knowing how likely a specific performance issue in their app will be resolved from the code base. Moreover, having this information will help developers in understanding how likely other app developers are fixing the issues and which ones are considered and ignored.

RQ2.4 – To what extent the resolution of statically-detectable performance issues of Android apps have been documented by developers?

This research question aims at (i) assessing whether developers document in their commit messages the resolution of performance issues and if yes, how many such resolutions commits were considered for each type of performance issues) and (ii) providing a minimal catalog of representative solutions, one for each type of Android performance issue. The results of RQ2.4 provide empirical evidence about whether developers consciously document their activities related to the resolution of performance issues. Researchers can use such evidence as a foundation for further studies on the relationship between documented and not-documented activities related to statically-detectable performance issues of Android apps. Finally, developers can use the catalog of solutions for standing on other developers' shoulders and use it as a reference for solving the issues raised by their instance of *Android Lint*.

The research questions discussed above drive the whole study, ranging from the selection of *Android Lint* as analysis tool, to the activities related to apps selection, data extraction, and analysis.

5.1.2 Context Selection

This study focuses on *real-world Android apps* for which we can execute the Android lint analysis tool across different *versions* of their source code. More specifically, the context of this study consists of a set of Android apps that (i) have their versioning history hosted on GitHub and (ii) are distributed in the Google Play store. We chose GitHub as target of source code repositories because (i) it is extremely popular among

developers (as of June 2018, it has a community of 24 million developers[2]), (ii) it hosts a huge amount of metadata that can be accessed through its API[3], and (iii) there is a variety of available tools for mining and processing data and metrics from GitHub repositories (*e.g.,* the `git log` and `git diff` tools). We focus on apps distributed on the Google Play Store because it is the official distribution channel of Android apps.

In the remainder of this section, a detailed overview of the process for building the dataset of Android apps is given. Identifying the required target dataset of Android apps for this research requires applying several filtering steps, which are documented alongside the respective numbers of apps resulting from each filtering step. The dataset building process of this study is similar to the one proposed in [48] and its 10 steps are shown in Fig. 5.1. The initial collection of those apps originates from three different sources, namely: FDroid, GitHub, and Wikipedia. The reason why we chose them is because we wanted to achieve a diverse set of sources, including the most popular host for open source (GitHub), a store of open source Android apps (FDroid), and finally an online-compiled catalog (Wikipedia, which was included for the sake of completeness, because as explained below it contributed with a fairly limited number of apps).

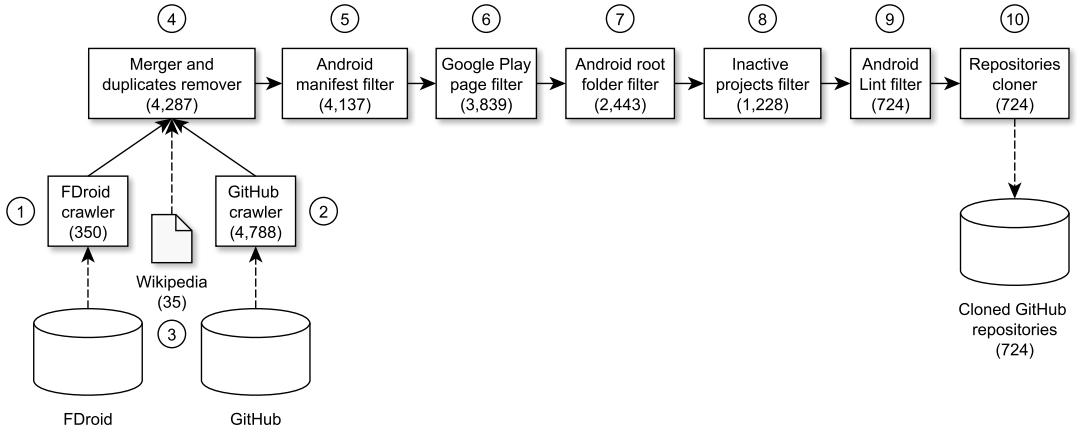


FIGURE 5.1: The dataset creation process.

The first source for our dataset is FDroid [29], a well-known online catalog of free and open-source Android projects (step ①). From this catalog, a search is applied that locates apps that contain: a) a link to the respective GitHub repository, and b) a link to the respective Google Play store page. Mining the FDroid repository resulted in 350 potentially relevant GitHub repositories[4].

[2] `https://github.com/features`

[3] `https://developer.github.com/v3/`

[4] At the time of conducting this study (June 2018) the search functionality on FDroid appears to be broken or not working. Furthermore, the `https://f-droid.org/forums/search/` endpoint that was used in the mining script does not exist anymore.

From Github (step ②), a custom search targeting all the repositories containing a link to a Google Play Store app page in their readme files is performed. In order to do not occur into the GitHub limit of 1,000 results per search, we stratify our search queries by date range so that each search results in less than 1,000 results. The whole set of considered dates ranges from the creation of GitHub (*i.e.,* Jan 1, 2001) to the day in which the searches were performed (*i.e.,* Feb 15, 2016). This search resulted in a total of 4,788 potentially relevant GitHub repositories.

The third source for our dataset is a Wikipedia [30] page containing a maintained list of free and open-source Android apps (step ③). We manually screened this list of apps to select the ones that, again, contain a link to the respective GitHub repository and are published on the Google Play Store. This step results in a total of 35 potentially relevant GitHub repositories.

In step ④, each repository coming from three data sources is uniquely identified by its < *repository owner* , *repository name* > pair and all duplicates are merged. The execution of this step results in a total of 4,287 unique GitHub repositories.

In the next filtering step (step ⑤), we identify those repositories which potentially contain the source code of an Android app. This filtering step is done by considering only the repositories containing the mandatory `Android Manifest.xml` file. Indeed, as mentioned on the official Android developers website[5], an Android app shall always come with an Android manifest file. The manifest of an Android app contains all the essential information supplied to the Android system, allowing it to run the actual app. For example, the manifest file is in charge of naming the Java package for the app, which serves as an unique identifier of each app. Information contained in the Android manifest file includes the main components of the app, its required permissions, the minimum required API level for the app, the third-party libraries used by the app, etc.

In step ⑥ we filter out all those repositories containing apps that are not published in the Google Play store. In the context of this process, this may occur if (i) the considered GitHub repository is a simple demo or toy example, which has been developed for personal or internal usage only, (ii) developers removed the app from the Google Play store, or (iii) Google deliberately took down the app because it was violating some of its distribution policies. This step has been done by extracting the package name of the app from its manifest file, and performing a network request to the URL where the Google Play page of the app should be present[6].

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

[6] This check is sound since the web page of an app in the Google Play store follows a fixed pattern, *i.e.,* `https://play.google.com/store/apps/details?id=[app package name]`.

Step ⑦ involves the identification of the app's root folder containing its source code. The rationale for this step is that the folder containing the Android manifest file should also contain the complete source code for the app. Indeed, in this step we need to exclude those repositories where the manifest file actually refers to an Android library, to the binaries of some other apps, etc. This step is realized by checking if the folder containing the Android manifest file follows the structure mandated by the Android platform[7].

Step ⑧ involves the identification and filtering of inactive GitHub repositories. Indeed, it is well known that mining GitHub repositories brings the risk of considering inactive or unmaintained repositories, thus adding noise to the results of the study[71]. For each repository, we extract (i) the *app development lifetime* and (ii) the *number of commits*. The app development lifetime is defined as the range between the first and last commits, whereas the number of commits is defined as the count of all the commits performed in the repository. In order to avoid inactive or unmaintained repositories [71], we considered only the apps having a lifetime span of at least 4 weeks and with at least 10 commits.

Step ⑨ involves the filtering of all those apps which cannot be properly analyzed by the *Android Lint* tool. It is important to note that Android Lint requires that the app under analysis is fully built. Building arbitrary software mined from third-party GitHub repositories is notoriously difficult, mainly due to missing dependencies (*e.g.,* the server hosting a dependency is no longer reachable), Java compilation errors (*e.g.,* undefined symbols, missing packages), and project-specific build commands (*e.g.,* non-default Gradle tasks) [59, 108]. In this step we managed to cover many recurrent non-standard cases by iteratively running the *Android Lint* tool on all repositories and (i) manually refining its configuration and (ii) adding preprocessing steps for making the app and its Gradle configuration more Lint-friendly (see step 2 in Section 5.1.3). Excluded repositories include apps with very peculiar Gradle configurations, Kotlin-based apps (we focus on Java-specific issues), apps heavily based on the Native Development Toolkit (NDK[8]), unbuildable apps due to missing keystore information.

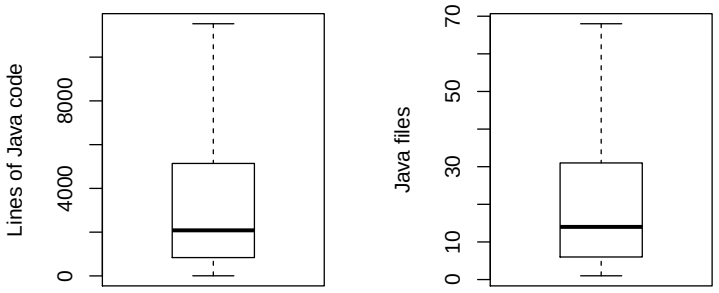
Finally, in step ⑩ we locally clone all the selected GitHub repositories. After this process, our final dataset is composed of **724 GitHub repositories containing open, published, and actively maintained Android apps, for which an analyzable commit history is available**. Out of them, a large majority is from Github(630), followed by FDroid (88) and Wikipedia (6).

As shown in Figure 5.2, the dataset is quite heterogeneous in terms of both lines of Java code (median = 2083, mean = 5325, IQR = 4299) and number of Java files per

[7]Step 5 and Step 7 are redundant, we deliberately decided to keep both of them because during the execution of the dataset creation process we had to experiment with different heuristics in Step 7 and having it as a stand-alone step within the pipeline helped us in easily run it in isolation.

[8]http://developer.android.com/ndk

app (median = 14, mean = 29.38, IQR = 25.0). Moreover, the dataset also covers 24 different Google Play categories (see Figure 5.3) and all downloads ranges (see Figure 5.4).



(a) boxplot for Java loc (b) boxplot for Java files

FIGURE 5.2: Size of the apps in the dataset

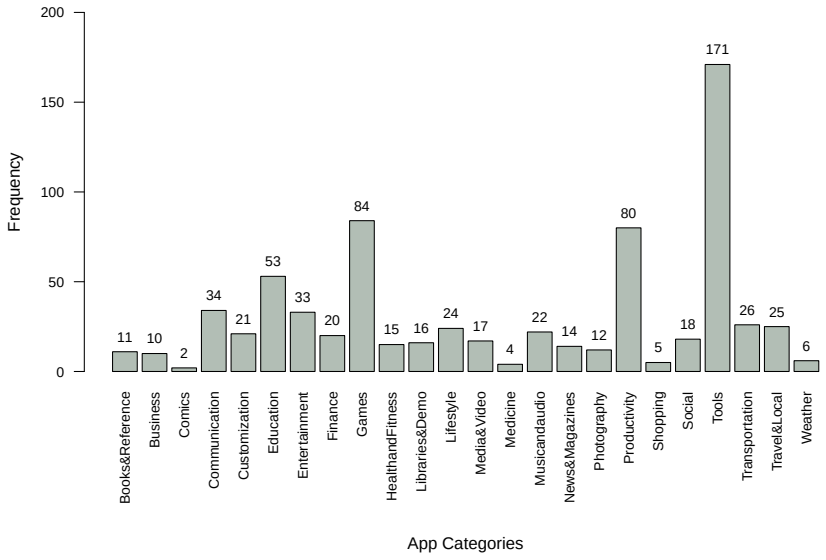


FIGURE 5.3: Google Play categories of the apps in our dataset

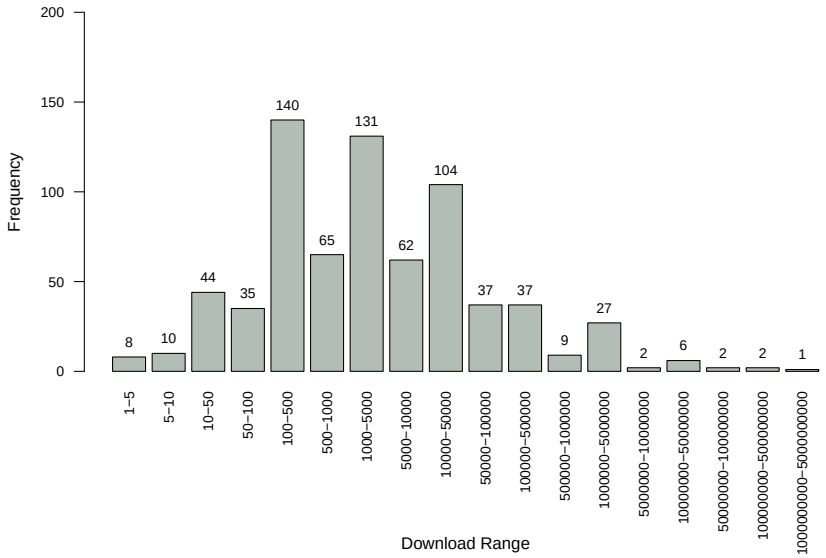


FIGURE 5.4: Google Play download ranges of the apps in our dataset

5.1.3 Data Extraction

Starting from the 724 GitHub repositories, we designed and implemented a tool chain for extracting the data for answering our research questions. As shown in Fig. 5.5, the tool chain is composed of five main steps, which have been implemented as a combination of Python scripts, shell scripts, and third-party tools. Each step will be discussed in details in the following.

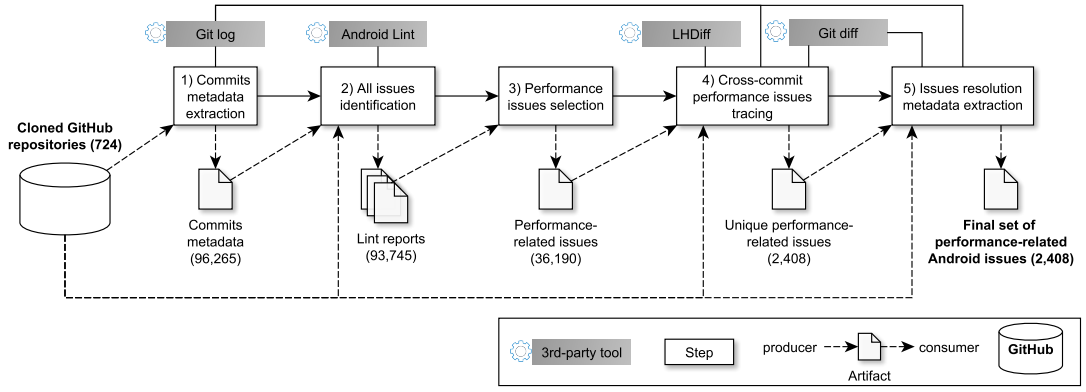


FIGURE 5.5: Data extraction process.

Step 1 – Commits metadata extraction. The first step of our tool chain clones all 724 repositories locally in a dedicated machine. Then, for each cloned repository,

we extract th change log, which contains SHA-1 hash, author, commit message, and timestamp. The extracted data is composed of 96,265 unique items (one for each commit within all GitHub repositories).

Step 2 – issue identification. In this step the tool chain iteratively (i) checks out each cloned GitHub repository at each of the 96,265 commits of our dataset and (ii) runs *Android Lint* on the each commit (as identified in Section 5.1.2) after each check out operation. This results in 96,265 runs of the *Android Lint* tool, which took a total of around 336 hours of uninterrupted execution time (∼2 weeks) on a machine equipped with an Intel i7 processor with 1.80GHz of frequency and 8Gb of memory. It is important to note that Android apps can target different versions of the Android Software Development Kit (SDK). The target SDK of each app must be known in order to properly resolve calls to the Android APIs. Therefore, before executing the *Android Lint* tool, we manually download all Android SDKs used in the apps of our dataset (from API levels 17 to 24) and locally stored them in a known location. Before each execution of the *Android Lint* tool, some preliminary action is necessary to be able to run *Android Lint* on arbitrary-developed Android apps. Firstly, our tool looks for the the app's Lint configuration file (*i.e.,* the `lint.xml` file which may be in the folder containing the the app source code), checks if the Lint configuration file has the `abortOnError` set to `true` and removes it; this check is needed for allowing us to always fully run *Android Lint*, instead of stopping at the first error. Secondly, it removes all statements in the Lint configuration file for disabling any specific check; this check is needed because in some projects the developer may decide to explicitly disable some checks related to Android performance-related issues. Finally, our tool runs *Android Lint* by (i) resolving calls to the Android APIs by referring to the locally-downloaded SDK with the same version as the one specified in the Android manifest file of the current app and (ii) ignoring all calls to external libraries, as we are interested in performance-related issues specific to the apps in our dataset. Starting from the 96,265 commits, *Android Lint* failed 2,520 times (2.61%) because of internal errors of the Android Lint tool (mainly due to peculiar configurations of the Android project). In these cases, the tool discards the considered commit and proceeds with the next one along the versioning history of the current app. This leads to the final set of 93,745 Lint reports, each of them stored as a separate HTML file.

Step 3 – Performance issues selection. In this step our we consider all Lint reports produced in step 2 and extract performance-related issues, as identified by *Android Lint*. To this aim, we developed a parser that takes as input the HTML file of a Lint report and automatically extracts the information we need about each identified issue contained in it. Specifically, the extracted information includes the following data items: (i) the

category of the issue as defined in *Android Lint*[9], (ii) the path to the source code file where the identified issue is located, (iii) the line number in the source code file where the issue is located, and (iv) the raw contents of the warning/error message. Finally, since we are interested in performance-related issues, we filter out all identified issues whose category is different from *performance*. The obtained dataset is composed of 36,190 performance-related issues found in 316 out of 724 apps.

Step 4 – Cross-commit performance issues tracing. After step 3 our set of performance-related issues contains a large number of duplicates. This is expected since an issue remaining in the code base for more than one commit appears multiple times among our set of 36,190 issues; more specifically, it appears exactly once for each commit where it is present. The main goal of this step is to remove those duplicates in order to have a set of unique Android performance-related issues, where each of them can possibly span more than one commit. A naive solution to this problem could have been to simply merge issues reported in subsequent commits which appear in the same Java file and in the same line number. However, a line of code can move during the lifetime of a GitHub project, both across different files (*e.g.,* when a file is renamed or moved within the repository) and within the same file (*e.g.,* some lines of code are added before the considered line of code). In order to correctly match potentially moving lines of code, we exploit the `git diff` tool and the *LHDiff* technique [34, 35] in combination. Specifically, we use `git diff` for building the chain of versions of each file containing at least one performance-related issue, even when it is renamed or moved within the file system. We use `git diff` because it is accurate in identifying the renamed files in GitHub repositories and it is easy to integrate into our tool chain. We use LHDiff for tracking source code lines across two versions of the same file [34]. This tool (i) is language-independent, thus applicable to Java source code files, (ii) has been empirically evaluated and it its mapping process proved to be highly accurate, (iii) is publicly available[10], and (iv) is distributed as a command-line tool, making it easy to integrate into our tool chain. Having a fully reconstructed tracing information about how each issue moves across and within source code files across commits allows our tool to identify those commits which are relevant for our study. Specifically, given an issue i_a of type a (*e.g., UseSparseArrays*) and $C_{i_a} = \{c_1, \ldots, c_n\}$ the set of commits in which i_a is present (*i.e.,* still detected by the tool), we call c_1 the **introducing commit** and c_n the **pre-resolution commit**, The output of this step is composed of 2,408 unique performance-related issues. Each issue contains is represented by all the data items

[9]We take advantage of the fact that issues in *Android Lint* reports are tagged with a fixed set of categories (`http://tools.android.com/tips/lint-checks`) like performance, correctness, accessibility, usability, etc.

[10]`https://muhammad-asaduzzaman.com/research`

described in step 3 and the SHA-1 hash, message and timestamp of its introducing and pre-resolution commits.

Step 5 – Issues resolution metadata extraction. In this step we collect the metadata (*i.e.,* SHA-1 hash, message, timestamp, and number of changed Java lines of code) related to the commit in which each performance-related issue has been resolved by developers (we called them **resolution commits**). In this context, by issue resolution commit we mean the commit immediately after the pre-resolution commit (*i.e.,* the c_{n+1} commit in the discussion above). Moreover, if c_n is the last commit in the whole versioning history of the GitHub repository, it means that we reached the end of the lifetime of the project and the issue has never been resolved; in those cases, the issue is considered as **unresolved**, otherwise it is considered as **resolved** and we keep track of its resolution commit.

TABLE 5.2: Extracted data for each Android Lint performance-related issue type.

Attribute	Type	Description
ID	String	the unique ID of the issue
repository	String	identifier of the GitHub repository of the issue in the form `author/repositoryName`
issueType	factor	type of the performance-related issue as identified by Android Lint (*e.g.,* UseSparseArray, UseValueOf)
LintMessage	String	the warning/error message provided by Android Lint
introHash	SHA-1 hash	SHA-1 hash of the issue-introducing commit
introMessage	String	message of the issue-introducing commit
introTs	Integer	timestamp of the issue-introducing commit
preResHash	SHA-1 hash	SHA-1 hash of the issue pre-resolution commit
preResMessage	String	message of the issue pre-resolution commit
preResTs	Integer	timestamp of the issue pre-resolution commit
isResolved	Boolean	*true* if the issue is resolved, *false* otherwise
resHash	SHA-1 hash	SHA-1 hash of the issue-resolution commit
resMessage	String	message of the issue-resolution commit
resTs	Integer	timestamp of the issue-resolution commit
resLOC	Integer	number of Java LOCs changed in the issue-resolution commit

In Table 5.2 we summarize the data extracted for each Android performance-related issue. It will be used across the whole study and will be the base for the data analysis phase.

5.2 RQ2.0 Results – Performance issues identified by Android Lint

5.2.1 Data Analysis (RQ2.0)

For answering RQ2.0, we present and discuss (i) the number of performance issues identified by *Android Lint* across all apps, (ii) the frequency and distribution of each type of performance issue across all apps, and (iii) the distribution of the number of occurrences of each type of performance issues per app. Moreover, in order to better characterize statically-detectable Android performance issues, for each type of performance issue we provide and discuss an example of Java code exhibiting the issue. At the time of the experiment execution, *Android Lint* supports 9 types of performance issues[11], they are presented in Table 5.3. Finally, to provide an overview of the apps' popularity, we use bar plots to show the range of number of downloaded apps from Google Play Store for each types of identified performance issue.

TABLE 5.3: The types of performance issues considered in this study.

Issue name (priority)	**Description**
DrawAllocation (9/10)	It generally occurs due to allocating memory in a method that is invoked frequently to draw UI elements on the display. Allocating memory can be avoided by allocating the memory upfront, which leads to increased performance, thus potentially leading to a smoother user experience.
FloatMath (3/10)	It deals with the `FloatMath` data type; specifically, on modern devices the `FloatMath` Java object is slower than using `java.lang.Math` due to the way the JIT optimizes `java.lang.Math` objects.
HandlerLeak (4/10)	It is due to handler using the `Looper` or `MessageQueue` of the main thread. If the handler is not static, then the Android activity or service cannot be garbage collected, even after being destroyed. This may lead to memory leaks.
Recycle (7/10)	It occurs with the lack of calls to the `recycle()` method, when dealing with recyclable objects, such as `TypedArray`, `VelocityTracker`, etc. Calls to the `recycle()` method should be done after one of the above mentioned objects have been used, in order to make it reusable in the future.
UseSparseArrays (4/10)	It is mainly due to the use of `HashMap` instead of `SparseArray`. The Android framework promotes the usage of `SparseArray` over `HashMap` since it is assumed that sparse arrays are more memory efficient than `HashMap`, while not exhibiting large performance differences when dealing with hundred of items.
UseValueOf (4/10)	It is mainly due to direct calls to the constructor of wrapper classes (*e.g.,* `new Integer(42)`), as opposed to calling the *valueOf* factory method (*e.g.,* `Integer.valueOf(42)`). Calling factory methods is typically more memory efficient since common integers such as 0 and 1 share a single instance at run-time.
ViewHolder (5/10)	It occurs in the context of `ListViews`. When implementing a view Adapter, developers should avoid unconditionally inflating a new layout; if an available item is passed in for reuse, developers should try to use that one instead.
ViewTag (6/10)	Before the Android 4.0 version, `View.setTag(int, Object)` implementation stored the objects in a static map, where the values were strongly referenced. This implies that if the object references its calling context, the leak will happen from the context (which potentially may point to a large number of other objects within the app).
Wakelock (9/10)	It is due to failing to release a *WakeLock* properly, thus keeping the mobile device in high power mode, which decreases the lifetime of battery.

[11] `http://tools.android.com/tips/lint-checks`

We anticipate that in the remainder of this study we will not consider the *ViewTag* and *Wakelock* issues since they both occur only 3 times each within the whole dataset.

5.2.2 Results (RQ2.0)

A total of 2,408 performance issues have been detected by Android Lint. Among them, 316 (43.64%) over 724 distinct apps suffered from at least one statically-detectable performance issue along their lifetime. Fig. 5.6 presents the frequency of different types of performance issues in our dataset.

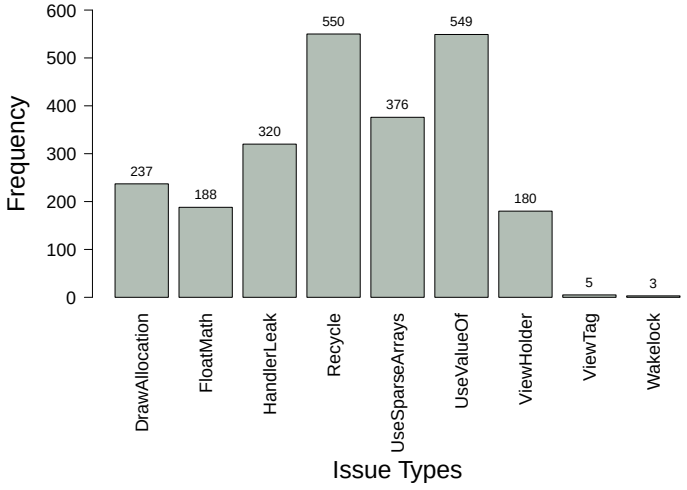


FIGURE 5.6: Frequency of performance issue belonging to different categories.

First, we can immediately notice how *Recycle* performance issues occur more frequently (550, 22.84%) than others. In order to optimize performance, collections such as *TypedArray*, *VelocityTracker*, *Parcel* or *MotionEvent* should be recycled after use, instead re-created again and also database cursor should be freed up after use. For example, as shown in Listing 5.1, *TypedArray preset_vals* should be recycled by a *recycle()* call (*i.e., preset_vals.recycle()*) for further reuse. The lack of a recycle could noticeably degrade the performance of the app.

```
protected void presetClicked(int i) {
  Log.i(TAG, "presetClicked "+i);

  TypedArray preset_vals = mContext.getResources().obtainTypedArray(
  mContext.getResources().getIdentifier(
  "presets"+mFnId+"_"+i, "array", mContext.getPackageName()));

  for(int j=0; j<mSliders.size(); j++) {
    float val = preset_vals.getFloat(j, 0);
```

```
    Log.i(TAG, "slider["+j+"]="+val);
    mSliders.get(j).setValue(val);
  }
}
```

LISTING 5.1: Example of occurrences of the *Recycle* issue (`dstahlke/rdn-wallpaper` - src/org/stahlke/rdnwallpaper/PresetsBox.java).

Also *UseValueOf* issues type are quite frequent in our dataset (549, 22.79%). Since issues of type *UseValueOf* primarily deal with primitive types, we can conjecture that developers deem as negligible the potential performance improvement when resolving this kind of issues. Nevertheless, developers should take care of those issues since after a manual analysis we noticed that they may occur in the burst (Listing 5.2 for an example obtained from our dataset), thus potentially impacting the performance of the app in a noticeable manner.

```
//The Special Key Codes
mSpecialCodes = new Hashtable<Integer, String>();
mSpecialCodes.put(new Integer(-900), "++");
mSpecialCodes.put(new Integer(-901), "--");
mSpecialCodes.put(new Integer(-902), "&&");
mSpecialCodes.put(new Integer(-903), "||");
mSpecialCodes.put(new Integer(-904), "\\\\");
mSpecialCodes.put(new Integer(-905), "//");
mSpecialCodes.put(new Integer(-906), "==");
mSpecialCodes.put(new Integer(-907), "<=");
mSpecialCodes.put(new Integer(-908), ">=");
mSpecialCodes.put(new Integer(-909), "!=");
mSpecialCodes.put(new Integer(-910), ">>");
```

LISTING 5.2: Example of bursty occurrences of the *UseValueOf* issue (`dyne/ZShaolin` - termapk/src/com/spartacusrex/spartacuside/keyboard/TerminalKeyboard.java).

UseSparseArrays (376, 15.61%) type of issues are third most common occurrence in our dataset. As discussed in Table 5.3, a performance degradation may occur when developers use a `HashMap` and the maps grows in an unpredicted manner. Listing 5.3 provides an example of `HashMap` Usage.

```
private void randomizeOptions(){
HashMap<Integer, Word> optionsMap = new HashMap<Integer, Word>();
for(int i = 0 ; i < optionsList.size() ; i++){
  while(true){
    int rand = (int)((Math.random() * 10) % 5);
    if(optionsMap.containsKey(rand)){
      continue;
    }else{
      optionsMap.put(rand, optionsList.get(i));
```

```
      break;
    }
  }
}
```

LISTING 5.3: Example of occurrences of the `UseSparseArrays` issue (`ric03uec/cramit` - src/com/dev/cramit/models/Problem.java).

Then, there are potential *HandlerLeak* issues (320, 13.28%) in our dataset. The main consequences of this type of issue are memory leaks in the projects. To avoid this issue, developers should declare the handler as static. From the manual analysis in our dataset, we observed that in few cases developers intentionally resolved this issues, for example in Listing 5.4, developers mentioned in their comment (in the Java source code file) to declare static handler and for that, in the very next commit, they declared handler as static to get rid from potential memory leak issues. There are also many projects present in our dataset where developers do not declare the handler as static, which may lead to suffering from potential memory leaks as shown in Listing 5.5.

```
// TODO make this Handler static to prevent memory leaks
private Handler communicatorServiceHandler = new Handler() {
/**
* Needs to know which parameters are passed back in which predefined
* fields of the {@link Message}. </p>
* <ul>
* <li><code>what</code> - hash of the related
* {@link IntentAction.WebService} constant</li>
* <li><code>arg1</code> - one of the constants declared in
* {@link WSConstants.Result}</li>
* <li><code>obj</code> - the returned {@link Bitmap}</li>
* </ul>
*/
```

LISTING 5.4: Example of occurrences of the `HandlerLeak` issue (`mobiRic/StackFlairWidget` - src/com/mobiric/stackflairwidget/service/FlairWidgetService.java).

```
private final Handler mHandler = new Handler() {
  @Override
  public void handleMessage(Message msg) {
    switch (msg.what) {
    case MESSAGE_UPDATE_HARDWARE_PARAMETERS:
      updatedHardwareSettingsValues();
      break;
    }
  }
};
```

LISTING 5.5: Example of occurrences of the `HandlerLeak` issue (`alistairdickie/BlueFlyVario_Android` - src/com/bfv/hardware/HardwareListActivity.java).

Also there are issues of type *DrawAllocation* (237, 9.84%), can be found in various projects in our dataset. This is mainly due to allocation of memory when draw or layout operation is frequently invoked in a method. Thus it causes assigning memory each time whenever the function is called. From the manual analysis, we can presume that there are few projects where developers resolved these issues but still many projects suffering from these type of issues. For example, there are several new objects such as `new ArrayList` (at line 4 and 12) and `new pie (500)` (at line 18), are allocated in `onDraw()` (*i.e.,* allocating objects during a draw operation), heavily lead to UI lag as shown in Listing 5.6.

```
protected void onDraw(Canvas canvas) {
  List<Double> numbers = new ArrayList<Double>();
  Employee employee = new Employee(_hoursWorked, _this);
  //numbers.add(Double.parseDouble(employee.gross()));
  numbers.add(employee.payeDouble());
  numbers.add(employee.studentLoanDouble());
  numbers.add(employee.kiwiSaverDouble());
  numbers.add(employee.nettDouble());
  double total = employee.grossDouble();
  List<Integer> colours = new ArrayList<Integer>();
  colours.add(Color.BLUE);
  colours.add(Color.GREEN);
  colours.add(Color.RED);
  colours.add(Color.MAGENTA);
  colours.add(Color.YELLOW);
  Pie p = new Pie(500);
  Paint wallpaint;
```

LISTING 5.6: Example of occurrences of the `DrawAllocation` issue (`kurtmc/MyEarnings` - src/com/mcalpinedevelopment/calculatepay/CalculateActivity.java).

There are (188, 7.80%) issues of type *FloatMath* in our dataset. Listing 5.7 shows an example of this type of performance issue.

In previous versions of Android, when working on floats, `android.util.FloatMath` was referred to ensure for performance reasons. However, on the latest hardware the performance achieved when using floats is equal than when using doubles (though they take more memory), and on the current Android versions, FloatMath is slower than using `java.lang.Math` [2] .

```
// if values before left of decimal are the same -> need to show float decimals
if (android.util.FloatMath.floor(min) ==    android.util.FloatMath.floor(max))
{
  minS = Float.toString(min);
  maxS = Float.toString(max);

// show same length decimals!
  if (minS.length() > maxS.length())
  {
    minY.setText(minS.substring(0, maxS.length()));
    maxY.setText(maxS);
  }
  else
  {
    minY.setText(minS);
    maxY.setText(maxS.substring(0, minS.length()));
  }
}
else  // otherwise only show integers
{
  minY.setText(Integer.toString((int)(min)));
  maxY.setText(Integer.toString((int)(max)));
}
```

LISTING 5.7: Example of occurrences of the `FloatMath` issue (`dirktrossen/AIRS` - src/com/airs/TimelineActivity.java).

Issues of type *ViewHolder* (180, 7.47%) are more recurrent in our dataset after *FloatMath*. This type of issue primarily deals with the smoother scrolling of `ListView`. To show the `ListView` items, system has to draw each item separately. To reduce the number of `findViewById()` calls every time (when a list object has to draw), data from last drawn object can be reused (*i.e.,* mainly by creating `ViewHolder` patterns). As shown in Listing 5.8, in `getView()` function every time a new object is draw (at line 5) followed by calling of `findViewById()` (at line 8) each time which may degrade the performance of app *i.e.,* lag in smoother `ListView` scrolling. However, there are also certain cases in our dataset where developers specially implemented `ViewHolder` pattern class to avoid this issue.

```
@Override
public View getView(int position, View view, ViewGroup parent) {
  LayoutInflater inflater = (LayoutInflater)
    getContext().getSystemService(Context.LAYOUT_INFLATER_SERVICE);
  view = inflater.inflate(R.layout.text_with_delete, parent, false);

  final String textItem = getItem(position);
  TextView textView = (TextView) view.findViewById(android.R.id.text1);
```

```
  textView.setText(textItem);
  // add listener to the delete button
  Button button = (Button) view.findViewById(android.R.id.button1);
  button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
      //delete button clicked
      onDeleteListener.onDelete(textItem);
    }
  });

  return view;
}
```

LISTING 5.8: Example of occurrences of the `ViewHolder` issue (`asksven/BetterWifiOnOff` - BetterWifiOnOff/src/com/asksven/betterwifionoff/CreditsAdapter.java).

ViewTag (5, 0.20%) are issues type that can be found very rarely in our dataset. These issues are related to the implementation of *View.setTag(int, Object)* and occurred in prior to Android 4.0. The consequences of this issue are leaks in the apps.
WakeLock (3, 0.12%) type of issues which are very less recurrent in our dataset. The WakeLock happened due to the failure to release a WakeLock properly that could keep the mobile device in high power mode and reduces the lifetime of the battery. There are many reasons of this phenomenon, such as failing to call *release ()* in all possible code paths containing *acquire()*, releasing the WakeLock in *onDestroy()* instead of in *onPause()*, and so on and so forth. Since this, it is a very crucial performance issue and developers do take care about the lifetime of battery while developing the app (*i.e.,* releasing the wakelock properly), and thus it may be one of the reasons why we have less *WakeLock* issues in our dataset.
Moreover, Table 5.4 reports the descriptive statistics for the number of statically- detectable performance issues per app. It can be clearly seen from Table 5.4, that *Recycle* issues occur quite frequently in our dataset (mean = 1.741 issues per app) followed by *UseValueOf* issues (mean = 1.737 issues per app) with higher standard deviation.

Also, issues of type *UseSparseArrays* and *HandlerLeak* are quite widespread in the analyzed apps, with an average 1.190 and 1.013 issues per app, respectively. Instead, issues of type *DrawAllocation* and *FloatMath* are relatively less frequent in our dataset, with 0.750 and 0.594 issues in each app respectively. *ViewHolder* issues are less frequent in our dataset (*i.e.,* Mean = 0.569 issues). These issue type have also the lowest coefficient of variation than other issues.

TABLE 5.4: Descriptive statistics for the number of statically-detectable performance issues per app (SD = standard deviation, CV = coefficient of variation).

Issue type	**Max.**	**Mean**	**SD**	**CV**
DrawAllocation	28	0.750	2.543	3.39
FloatMath	61	0.594	4.404	7.402
HandlerLeak	42	1.013	3.855	3.807
Recycle	75	1.741	6.114	3.513
UseSparseArrays	84	1.190	5.394	4.533
UseValueOf	171	1.737	10.478	6.031
ViewHolder	17	0.569	1.655	2.906

Regarding the relationship between issue occurrence and popularity of apps in terms of number of downloads, issues of type *UseValueOf* were found 244 times in apps downloaded 10,000-50,000 times from Google Play Store; issues of type *UseSparseArrays* occurred 129 times in highly-downloaded apps (1M-5M times). Fig. 5.7 provides an overview of the different types of performance issue identified in apps having varying download ranges. One thing that immediately leaps to the readers' eyes is that, for apps having a very high number of downloads (greater than 5M) the issue frequency suddenly decreases. Although we do not have evidence of that, it is possible that such very popular apps undergo a more accurate quality check (*e.g.,* code review), which may explain such a drop. We can also notice a very high frequency of *UseValueOf* issues for apps having a medium (1k-5k) number of downloads. Although this issue (calling constructors of wrapper classes instead of factory methods) may result is a waste of memory, we cannot tell why it happens for apps in that specific range of downloads.

Summary – RQ2.0 – A total of 2,408 performance-related issues has been detected by Android Lint. *Recycle* issues are the most recurrent ones in our dataset (550, 22.84%), whereas *WakeLock* are the least recurrent ones (3, 0.12%).

5.3 RQ2.1 Results – Evolution of the Number of Android Performance Issues over Time

5.3.1 Data Analysis (RQ2.1)

We answer RQ2.1 by analyzing each of the 316 apps with at least one performance-related issue. For each app, we firstly reconstruct its versioning history by considering the sequence of all commits in its GitHub repository; then, for each commit we count the occurrences of any type of performance issue. The final result of this activity is a

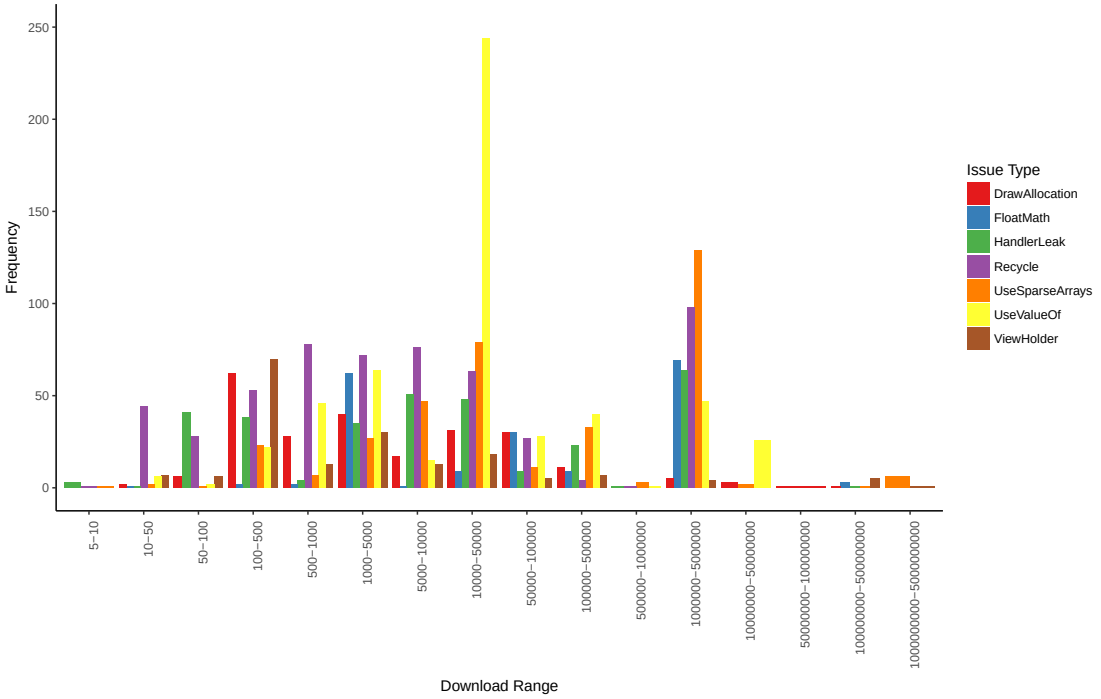


FIGURE 5.7: RQ_0: Relationship between performance issue types and number of apps' downloads.

set of 316 plots, showing the evolution of the number of performance issues over the lifetime of each app, as well as the app size evolution in terms of LOC. The reason why we observe these two variables is because the former represents the main factor being investigated in this study, and the latter is a factor we need to control. This is because the growth of performance issues could be considered more problematic when it happens more rapidly than size increase. Examples of generated plots are shown in Section 5.3.2.

In order to characterize how performance issues evolve in Android apps, we perform a *qualitative study* on the plots and manually categorize them into relevant groups by applying the *open card sorting* technique [107]. We perform the card sorting in two phases. First, we manually tag each plot with considerations about the presence of relevant evolution patterns, *e.g.,* presence a spike, plateaus, sudden drops, etc. Then, we cluster identified patterns into meaningful groups with a descriptive title; each plot can exhibit more than one issue evolution pattern, *i.e.,* it can belong to more than one group. To minimize bias, three researchers have been involved in the card sorting activity. Specifically, we randomly selected 100 apps from the dataset and the main author of this study categorized them. Then, the same 100 apps have been randomly assigned to the other two researchers involved in this study (50 apps each), who categorized them independently. The three emerging sets of categories were slightly different (see our replication package for their specific items) and have been collaboratively discussed

and merged in order to agree on a final set of categories of evolution trends of Android performance issues.

Finally, the main author of this study categorized the 316 plots into the different possible evolution patterns, and the task was repeated by the second author for the first 158 apps and by the third author for the last 158 apps, so that for each app there were at least two taggers performing the classification. In order to further reduce bias, the other two researchers re-applied the final set of categories to their initially assigned 50 apps and the level of agreement between each researcher and the first one has been assessed by means of the Cohen-Kappa statistics [42].

After the application of the open card sorting technique, we report and analyze the frequency of performance issue evolution patterns across all 316 apps. Moreover, from our analysis it also emerged that a large number of apps contain a combination of different categories of issue evolution patterns. In order to better investigate this aspect, we statistically assess their correlation by building a contingency table with rows and columns representing each issue evolution pattern and computing its Cramer's V coefficient [102]. The Cramer's V coefficient is a well-known measure of association applicable to contingency tables involving two categorical variables and it is defined within the $[0, 1]$ range, where 0 indicates no correlation and a value of 1 indicates perfect correlation.

5.3.2 Results (RQ2.1)

To answer $\mathbf{RQ}_1$, we analyzed the evolution of the number of performance issues throughout the lifetime of each GitHub repository. The analysis of the 316 repositories of our dataset via the open card sorting technique described in Section 5.3.1 resulted in the identification of five different evolution patterns. Table 5.5 reports the emerging patterns[12].

As discussed in Section 5.3.1, three researchers have been involved in the identification of the categories of issue evolution patterns iteratively and collaboratively. For each evolution pattern category, the Cohen Kappa index between the first author and the second and between the first author and the third is calculated. In all cases the Cohen Kappa is > 0.6, hence indicating a strong agreement.
Fig. 5.8 reports the distribution of the issue evolution patterns across the 316 apps containing at least one occurrence of each performance issue. We can observe that STICK (209) and REF (124) are the most frequent performance issue evolution patterns, followed by BEG (111), INJREM (69), and GRAD (41). In the following, we discuss in detail about these evolution patterns.

[12]In the remainder of the study we will refer to performance issues as P and to lines of code as LOC.

TABLE 5.5: Categories of evolution patterns of Android performance issues, where P = the number of performance issues, LOC = lines of code, ∼ = irrelevant for the identification of the evolution pattern.

Pattern	**Description**	**P**	**LOC**
STICK	*Sticky issues*, *i.e.,* issues are injected abruptly and they remain in the app across several commits	step ↑	∼
REF	*Refactoring* of performance-related issues	step ↓	∼
BEG	Issues since the *beginning* of the project	stable != 0	∼
INJREM	*Injection and removal*, *i.e.,* a relatively large number of issues is injected in the project, followed by a sudden removal	spike	∼
GRAD	*Gradual*, *i.e.,*, performance issues gradually occur during the app development process	=LOC	=P

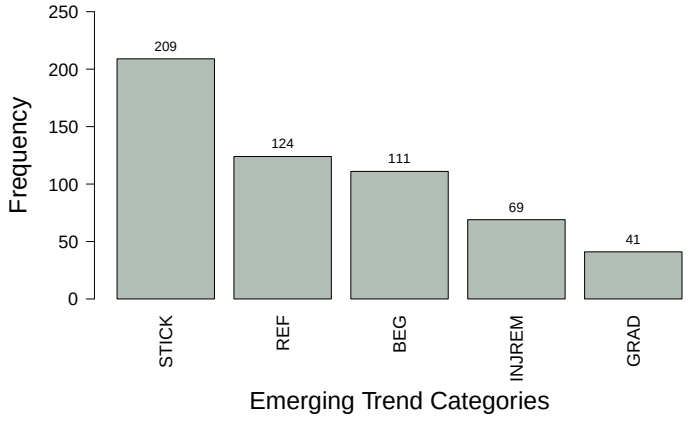


FIGURE 5.8: Occurrences of performance issue evolution patterns across apps.

STICK (Sticky issue). This pattern refers to issues which are introduced and remain in the project for several commits. In such a case, either the issue was not considered a serious concern (or even it was a tool's false positive), or developers had other priorities but resolving the performance issue. Fig. 5.9 show the example of STICK patterns found in the analyzed apps. As it can be seen from Fig. 5.9, one issue (*i.e., HandlerLeak*) is introduced in the *alistairdickie/BlueFlyVario_Android project* and continue to remain in the system for many commits. As the project development progress further, another issue (*i.e., HandlerLeak*) is injected with the addition of new lines of code and then both these issues are stick to project for several commits (till the end of the project).

REF (Refactoring). This type of pattern indicates a possible refactoring action in the evolution of projects *i.e.,* performance issues are resolved from the project with the increase or decrease of lines of code. In REF patterns, the number of performance issues dropped consistently, regardless of whether the overall LOC increased or decreased.

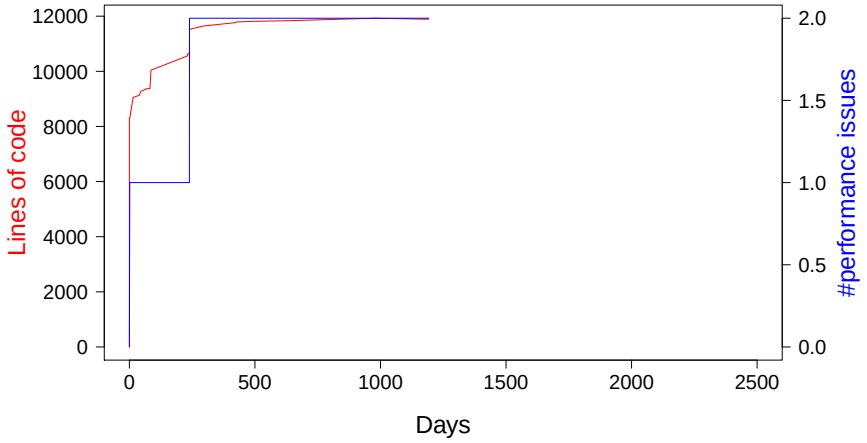


FIGURE 5.9: alistairdickie/BlueFlyVario_Android - An example of STICK Evolution Pattern.

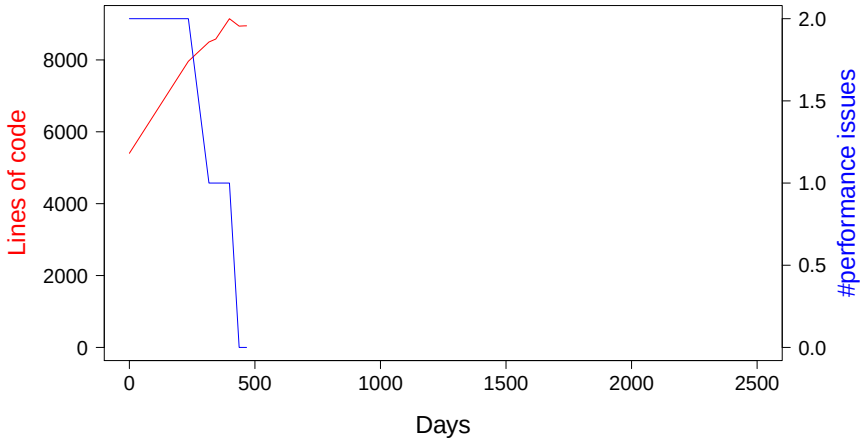


FIGURE 5.10: xperia64/timidity-ae - An example of REF Evolution Pattern.

From the manual analysis, we noticed that intentional refactoring done twice in the project xperia64/timidity-ae (as shown in the Fig. 5.10). Initially one *UseSparseArrays* issue was resolved by using `SparseIntArray` instead of `HashMap<Integer, Integer>`, whereas in the second refactoring ViewHolder Pattern was implemented inorder to resolved the *ViewHolder* issue.

vspace-10mm

BEG (issues since the Beginning). This pattern refers to cases in which performance issues which are present in the project since the beginning *i.e.,* when the project was created. As it can be seen from Fig. 5.11, related to the *offbye/ChinaTVGuide* app, this

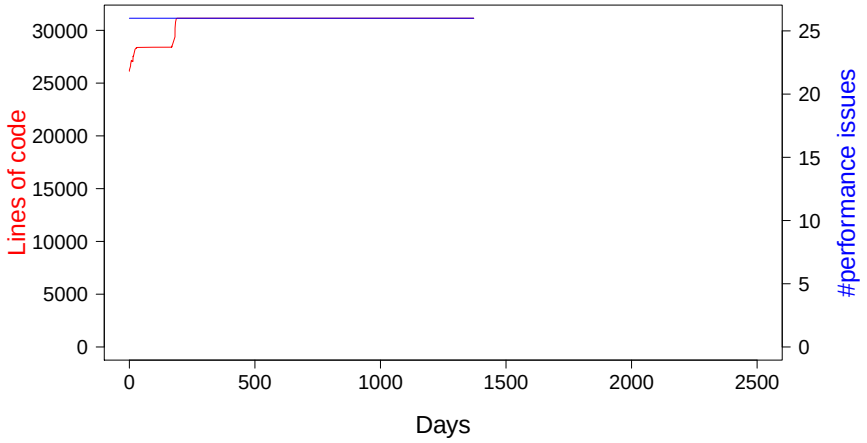


FIGURE 5.11: offbye/ChinaTVGuide - An example of BEG Evolution Pattern.

app contained 25 performance issues since the project's creation on GitHub, and they remained unaltered till the end of our observation period.

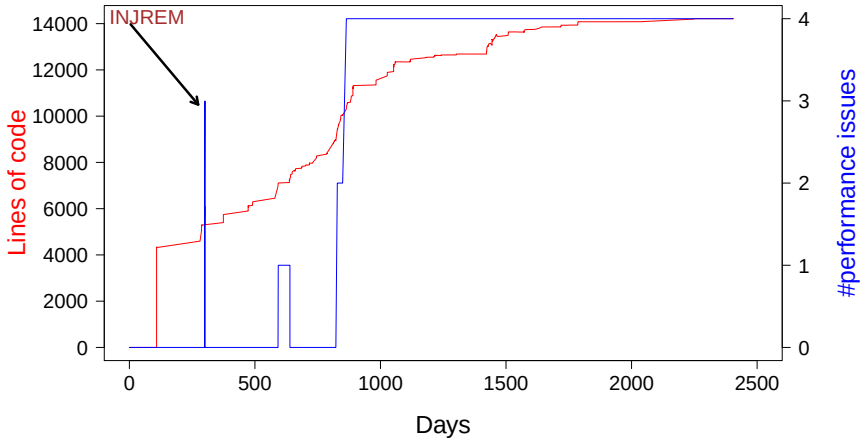


FIGURE 5.12: mi9rom/achartengine - An example of INJREM Evolution Pattern.

INJREM (Injection and removal). We identify the INJREM evolution patterns when a relatively large number of issues are injected in the project followed by a quick removal. We consider the resolution of a performance issue to be quick if it happens within two days from the introduction. Fig. 5.12 show the example of INJREM pattern from our dataset. From Fig. 5.12 (pattern shown by the black arrow), it can be noted that in the app *mi9rom/achartengine* three issues were introduced (*i.e., UseSparseArrays*) and in the very short time interval, these issues were suddenly resolved. By manually inspecting the documented commit, we observed that the resolution was not done with intention of improving the performance of the app (*i.e.,* the resolution was accidental).

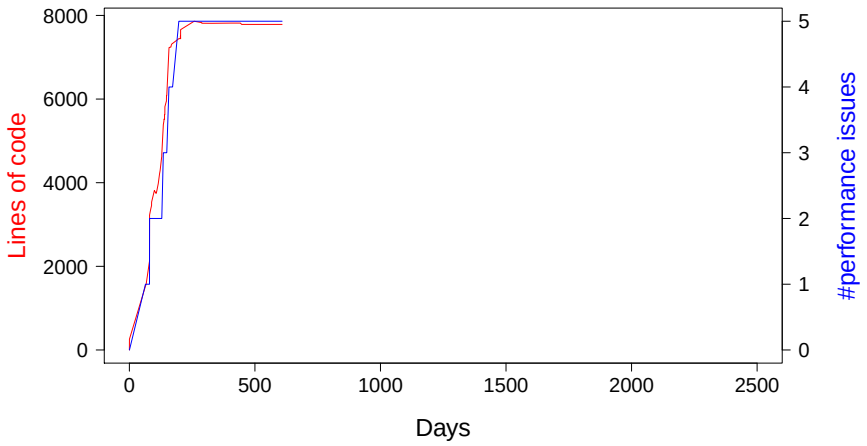


FIGURE 5.13: AlbertoCejas/GermanLearningUCA - An example of GRAD Evolution Pattern.

GRAD (Gradual issues). In GRAD evolution pattern, performance issues gradually increase with the LOC or vice versa. In this type of evolution patterns, performance issues and lines of code grow or decrease altogether. In other words performance issues are an integral part of the app development process. As shown in Fig. 5.13, performance issues in the app *AlbertoCejas/GermanLearningUCA* (an example from our dataset) grow with the same rate of LOC. In other words, performance issues increase gradually with the gradual increase of lines of code.

From the analysis of evolution patterns, we observed that many apps in our dataset which contain performance issues follow multiple co-occurrence patterns throughout the lifetime of their projects. To provide a quantitative indication of the association strength among patterns, we compute the Cramer's V coefficient, which measures the strength of association — varying between 0 to 1 — between two nominal variables. In our study, the computed Cramer's V coefficient value is 0.319 (which is low) meaning that there is a low association between the categories of emerging patterns.

Summary — RQ2.1 – Five different evolution patterns emerged from the card sorting technique. STICK is the most frequent pattern in our dataset (209 occurrences), meaning that for 209 over 316 apps, issues tend to remain in the source code for several commits after their introduction. Moreover, while many patterns in our dataset co-occur, we did not find any pair of patterns exhibiting a high level of association.

5.4 RQ2.2 – Performance Issues remaining in Android Apps over Time

5.4.1 Data Analysis (RQ2.2)

To answer RQ2.2, we introduce two variables: RS_a^i and US_a^i. RS_a^i is defined as the number of resolved issues of type i (*e.g., DrawAllocation*) across the whole lifetime of app a. As discussed in Section 5.1.3, in this study a resolved issue is an issue which is present in the GitHub repository (for any number of commits up to commit c_j) and it is not present in the repository from commit c_{j+1} (in this case commit c_{j+1} is called the issue-resolution commit). Conversely, US_a^i is defined as the number of unresolved issues of type i within the lifetime of app a. Clearly, the sum of RS_a^i and US_a^i is equal to the total number of issues of type i for app a.

Given its exploratory nature, we answer RQ2.2 by extracting and discussing the following information for each app a within our dataset and for each type of performance issue i: the ratio between the total number of performance issues of type i in a and its unresolved issue (resolved and unresolved issue are complementary to each other, therefore we only keep the unresolved ones).

Finally, we depict with bar plots the relationship between the type unresolved issue frequency and number of apps' downloads.

5.4.2 Results (RQ2.2)

TABLE 5.6: Total number of Android performance issues and their subset of unresolved issues

Category	**All**	**Unresolved (%)**
ViewHolder	180	101 (56.11%)
UseSparseArrays	376	196 (52.13%)
DrawAllocation	237	119 (50.21%)
HandlerLeak	320	160 (49.00%)
Recycle	550	240 (43.64%)
UseValueOf	549	218 (39.71%)
FloatMath	188	52 (27.66%)
Total	2,400	1,086 (45.25%)

As shown in Table 5.6, *ViewHolder* issues tend to remain more than other types of performance issues across all apps in our dataset (101/180, 56.11%). Since *ViewHolder* issues primarily deal with the smoother scrolling of *ListView* items in Android apps, they can be more problematic in terms of end users' experience. These issues tend to arise in the apps either due to lack of declaring a *ViewHolder* pattern, or not reusing

the previously drawn items *i.e.,* by reducing the number of calls to the *findViewById()* method.

Similarly, *UseSparseArrays* issues (196/376, 52.13%) tend to remain unresolved in our dataset. For the purpose of efficient memory usage and less garbage collection, the Android platform promotes the use of `SparseArrays` instead of `HashMaps` for maps that contain up to a hundred values. From manual inspection, we assume that these issues remain unresolved due to unnoticeable memory improvements in terms of amount of memory (in bytes) allocated in the heap of the Java Virtual Machine.

The *DrawAllocation* issues (119/241, 50.21%) are the third less frequent issue type remaining unresolved in our dataset. Since draw operations are quite sensitive in terms of user-perceived performance, it is a bad programming practice to allocate memory (*i.e.,* by declaring new instances) during draw or layout operations. This is one of the troublesome issue type that continues to remain alive in the apps.

The consequence of having *HandlerLeak* (160/320, 50.00%) issues is to have a memory leaks in the app, potentially leading to the usage of unneeded memory over long usage sessions. This type of issues can be resolved by declaring the handler as static. Finally, we analyzed that *FloatMath* (52/188, 27.66%) issue type, which is resolved more frequently in our dataset as compared to other types of performance issues. Similar to *UseValueOf* issues, they also mainly deal with primitive data types and are resolvable in a relatively straightforward manner.

Concerning the relationship between of unresolved issues and apps' downloads, results shown in Fig. 5.14, highlight how `UseSparseArrays` issues remained unresolved 69 times in apps downloaded in the range of 1M-5M. Similarly to what found in RQ_0, also in this case we notice a difference (drop in the frequency of unresolved issues) for apps having a high number of downloads, above 5M. At the same time, it is also interesting to notice how *UseSparseArray* issues (see the orange bar) tend to exhibit a relatively high frequency also for apps with a high number of downloads. This kind of issue suggests the use of `SparseArrays` instead of `HashMaps`, but it is possible that developers do not consider such an optimization as important given the size of the data they have to deal with in their apps.

Summary — RQ2.2 – Overall, 45.25% of performance issues remain unresolved. *ViewHolder* issues are the ones remaining more in the app, even though they can be problematic in terms of end users experience since they primarily deal with a smoother scrolling of `ListView` items. *FloatMath* issues are the most resolved performance issues (52/188, 27.66%).

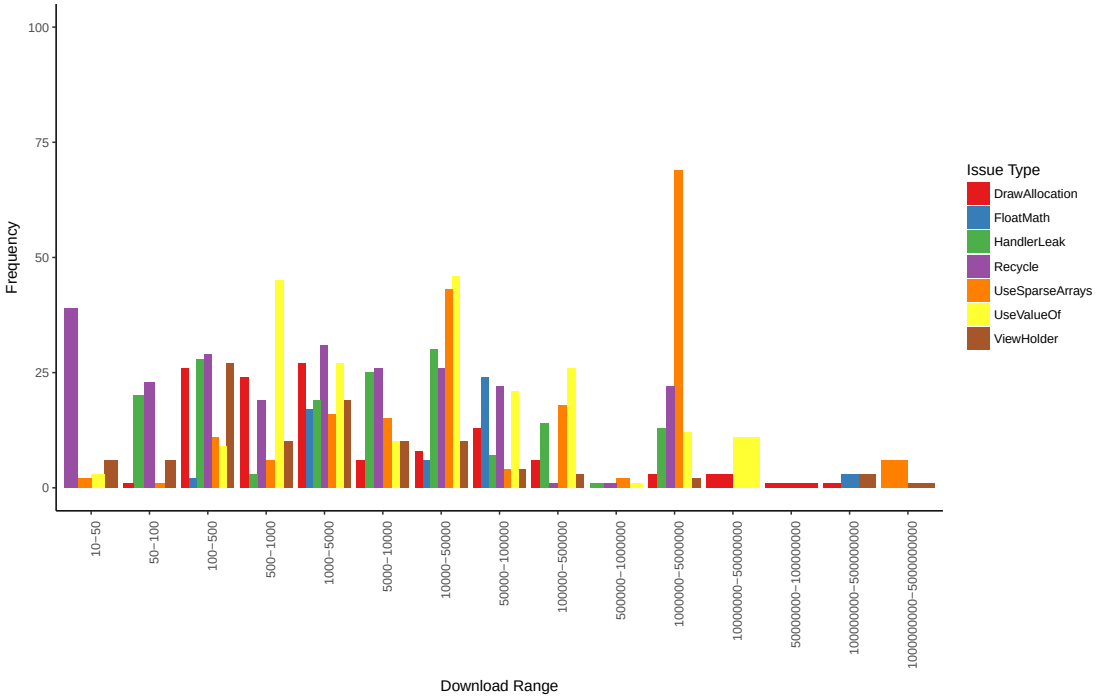


FIGURE 5.14: RQ_2: Relationship between types of unresolved issues and number of apps' downloads.

5.5 RQ2.3 Results – The Lifetime of Android Performance issues

5.5.1 Data Analysis (RQ2.3)

We define the lifetime of an Android performance issue as the number of days between its issue-introducing commit and its issue pre-resolution commit. Intuitively, such a lifetime represents the time interval in which the issue is present in the source code of an app.

We answer RQ2.3 in two phases. In the first phase we perform an initial exploration of the obtained lifetimes across all apps and report the summary statistics (together with box plots). In this phase, we order all issues according to their lifetime and trim outliers by removing the top 1% of all issues so to avoid potential issues related to repositories which have not been actively maintained in the last months[13]. In order to compare the different duration distributions, we apply the Kruskal-Wallis test [75] for each type of performance-related issue, followed by a Dunn post-hoc analysis [49]. Since we are applying multiple statistical tests, in order to reduce the chance of Type-I error we correct the obtained p-values via the Holm p-value adjustment procedure [64]. The

[13]It is important to note that we performed the statistical analysis for RQ2.3 both with and without outliers and the conclusions did not change

procedure sorts the n p-values obtained by the multiple comparisons in increasing order, and multiplies the smallest one by n, the second-smallest by $n-1$, and so on (the largest one is left unchanged).

In the second phase, we aim at understanding whether Android performance issues exhibit some recurrent patterns in terms of resolution time. Specifically, we firstly reconstruct the cumulative distribution function (CDF) of the lifetime of each type of performance issues across all apps, then we plot it on a day-scale, and finally we investigate on whether each built CDF can be modeled using known statistical distributions. Specifically, we consider six known statistical distributions – *Cauchy, Exponential, Gamma, Lognormal, Normal and Weibull* – and assess to what extent each type of Android performance issues fits each of them. In total, we obtained 42 pairs (*i.e.,* 6 statistical distributions $\times$ 7 types of Android performance issues) and assessed their fits. When fitting the CDFs to the known statistical distributions we follow a procedure similar to the one applied in [97], *i.e.,* we (i) visually inspect each of the 42 plots showing together the CDF and known distribution in combination and (ii) statistically test how the known statistical distributions fit the CDFs by applying the Kolmogorov-Smirnov (KS) test to each pair. Specifically, we firstly estimate the distribution parameters of each CDF using the method of Maximum Likelihood, which maximizes the likelihood that the set of data used for the estimation can be obtained from the statistical distribution modeled with the estimated parameters. Then, we apply the Kolmogorov-Smirnov (KS), a non-parametric test that checks whether a distribution fits a given data (H_0 – there is no significant difference between the theoretical distribution and the actual data distribution). This means that every time the p-value of the applied KS test is greater than $\sigma = 0.05$, then the CDF fits the distribution. In subsection 5.5 we report all CDFs with the best fitting known statistical distribution, all the obtained p-values, and a discussion of the obtained results. Finally, we use bar plots to depict the relationship between the type of frequently-resolved issues and the apps' number of downloads.

5.5.2 Results (RQ2.3)

As discussed in section 5.5.1, we answer $\mathbf{RQ_3}$ by analyzing the lifetime of each type of Android performance issue. As an initial exploration of the obtained results, Fig. 5.15 and Table 5.7 present the distributions and descriptive statistics of the lifetime of each type of performance issue across all apps.

We can notice that the medians of lifetimes across issues types range from about 1.5 days (*UseSparseArrays*) to about 56 days (*FloatMath*), with very high standard deviations, which range from about 130 days (*UseValueOf*) to about 324 days (*FloatMath*). Having

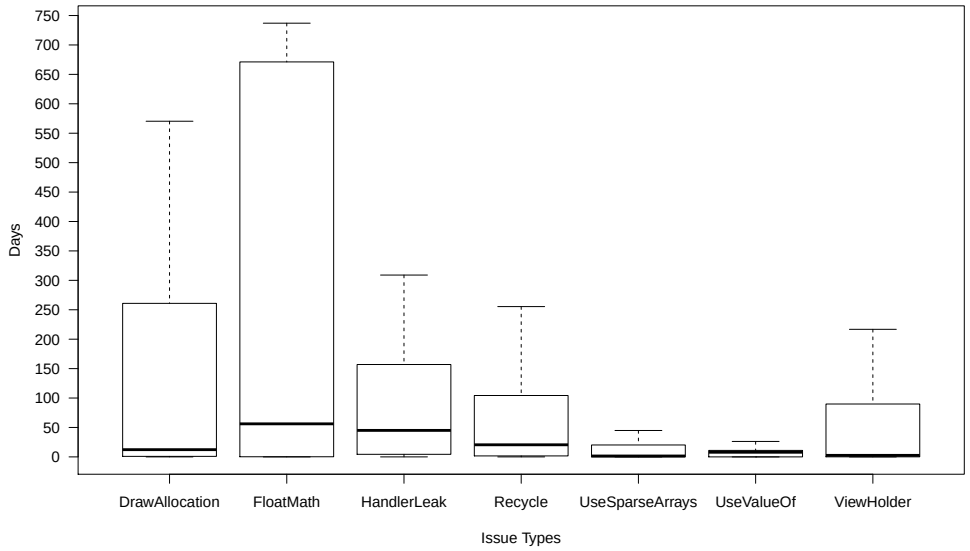


FIGURE 5.15: Lifetime of each type of performance issues (outliers are not shown to help readability).

high standard deviations provides an indication that the lifetime of statically-detectable performance issues can vary across apps and projects.

TABLE 5.7: Descriptive statistics for the lifetime (in days) of each type of performance issues per app (SD = standard deviation, CV = coefficient of variation).

Issue type	**Min.**	**Max.**	**Median**	**Mean**	**SD**	**CV**
DrawAllocation	0.0008	865	12	139	204	147
FloatMath	0.0009	737	56	294	324	110
HandlerLeak	0.0001	1221	45	169	289	171
Recycle	0.0001	961	20	106	183	173
UseSparseArrays	0.0008	1147	1	96	220	230
UseValueOf	0.0011	882	8	50	131	263
ViewHolder	0.0022	833	3	107	209	195

Results of the pairwise comparisons performed by the Dunn's procedure are reported in Table 5.8. Performance issues of type *DrawAllocation*, *FloatMath*, *HandlerLeak*, and *Recycle* have significantly longer lifetimes with respect to both *UseSparseArrays* and *UseValueOf*. Moreover, in our dataset *HandlerLeak* issues also have significantly longer lifetimes with respect to *ViewHolder* issues.

Table 5.9 reports the fitted distributions, their parameters, and the p-values of the KS test for each type of considered Android performance issue. The bold values represent the case when there is a possible distribution fitting (*i.e.,* p-values > 0.05). As shown in Table 5.9, *HandlerLeak* issues follow a *weibull* and *gamma* distribution with p-`value`

TABLE 5.8: Results of the Dunn's post-hoc analysis for comparing duration distributions (p-values are in parenthesis, statistically significant p-values are shown in bold and marked with *).

Col Mean- Row Mean	**DrawAlloc.**	**FloatM.**	**HandlerLeak**	**Recycle**	**UseSparseA.**	**UseValueOf**
FloatMath	-1.064105 (0.5746)					
HandlerLeak	-1.661472 (0.4348)	-0.588860 (0.8339)				
Recycle	0.110759 (0.4559)	1.431713 (0.4567)	2.214711 (0.1473)			
UseSparseA.	3.374780 **(0.0052*)**	4.729559 **(0.0000*)**	5.558179 **(0.0000*)**	4.159201 **(0.0003*)**		
UseValueOf	3.983202 **(0.0005**)**	5.572733 **(0.0000*)**	6.594601 **(0.0000*)**	5.317560 **(0.0000*)**	0.267410 (0.7892)	
ViewHolder	1.624728 (0.4169)	2.636551 (0.0503)	3.209208 **(0.0087*)**	1.790917 (0.3665)	-1.248851 (0.5293)	-1.555902 (0.4191)

> 0.05, whereas *ViewHolder* issues fit the *weibull* and *lognormal* distribution. Furthermore, *DrawAllocation* issues fits the *gamma* distribution, *Recycle* issues fit the *weibull* distribution and *UseSparseArrays* issues fit a *lognormal* distribution. *FloatMath* and *UseValueOf* performance issues do not fit any considered distribution.

TABLE 5.9: Results (p-values) of the KS test fitting the lifetime of different types of Android performance issue to different distribution models.

Issue type	**Norm**	**Exp**	**Weibull**	**Gamma**	**Lognorm**	**Cauchy**
DrawAllocation	< 0.05	< 0.05	< 0.05	**0.17**	< 0.05	< 0.05
FloatMath	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05
HandlerLeak	< 0.05	< 0.05	**0.33**	**0.13**	< 0.05	< 0.05
Recycle	< 0.05	< 0.05	**0.13**	< 0.05	< 0.05	< 0.05
UseSparseArrays	< 0.05	< 0.05	< 0.05	< 0.05	**0.23**	< 0.05
UseValueOf	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05
ViewHolder	< 0.05	< 0.05	**0.28**	< 0.05	**0.29**	< 0.05

Fig. 5.17 highlights the Cumulative Distribution Function (CDF) of few types of performance issues, in which the actual CDF is represented through a red line and the theoretical CDF is shown as a blue line. Fig. 5.17 reports that the actual CDF for *HandlerLeak* issues follows the Weibull distribution. Instead, *UseSparseArrays* best fits the lognormal distribution and *DrawAllocation* fits a Gamma distribution. These plots indicate that those three types of issues tend to be resolved either (i) immediately (in all cases there is at least a 50% probability of resolving them within few days) or (ii) very late during the lifespan of the project (*e.g.,* approximately 30% of *DrawAllocation* issues are resolved after 200 days).

We also analyzed the relationship between the type of issues frequently resolved and the apps' number of downloads. Results are shown in Fig. 5.16, and indicate that `UseValueOf` issues exhibit a high number of resolutions in apps having a relatively medium number of downloads. While, as we discussed in RQ_0 (Section 5.5) we noticed

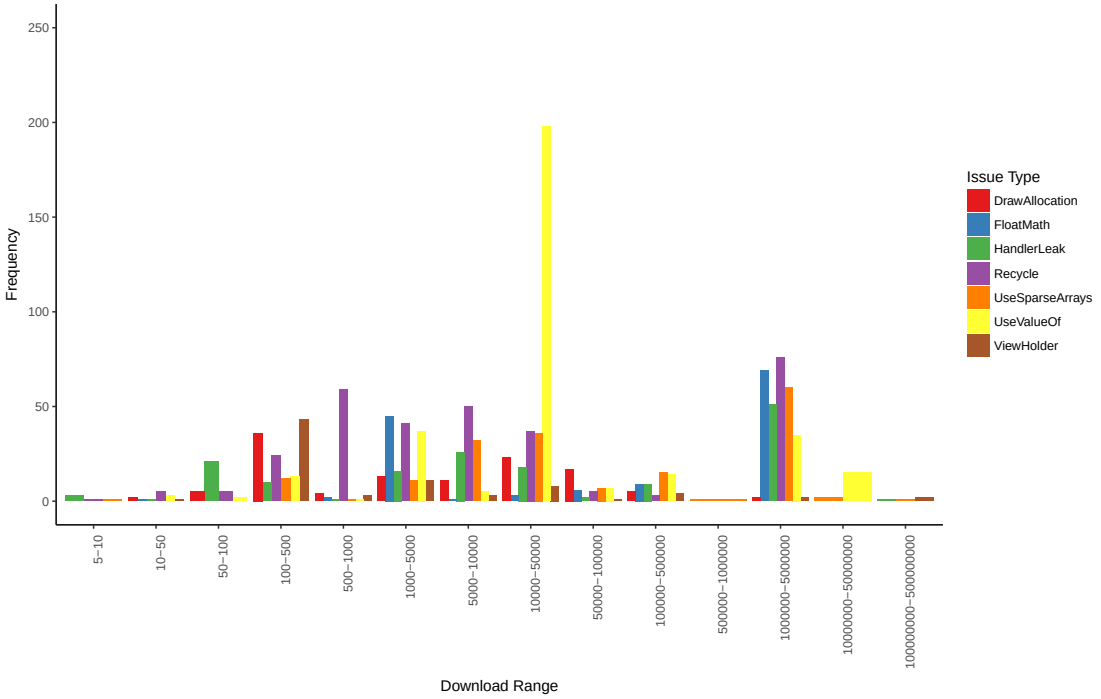


FIGURE 5.16: RQ_3: Relationship between types of resolved issues and number of apps' downloads.

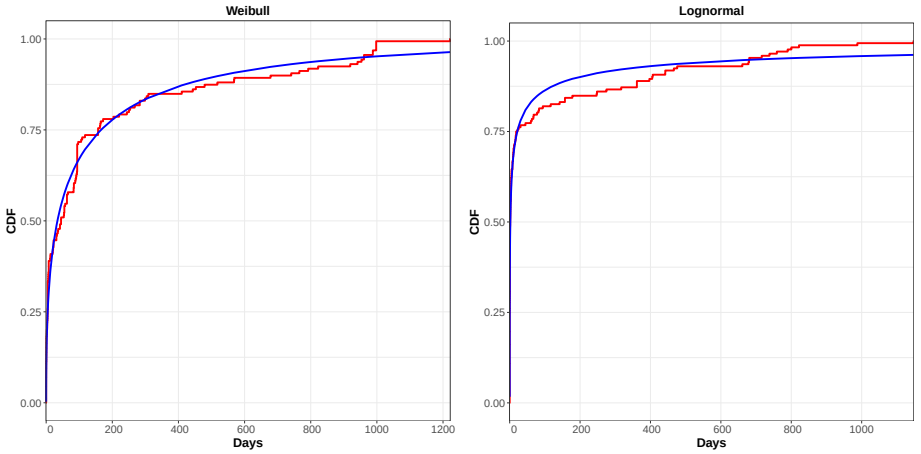
a high number of issues of this type being introduced, we can also see that they are removed after a while.

> ***Summary – RQ2.3*** – Statically-detectable Android performance issues tend to remain in the project for a relatively long period. On average (median) they are resolved after more than 137 (21) days during the lifetime of the whole project. When resolved, there is a 50% probability that a statically-detectable Android performance issue is resolved before 2 months after its introduction in the app's repository.

5.6 RQ2.4 Results – Documented Resolutions of Android Performance Issues

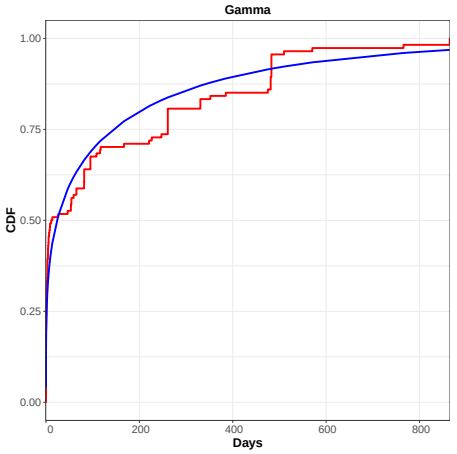
5.6.1 Data Analysis (RQ2.4)

RQ2.4 is about *documented* resolutions of Android performance issues. In this context, a documented resolution of a performance issue is a special type of issue resolution where the developer performing the resolution is consciously improving the performance of the app. The starting point of our analysis is the set of issue-resolution commits, which we collected during the data extraction.



(a) HandlerLeak Issue (Best Fit).

(b) UseSparseArrays Issue (Best Fit).



(c) DrawAllocation Issue (Best Fit).

FIGURE 5.17: Empirical and theoretical CDFs.

In this phase, we consider GitHub *commit messages* as indicators of the actual documented intention of the developer. Then, for each type of Android performance issue, we define a dedicated set of terms for identifying which commit messages are dealing with that issue. Clearly, the identification of the sets of terms is a key factor for the success of this phase. We follow a semi-systematic approach for the identification of the terms related to each Android performance issue: (i) we extract an initial set of terms based on the name of each performance issue (*e.g., DrawAllocation* is composed of the keywords `draw` and `allocation`) and (ii) we manually extract relevant terms from the description of each type of performance issue in the official *Android Lint* web page. For example, the description provided by *Android Lint DrawAllocation* is provided below and it leads to the identification of the following terms: $< Draw, Allocate, Layout, User\ Interface, UI, Memory >$.

You should avoid allocating objects during a drawing or layout operation.

TABLE 5.10: Regular expressions for identifying documented Android performance issues.

Type of issue	**Keywords**
DrawAllocation	[Dd]raw*, [Aa]lloca*, [Ll]ayout, UI, [Mm]emory
Recycle	[Cc]ursor*, [Rr]ecycle*, [Tt]ypedArray*, [Vv]elocityTrack*
ViewHolder	[Vv]iewHolder*, [Aa]dapter*, [Ll]istView, [Ss]crol*
HandlerLeak	[Hh]andler*, [Ww]eakReferenc*, [Mm]essageQueue*, [Ll]eak*
UseSparseArrays	[Ss]parseArray*, [Hh]ashMap*
UseValueOf	[Vv]alue*, [Dd]ouble, [Ll]ong
FloatMath	[Ff]loatMath
Miscellaneous	[Ll]int, [Ww]arn*, [Pp]erf*

> *These are called frequently, so a smooth UI can be interrupted by garbage collection pauses caused by the object allocations. The way this is generally handled is to allocate the needed objects up front and to reuse them for each drawing operation. Some methods allocate memory on your behalf (such as Bitmap.create), and these should be handled in the same way.*

By applying the afore mentioned procedure to each type of Android performance issue, we identify the regular expressions shown in Table 5.10. It can also be noted from Table 5.10 that in last row of the table, we define an additional set of regular expressions containing few more general performance-related terms. We add this set of regular expressions in order to cover also those cases where the developers are addressing performance issues, but are not referring to any specific *Android Lint* check.

To answer RQ2.4, we present and discuss (i) the frequency of documented resolutions of Android performance issues within the whole dataset of Android apps, (ii) the distribution of documented resolutions of performance issues across the seven types of Android performance issues, and (iii) the distribution of the *code churns* associated to each of the 1,314 PRI[14]-resolving commits across each type of Android performance issue. Code churns refer to the total number of changed lines of code in a commit (either added, removed or updated) [90]. In this study, we rely on code churns because (i) it is one of the most used metrics for representing the change volume between two versions of the same system [90], (ii) it can be considered as a relatively good estimator for the development effort devoted to a GitHub commit, and (iii) it can be extracted automatically with low computational effort, and (iv) the `git log` command can compute it out of the box[15].

Also, to better explain the results, we provide an example of solution for each type of Android performance issue. This allows us to build a minimal catalog of reusable

[14]PRI is an acronym for Performance-Related Issue
[15]`https://git-scm.com/docs/git-log`

solutions, which can be used by Android developers for better understanding how to resolve *Android Lint* warnings in their projects. We select the solutions in two steps. First of all, we randomly select five commits for each type of Android performance issue, where a developer documented a performance-related change. Then, we manually inspect the changes in the Java source code in each identified commit and select the most recurrent and representative one (also based on the description of the corresponding *Android Lint* check). We present and discuss in details all the representative solutions we identified during manual inspection. We believe that developers can exploit this emerging collective knowledge for solving the performance issues of their mobile apps in a more effective manner.

5.6.2 Results (RQ2.4)

Starting from the 1,314 commits where performance issues have been resolved (either intentionally or not), we identified the subset of commits whose commit message mentions Lint-related performance issues via the keyword-based strategy described in Section 5.1.3. For each type of Android performance issue, Table 5.11 reports the number of commits in which (i) developers explicitly document the resolution of a given type of Android performance issue (second column) and (ii) developers just generically mention that they resolved a performance issue (third column). We add this Miscellaneous set of keywords in order to cover also those cases where the developers are addressing performance issues, but are not referring to any specific Android Lint check (these Miscellaneous set of keywords are shown in the last line of Table 5.10).

TABLE 5.11: Number of documented performance issue resolutions.

Issue type	**#Commits**	**#MiscellaneousCommits**	**Total**
DrawAllocation	12 (8.4%)	1 (0.7%)	13 (9.1%)
FloatMath	20 (13.9%)	2 (1.4%)	22 (15.3%)
HandlerLeak	17 (11.9%)	1 (0.7%)	18 (12.6%)
Recycle	48 (33.6%)	9 (6.3%)	57 (39.9%)
UseSparseArrays	2 (1.4%)	4 (2.8%)	6 (4.2%)
UseValueOf	1 (0.7%)	9 (6.3%)	10 (7.0%)
ViewHolder	16 (11.1%)	1 (0.7%)	17 (11.9%)
Total	116	27	143

Firstly, we observe that the resolution of every type of performance issue has been explicitly documented at least more than once, summing up to a total of 143 (10.88%) commits out of 1,314. Table 5.11 indicates that *Recycle* issues are more often documented (*i.e.,* 48 times), followed by *FloatMath* (20 times), *HanderLeak* (17 times), and *ViewHolder* (16 times). This may be considered as an indications of the extent to which

developers are aware of issues, which could be possibly related to the actual usage of static analysis tools like Android Lint.

There are 27 commits notes identified after applying the miscellaneous regular expressions (see Table 5.10). Among them, 9 commits are related to *Recycle* and *UseValueOf* issues, respectively.

In Table 5.12 we provide an example of commit for each type of performance issue and three examples of commits matching the generic regular expression. We also check how many of the 143 commits are exclusively related to the resolution of performance issues, and how many are tangled commits also related to other changes (*e.g.,* the implementation of a feature, etc.). It is interesting to note that many of such commits refer only to performance issues resolution (*i.e.,* 117 out of 143 commits). This suggests that, when documenting Lint-related resolutions in their commit messages, Android developers tend to do dedicated "issue resolution sessions", which are 100% focused on resolving Lint-related performance issues.

TABLE 5.12: Examples of commits with documented performance issue resolution.

Category	**Repository**	**Commit ID**	**Commit message**
DrawAllocation	mchow01/FingerDoodle	15c6432	▲ *Fixed NullPointerException at FingerDoodleView.java line 67 edu.cs.tufts.mchow. FingerDoodleView.onDraw*
FloatMath	almalence/OpenCamera	b232324	▲ *Partially fixed issue with preview on Android 6 in camera2 mode. Fixed nexus naming in CC. Changed depricated FloatMath to Math.*
HandlerLeak	stdev293/battery-waster-android	7212e95	▲ *static handler to avoid potential memory leak*
Recycle	uberspot/AnagramSolver	be502aa	▲ *close cursors after using them.*
UseSparseArrays	pocmo/Yaaic	fb90f72	▲ *Use SparseArray instead of HashMap for in-memory server storage.*
UseValueOf	AmrutSai/sikuna	696873b	▲ *Made changes based on recommendations from the lint tool...*
ViewHolder	chaosbastler/opentraining	3052768	▲ *Implemented ViewHolder-Pattern for ExerciseImageListAdapter (used for CreateExerciseActivity). Awesome performance improvements.*
Other(UseValueOf)	Anasthase/TintBrowser	d6f86cd	▲ *Correct lint warnings.*
Other(Recycle)	shlusiak/Freebloks-Android	ec65540	▲ *fix lint warnings*
Other(ViewHolder)	LukeStonehm/LogicalDefense	931e2039	▲ *FIXED: #re-use view if already exists, this will increase performance a little*

When looking at the minimum number of lines of code to resolve issues (see Table 5.13), the amount of code to be written to resolve some performance issues is fairly limited. For example, *Recycle*, *UseValueOf*, *FloatMath*, and *UseSparseArrays* can be resolved by changing a limited number of lines of code (i.e., up to 5 in 14.68% of cases). This is

because these kinds of problems mainly deal with the usage of primitives data types and thus their resolution can be performed with a very limited number of changes. For example, a *Recycle* issue can be resolved by closing a cursor `c` by invoking the *c.close()* or through recycling a TypedArray `t` *i.e.*, by invoking the *t.recycle()* method. Moreover, issues such as *UseValueOf* can be resolved by calling the `valueOf` factory method instead of directly calling the constructor of a wrapper class like `Integer(int)`. With a minimum number of 16 and 33 changed LOCs, *DrawAllocation* and *HanderLeak* issues seem to be not trivially resolvable. For example, resolving an *HandlerLeak* usually implies the creation of a new static handler and setting a weak reference to it.

TABLE 5.13: Descriptive statistics for the LOCs for resolving each type of performance issue (SD = standard deviation, CV = coefficient of variation).

Issue type	Min.	Max.	Median	Mean	SD	CV
DrawAllocation	16	170	170	138.8	59.9	0.4
FloatMath	2	25	21	19.4	7.5	0.4
HandlerLeak	33	936	810	639.8	303.5	0.4
Recycle	1	598	578	328.2	277.5	0.8
UseSparseArrays	5	1,313	21	446.3	671.4	1.5
UseValueOf	2	148	4	53.3	65.6	1.2
ViewHolder	3	126	54	49.4	29.5	0.6

In addition, we provide a catalog of solutions (given later in this RQ) that will surely help developers to quickly fix the issue and save their time.

```
±¼   @@ -110,16 +110,21 @@ private void insertOrUpdate(String table, String nullColumnHack,
110  110
111  111                        Cursor c = db.query(table, projection, whereClause, whereArgs,
112  112                                null, null, null, "0,1");
113      -                      if (c.moveToFirst()) {
114      -                              String[] id = { c.getString(c
115      -                                      .getColumnIndexOrThrow(idColumnName)) };
116      -                              values.put(FACT_MATCH_DATA_Entry.COLUMN_NAME_TIMESTAMP,
117      -                                      DBSyncService.dateParser.format(new Date()));
118      -                              db.update(table, values, idColumnName + "=?", id);
119      -                      } else {
120      -                              db.insert(table, nullColumnHack, values);
     113  +                      try {
     114  +                              if (c.moveToFirst()) {
     115  +                                      String[] id = { c.getString(c
     116  +                                              .getColumnIndexOrThrow(idColumnName)) };
     117  +                                      values.put(FACT_MATCH_DATA_Entry.COLUMN_NAME_TIMESTAMP,
     118  +                                              DBSyncService.dateParser.format(new Date()));
     119  +                                      db.update(table, values, idColumnName + "=?", id);
     120  +                              } else {
     121  +                                      db.insert(table, nullColumnHack, values);
     122  +                              }
     123  +                      } finally {
     124  +                              if (c != null)
     125  +                                      c.close();
     126  +                              ScoutingDBHelper.getInstance().close();
121  127                        }
122      -                      ScoutingDBHelper.getInstance().close();
123  128                }
124  129        }
```

FIGURE 5.18: Example of resolving of the *Recycle* issue.

```
4 src/de/saschahlusiak/freebloks/game/FreebloksActivity.java
@@ -537,7 +537,8 @@ boolean restoreOldGame() throws Exception {
537 537                         p.unmarshall(bytes, 0, bytes.length);
538 538                         p.setDataPosition(0);
539 539                         bundle = p.readBundle(FreebloksActivity.class.getClassLoader());
540     -
    540 +                       p.recycle();
    541 +
541 542                         deleteFile(GAME_STATE_FILE);
542 543
543 544                         if (readStateFromBundle(bundle)) {
@@ -565,6 +566,7 @@ private void saveGameState(String filename) {
565 566                         fos.write(p.marshall());
566 567                         fos.flush();
567 568                         fos.close();
    569 +                       p.recycle();
568 570                 } catch (Exception e) {
569 571                         e.printStackTrace();
570 572                 }
```

FIGURE 5.19: Example2 of resolving of the *Recycle* issue.

Recycle. It can be seen in code snippet (Fig. 5.18) that developers use *cursor.close()* at the end (at line 174) to close the database query cursor properly and this changes in source code led to resolution of `Recycle` issues from the file. Whereas, in other (Recycle) category, `Recycle` issue was solved in the project *shlusiak/Freebloks-Android* by recycling the resource, *i.e.*, by calling the *p.recycle()* method (see lines 540 and 569 in Fig. 5.19)

```
76  82          public View getView(final int position, View convertView, ViewGroup parent) {
77      -               //View vi = convertView;
78      -               View vi = mInflater.inflate(de.skubware.opentraining.R.layout.exercise_image_list_row, null);
79  83
80      -               ImageView imageView = (ImageView) vi.findViewById(R.id.exercise_image);
81      -               imageView.setImageBitmap(getBitmap(position));
82      -               imageView.setOnClickListener(new OnClickListener(){
83      -                       @Override
84      -                       public void onClick(View v) {
85      -                               ShowImageDialog.showImageDialog(getBitmap(position), mContext);
86      -                       }
87      -               });
88      -
89      -
90      -               return vi;
    84  +               ViewHolderItem viewHolder;
    85  +
    86  +               if(convertView==null){
    87  +               // inflate the layout
    88  +               convertView = mInflater.inflate(de.skubware.opentraining.R.layout.exercise_image_list_row, null);
    89  +
    90  +
    91  +               // set up the ViewHolder
    92  +               viewHolder = new ViewHolderItem();
    93  +               viewHolder.imageViewItem = (ImageView) convertView.findViewById(R.id.exercise_image);
    94  +
    95  +               // store the holder with the view.
    96  +               convertView.setTag(viewHolder);
    97  +
    98  +           }else{
    99  +               // avoided calling findViewById() on resource each time, use the viewHolder
   100  +               viewHolder = (ViewHolderItem) convertView.getTag();
   101  +           }
```

FIGURE 5.20: Example of resolving of the `ViewHolder` issue.

ViewHolder. To resolve this issue developers should implement `ViewHolder` pattern in *getView()* callbacks. For example, it can be noted from the manual observation that developers specially implemented `ViewHolder` Pattern for ExerciseImageListAdapter to resolve the `ViewHolder` issue as shown in Fig. 5.20 (an example from our dataset). The idea is to reuse earlier recycled list items. It prevents the inflation of list items layout when there are recycled items available for reuse (Lines 99-100). When the list of items is created for the first time, the references to inner view object are identified and stored in a particular data structure for reuse (Lines 95-96). The main advantages of using the `ViewHolder` pattern are that (1) it can save computation for inflation of list items layout by reducing *findViewById()* calls and also invokes less inner view retrieval computations, and (2) it is memory efficient for building new list items. Furthermore, to reduce the impact of frequently invoked callbacks in *getView()* implementation, developers should use this kind of efficient design.

```
32 +      private static class SinksControlActionHandler extends Handler {
33 +              private WeakReference<BatteryWasterActivity> mActivityRef;
34 +
35 +              public SinksControlActionHandler(BatteryWasterActivity activity) {
36 +                      super();
37 +                      setActivityReference(activity);
38 +              }
39 +
40 +              public void setActivityReference(BatteryWasterActivity activity) {
41 +                      mActivityRef = new WeakReference<BatteryWasterActivity>(activity);
42 +              }
43 +
44 +              public void handleMessage(Message msg) {
45 +                      BatteryWasterActivity activity = mActivityRef.get();
46 +                      if (activity==null) {
47 +                              return;
48 +                      }
49 +
50 +              // process incoming messages
51 +              switch (msg.what) {
52 +                      case ACTION_START_ALL:
53 +                              activity.startWasting();
54 +                              break;
55 +                      case ACTION_LIGHT_ON:
56 +                              activity.startLight();
57 +                              break;
58 +                      case ACTION_LIGHT_OFF:
59 +                              activity.stopLight();
60 +                              break;
61 +                      case ACTION_STOP_ALL:
62 +                      default:
63 +                              activity.stopWasting();
64 +                              break;
65 +              }
```

FIGURE 5.21: Example of resolving of the `HandlerLeak` issue.

HandlerLeak. To resolve the `HandlerLeak` type of issues, developers declared the handler as static class. It can be clearly shown from Fig. 5.21 (an example from our dataset), that developers create a new static class for the handler to avoid the potential

memory leaks. Moreover, developers also used a `WeakReference` to outer class and pass the object to instantiate a handler.

FIGURE 5.22: Example of resolving of the `UseSparseArrays` issue.

UseSparseArrays. We manually analyzed the source code of several projects to know how the `UseSparseArrays` issue type is resolved, we observed that for better performance of memory server storage, developer removed HashMaps and used `UseSparseArrays` instead. One of the sample documented example can be shown in the Fig. 5.22(from our dataset) where the `UseSparseArrays` type of issue is resolved from the project. However, `UseSparseArrays` are assumed to be more memory efficient and trigger less garbage collection as compared to its counterpart `HashMaps` with no key impact on operations performance of maps. Moreover, `SparseArrays` allocate less memory as compared to `HashMaps` [22].

DrawAllocation. Related to `DrawAllocation` issue category, we consider the code snippet as shown in Fig. 5.23 (an example from our dataset) to analyze the solution.

From our manual analysis, we observed that developer removed the Paint object from *onDraw()* function (*i.e.,* a background Paint object is created each time when the draw operation takes place and memory is allocate every time) as shown in Fig. 5.23. To resolve this issue, developer pre-allocate the background Paint (at line 18) upfront (*i.e.,* outside from *onDraw()* function) and reuse it instead, which will prevent from the UI lag since memory will not allocate at each time when *onDraw()* or *Layout()*function is called. Therefore, it is a good suggestion for developers to allocate new object before the draw or layout operation [3].

```
13 14         public FingerDoodleView (Context context)
14 15         {
15 16                 super(context);
16 17                 doodle = new Doodle();
   18 +              backgroundPaint = new Paint();
17 19                 getHolder().addCallback(this);
18 20                 thread = new DrawingThread(getHolder(), this);
19 21                 setFocusable(true);
@@ -25,10 +27,7 @@ public DrawingThread getDrawingThread()
25 27         }
26 28
27 29         @Override
28    -        public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
29    -                // TODO Auto-generated method stub
30    -
31    -        }
   30 +        public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {}
32 31
33 32         @Override
34 33         public void surfaceCreated(SurfaceHolder holder)
@@ -64,11 +63,12 @@ public void surfaceDestroyed(SurfaceHolder holder)
64 63         @Override
65 64         public void onDraw (Canvas canvas)
66 65         {
67    -                Paint backgroundPaint = new Paint();
68 66                 backgroundPaint.setColor(doodle.backgroundColor);
69    -                canvas.drawPaint(backgroundPaint);
70    -                for (DoodlePath p: doodle.paths) {
71    -                        canvas.drawPath(p.path, p.paint);
   67 +                if (canvas != null) {
   68 +                        canvas.drawPaint(backgroundPaint);
```

FIGURE 5.23: Example of resolving of the `DrawAllocation` issue.

```
25 src/com/almalence/plugins/capture/panoramaaugmented/AugmentedRotationListener.java
@@ -18,14 +18,11 @@ Portions created by Initial Developer are Copyright (C) 2013
18 18
19 19   package com.almalence.plugins.capture.panoramaaugmented;
20 20
21    - import android.annotation.SuppressLint;
22 21   import android.hardware.Sensor;
23 22   import android.hardware.SensorEvent;
24 23   import android.hardware.SensorEventListener;
25 24   import android.hardware.SensorManager;
26    - import android.util.FloatMath;
27 25
28    - @SuppressLint("FloatMath")
29 26   public class AugmentedRotationListener implements SensorEventListener
30 27   {
31 28         public interface AugmentedRotationReceiver
@@ -180,7 +177,7 @@ private void filterGravity(final float[] accel, final float[] gravity)
180 177                int idx = acc_filt_idx;
181 178                for (int i = 0; i < ACC_FILT_LEN; ++i)
182 179                {
183    -                       float g = FloatMath.sqrt(acc_filter[idx][0] * acc_filter[idx][0] + acc_filter[idx][1] * acc_filter[idx][1]
   180 +                       float g = (float)Math.sqrt(acc_filter[idx][0] * acc_filter[idx][0] + acc_filter[idx][1] * acc_filter[idx][1]
184 181                                        + acc_filter[idx][2] * acc_filter[idx][2]);
185 182
186 183                        // the farther the measured g from 9.81 the less the weight in a
@@ -281,7 +278,7 @@ public void magnetoFunction(final SensorEvent event)
281 278                        this.filt_magnetic[i] = event.values[i] = (this.filt_magnetic[i] + event.values[i]) / 2;
282 279                }
```

FIGURE 5.24: Example of resolving of the `FloatMath` issue.

FloatMath. For the `FloatMath` issue type, instead of `FloatMath` declaration, developers should use `Math` in the source code to resolve this issue. It can be noted from the Fig. 5.24, that after depreciated from `FloatMath` to `Math` (as developer wrote in commit note), the `FloatMath` issue is resolved. In the following we report the commit message provided by the developer when resolving this issue.
"Partially fixed issue with preview on Android 6 in camera2 mode. Fixed nexus naming in CC. Changed deprecated FloatMath to Math." This issue type focuses on the primitives data types, so developers can get rid of this issue by using Math in the source code.

FIGURE 5.25: Example of resolving of the `UseValueOf` issue.

UseValueOf. Regarding the `UseValueOf` type of issues, following is the commit message developer wrote after resolving the issue of Fig. 5.25 (an example from our dataset).
"Made changes based on recommendations from the lint tool... and removed unused strings. Preferences have been renamed to Settings. This also no longer appears in the ActionBar on Honeycomb and above... following recommendations at the Android design site. Changed Settings display based on these recommendations also. Made plenty of changes in BusStopDatabase and SettingsDatabase to follow good practices and possible performance enhancements."
As shown in Fig. 5.25, the developer used a call to `valueOf` (at line 310) to resolve this issue.

Summary — RQ2.4 – A total of 143 (10.88%) commits out of 1,314 document the resolution of PRIs. Out of those 143 commits, the resolution of *Recycle* issues is the most documented (57, 39.9%) and the resolution of *UseSparseArrays* issues is the least documented (6, 4.2%). A catalog of manually-extracted resolutions of performance-related issues is also provided.

5.7 Discussion

In the following, we firstly provide a detailed analysis about how the results of the research questions of this study are linked together (Section 5.7.1). Then, we discuss the implications that the results of our study have for developers (Section 5.7.2) and researchers (Section 5.7.3).

5.7.1 Summary of the study results

Table 5.14 presents an overview of the main results discussed in the previous sections. Specifically, for each type of considered performance issue (first column) we report: its priority as reported in the Android Lint documentation (second column), its frequency in our dataset (third column), the portion of resolved issues (fourth column), its median lifetime (fifth column), the median of the lines of code for resolving it (sixth column), and median of documented resolutions (seventh column).

TABLE 5.14: Combination of the obtained results.

Issue type	**Linter priority**	**Frequency (%)**	**Resolutions (%)**	**Lifetime (median)**	**LOCs (median)**	**Documented (%)**
DrawAllocation	9/10	237 (9.84%)	118 (4.90%)	12.2779	170	13 (9.1%)
Recycle	7/10	550 (22.84%)	310 (12.87%)	20.5430	578	57 (39.9%)
ViewHolder	5/10	180 (7.47%)	79 (3.28%)	2.5965	54	17 (11.9%)
UseValueOf	4/10	549 (22.79%)	331 (13.74%)	7.8848	4	10 (7.0%)
UseSparseArrays	4/10	376 (15.61%)	180 (7.47%)	1.4995	21	6 (4.2%)
HandlerLeak	4/10	320 (13.28%)	160 (6.64%)	45.0079	810	18 (12.6%)
FloatMath	3/10	188 (7.80%)	136 (5.64%)	56.1576	21	22 (15.3%)

By looking at the combined results we can draw a number of interesting insights. Firstly, *DrawAllocation* has the highest priority among the considered performance issues, but it is resolved in roughly half of its occurrences (similarly to the resolutions of the other types of issues). We argue that this phenomenon can be explained by the relatively high number of lines of code needed for resolving it (170) and by the fact that it is difficult for developers to precisely assess when drawing or layout operations are taking place at run-time. *DrawAllocation* issues may be an impactful scientific target for researchers working on automatic refactoring tools because of the high priority of this type of issue, which can be considered as a proxy of the severity of the impact that this type of issues can have on the performance of the app.

Recycle is the most frequent type of performance issue (550 occurrences) and they are resolved in more than half of the cases (310). In principles, resolving this type of issue mean changing only one line of code, *i.e.,* adding a call to the `recycle()` method of the resource being used; however, the median number of lines of code for resolving *Recycle*

issues is much higher (578 LOCs). At the time of writing, we do not have a precise explanation of this phenomenon and further investigation is needed.

We can observe that *UseValueOf* issues are resolved in a relatively short period (about 8 days) compared to other types of issues. This phenomenon may be explained by the fact that the median number of lines of code for resolving those issues is only 4. Indeed, by checking the official documentation of Android Lint, *UseValueOf* issues can be resolved simply by calling the `valueOf()` method of wrapper classes (*e.g.,* `Integer`) instead of directly calling their constructor. Nevertheless, *UseValueOf* is the most recurrent type of performance issues in our dataset (549 occurrences) and their resolution is documented only in 10 cases. We argue that since *UseValueOf* issues are mostly about wrapper classes of primitive types, developers perceive this type of issues as not impactful with respect to the overall performance of the app.

While resolving *FloatMath* issues requires a relatively small effort in terms of lines of code (median = 21), their lifetime is the highest across all performance issues (median = 56 days). One possible reason for this type of issue to remain for such long periods is that Android Lint rates it with a very low priority (3/10), therefore developers tend to defer their resolutions more than they do for other types of issues.

Finally, we notice that, in general, the considered performance issues tend to be unresolved and persist for long periods. We argue that this phenomenon can be explained by two facts: (i) in general the priority of the considered issues is quite low (only *DrawAllocation* and *Recycle* have 9/10 and 7/10 as priorities, respectively, while the others have a priority level lower or equal to 5/10) and (ii) Android Lint only raises a *warning* when it detects one of the issues in the source code, leaving to the developer the choice to either solve or ignore the detected issue without blocking his/her development flow. Under this perspective, as we will also discuss in Section 5.7.3, it will be fundamental to empirically characterize the actual impact of Android Lint issues on the overall performance of the app. The results of such an assessment will help developers in taking better informed decisions about whether and when Android Lint performance issues should be managed or can be safely ignored to have Android apps with good-enough performance levels.

5.7.2 Implications for Developers

Based on the obtained results, in the following we summarize how such results could guide developers in better handling Android performance issues.

D1 – Developers are generally aware of performance issues detected by Android Lint, but there is space for improvement. The results of R_0 are revealing

that in general developers are not injecting an extremely high number of performance issues in their apps ($min = 0$, $max = 171$ issues per app). Nevertheless, within the limits of the Android Lint accuracy, the 2,408 performance issues we identified in this study can be seen as *missed chances* for improving the performance of Android apps.

Some of the detected issues are not only about performance, but are indicators of (i) poor programming practices (*e.g., UseSparseArrays*), which may hinder the overall maintainability of the app and (ii) memory leaks (*e.g., HandlerLeak*), which may potentially lead to the OS forcefully closing the app to recollect all the (mis-)used resources. Those are risks that today's Android developers cannot afford in a crowded and fiercely competitive market as the Google Play store.

D2 – Do not treat all types of Android performance issues equally, but (re)use previous experience to prioritize and fix them. Indeed, Android Lint checks are organized in different levels of priority and severity in order to guide the developer in prioritizing them. Also, when answering RQ2.4 it emerged that *resolving Android performance issues demands different levels of effort.* For example, a *Recycle* issue can be simply resolved by a call to the *recycle()* method of the used recyclable resources (*e.g.,* `TypedArray`); such a call can heavily improve the performance of the app, since it frees the potentially large resource before the execution of the garbage collectors. Differently, the resolution of a *HandlerLeak* issue is usually implemented by (i) creating a static inner class for the handler, (ii) having a weak reference in the outer class pointing to the outer class, and (iii) always using the weak reference when referring to the outer class.

One possibility, to support developers, is that researchers (see Section 5.7.3) develop linters that prioritize warnings based on some knowledge. At the same time, as a lesson for developers, this chapter attempted to distill and discuss a catalog of solutions for the various kinds of issues, based on what developers have done in the studied apps. Such a catalog is discussed in our RQ_4 (Section 5.6).

Generally speaking, we advice developers to *establish different priority levels to different performance issues* depending on app's users' needs, project characteristics, and available resources and to build a prioritization model according to them. In this context, variations of the Weighted Shortest Job First (WSJF [77]) model may be a good starting point (WSJF model is utilize to sequence jobs such as capabilities, features and Epics to maximize the economic benefit. It is computed as the ratio between Cost of Delay (CoD) to job size). At the same time, developers should, (once again, possibly with the help of environments developed by researchers) build a knowledge base of previously adopted solutions, in order to apply them when appropriate.

D3 – Performance issues may be harmless when they occur in isolation, while they can be particularly concerning when they occur in combination. If we look at each Android performance issue in isolation, its overhead at run-time may be minimal in terms of computational resources demanded by the app. However, in this study we also observed that (i) in many cases issues tend to accumulate over time (*e.g.,* STICK is the most recurrent evolution pattern) and (ii) apps can exhibit many performance issues at the same time (*e.g.,* the app ChinaTVGuide exhibited 25 performance issues for more than 4 years).

Developers can alleviate these risks by *continuously monitoring* the number and types of Android performance issues in their code base during the whole project. This practice can also help in mitigating the well-known problem that developers are less likely to fix linter warnings on legacy code [36, 56]. Indeed, if the monitoring (and corresponding resolution) of performance issues is performed continuously along other development activities, then it will be unlikely that the code base will accumulate a high number of issues, thus avoiding the need to fix them on previously developed code. In this context, one concrete possibility is to configure linters (*e.g.,* in a Continuous Integration pipeline) so that they warn developers, by failing a build, when too many potential performance issues have been introduced in a commit, or in a sequence of consecutive commits. Last, but not least, it can be advisable to complement linters with performance regression tests.

The *identified evolution patterns can be used by Android developers as a common, shared, and aggregated viewpoint* for guiding maintenance activities and keeping under control the health of their apps from the perspective of performance issues. The integration of the tool chain discussed in Section 5.1.3 and visual dashboards may be a vital instrument for developers for early identifying dangerous evolution patterns (*e.g.,* issue-rich feature) and immediately fix them.

5.7.3 Implications for Researchers

In the following, we phrase the implications for researchers as perspective Research Questions, labeled as PRQ — which can be addressed in future research work.

PRQ$_1$ – Why some performance issues tend to remain in Android apps? In our study, Android performance issues tend to remain for many days within the repository; the average duration ranges from 53.95 days for *UseValueOf* to 293.9 days for *FloatMath* issues. We suspect that those long durations are due to the fact that performance issues raised by Android Lint are deemed to be not dangerous by developers.

We still do not have evidence about whether this perception is true or not, but objectively investigating *how developers perceive the issues raised by Android Lint* is a relevant research direction. An initial investigation is reported in [56], where the main causes for developers not to use Android Lint have been empirically extracted via interviews. There, the top three beliefs against the use of Android Lint for performance purposes are: (i) the performance issues should be handled soon (*i.e.,* before user's complains), (ii) the static analysis is not suitable to detect performance issues, and (iii) the warnings provided by Android Linter are irrelevant.

In addition to the common wisdom mentioned above, other factors strictly related to the tool itself may influence the lack of resolutions of Android performance issues. For example, given its static nature, sometimes Android Lint can produce false positives (*i.e.,* raising warnings when actually there is no issue), thus negatively impacting developers' trust in it [38]. The comprehensibility of the warnings may affect the tool's adoption: if developers find the error messages as confusing, then they deem the raised error as false [38]. Finally, developers may tend to perceive the resolution of Android performance issues time consuming with respect to the obtained gain.

The obtained results suggest that, not all warnings are considered by developers, and many of them are either false positives, or irrelevant for a given development context. While this is not entirely surprising, as it confirms findings coming from studies on general-purpose linters [43, 73, 106, 118], it suggests that, more advanced recommenders are needed. For example, as previously done for bug fixes [111], it could be interesting to learn from past changes to prioritize warning resolution.

PRQ_2 – To what extent it is possible to automatically refactor performance issues in Android apps? Almost half of the apps in our dataset (43.64%) exhibit at least one statically-detectable performance issue in their lifetime. Under this perspective, supporting developers with methods and techniques for automatically resolving performance issues is a valuable contribution. Despite the relative straightforwardness of manual resolutions of statically-detectable performance issues, the automatic resolution of some of them is far from being technically trivial. For example, the automatic resolution of *HandlerLeak*s involves extensive refactoring, where the resulting code is heavily based on weak references and object-oriented reachability. Initial steps in this direction are being already performed in the context of Android-specific energy-efficiency optimizations [45].

In this context, a relatively high number of apps (124) exhibit the *refactoring* evolution pattern, meaning that at some point of the app's lifetime the number of statically-detectable performance issues has a strong decrease, independently of the amount of changes in the source code. As researchers, we can closely investigate what is happening

during the occurrences of a refactoring pattern in order to *learn how developers are actually resolving statically-detectable performance issues*. The results of this analysis can be used as drivers for the design of methods and techniques for (semi-) automatically resolving Android performance issues in the future.

Finally, in the context of this specific study, having an automated refactoring technique will allows us to (i) automatically resolve all the unresolved issues we identified in RQ_3, (ii) submit their resolutions as pull requests in the original GitHub repositories, and (iii) measure the percentage of pull requests that are merged by app developers. All together, those activities will bring a better understanding about how Android developers consider detected and resolved performance issues in real industrial contexts.

PRQ_3 – What is the actual impact of Android performance issues at runtime? Being able to automatically resolve detected issues (see PRQ_2) opens also for the empirical assessment and measurement of the *impact in the resolution of statically-detectable Android performance issues* in terms of, *e.g.,* CPU usage, memory consumption, app's frame rate, etc.

At the time of writing, this line of research has not been explored yet and it can likely lead to an impactful contribution to the body of knowledge in the field of mobile software engineering. Indeed, recently it empirically emerged that a number of Android developers are indifferent to performance issues and they challenge their relevance and impact [56]. Clearly, providing empirical evidence about the actual impact of performance issues will help in the overall adoption of static analysis tools like Android Lint, promoting a more careful treatment of performance-related aspects of apps, and thus potentially leading to improving the apps' quality.

PRQ_4 – Are some performance issues more difficult to be resolved? As discussed when answering RQ2.3 and RQ2.4, some performance issues seem to be more difficult to be resolved, either in terms of code churn or days before the resolution. In order to better understand *why* this phenomenon is happening, we perform a preliminary analysis targeting specific subsets of issue-resolution commits in our dataset. Specifically, we firstly rank all 1,314 commits with resolved issues based on their code churn and then we select the top-10 commits in terms of LOCs for each type of performance issue. This leads to a set of 70 issue-resolving commits (10 for each type of performance issue) with very high code churn (average = 368.7, median = 249.5). At this point, we consider GitHub commit messages and conduct a content analysis session [82] on all 70 commit messages. Specifically, we categorize them according to the taxonomy of self-reported activities of Android developers proposed and empirically validated by Pascarella *et al.* [94]. The taxonomy entails a wide variety of different activities at different levels of abstraction (*e.g.,* bug fixes, functionality implementation, release management,

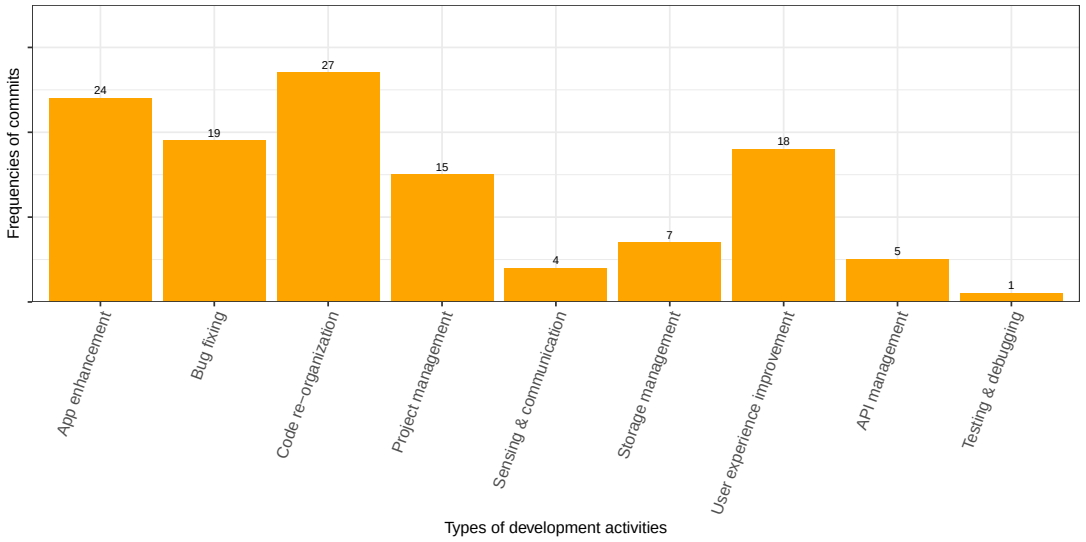
access to sensors, etc.). The taxonomy is composed of two levels, where the first layer (9 items) groups together activities with similar overall purpose (*e.g.,* app enhancement, bug fixing, API management), whereas the subcategories (49 items) in the lower level provide a finer-grained categorization [94]. In this study we focus on the top 9 level-categories only and we assigned one or more categories of development activities to each commit.

Fig. 5.26(a) and 5.26(b) present how frequently each category of Android developers' activities appears in all **commits with high code churn** in general and across the 7 types of performance-related issues, respectively.

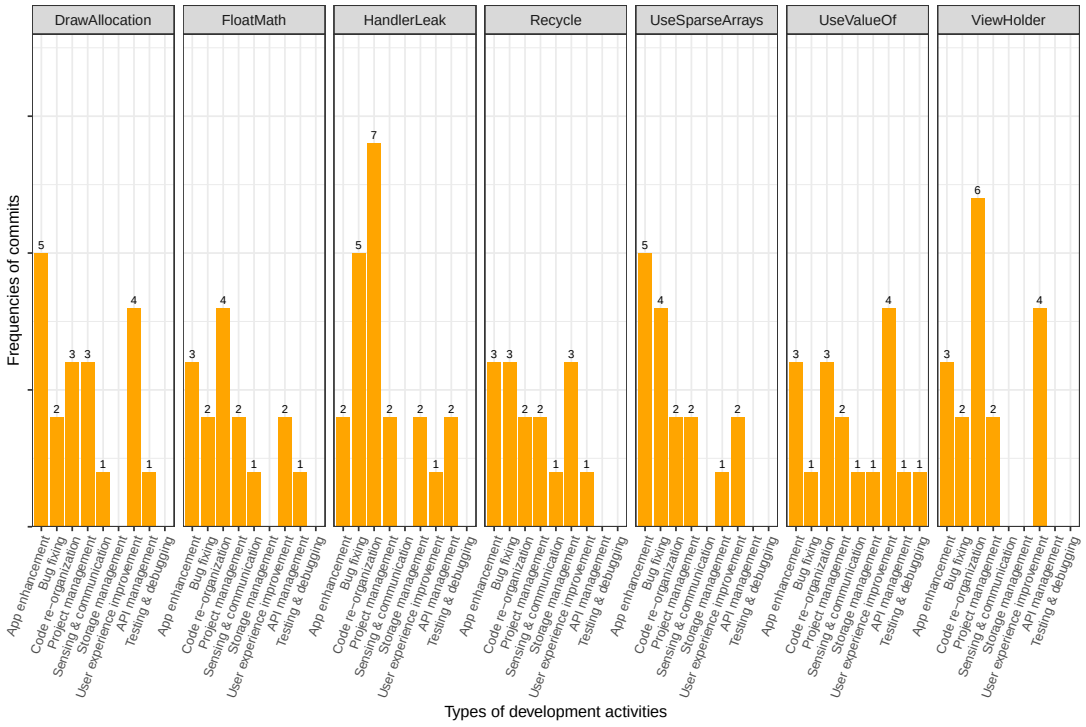
It does not come as a surprise that *code re-organization*, *app enhancement*, *bug fixing*, and *user experience improvement* are the most recurrent types of development activities co-occurring with issue resolutions involving long code churn; indeed, they are also the most recurrent activities in the original study in [94], which did not focus on any specific type of commit. Similarly, the most recurrent activity co-occurring with issue resolutions with high code churn is code re-organization (for both $HandlerLeak$ and $ViewHolder$), which includes activities like refactoring, and code cleanup. Also this result is quite expected since resolving those two types of issues can be considered as special cases of code refactoring or cleanup.

We consider also the cases requiring **low code churn** and analyze the resulting data with the same procedure we applied before and focusing on the issue-resolving commits with the lowest code churn. The resulting subset is composed of 70 commits (10 for each type of performance-related issue) and has an average of 22.61 and a median of 10.50 LOCs per commit. As shown in Figures 5.27(a) and 5.27(b), also when considering commits with low code churn we did not get patterns extremely different from what we observed in commits with high code churn. Overall, in many cases we could observe that the resolution of performance issues occurs together with other potentially unrelated development activities (*e.g.,* app enhancement or bug fixing). This may be an indication that Android Lint is commonly used as part of everyday development activities, potentially thanks to its default integration to the Android Studio IDE. This finding is also confirmed in [56], where many Android developers reported that they prefer to (i) use Android Lint from the project startup and (ii) try to keep the code as clean as possible by frequently considering Android Lint in their development workflow.

We performed the analyses described above in order to get an initial indication about what developers are doing contextually to the resolution of performance issues. In many cases we could observe that the resolution of performance issues occurs together with other potentially unrelated development activities (*e.g.,* app enhancement or bug fixing).



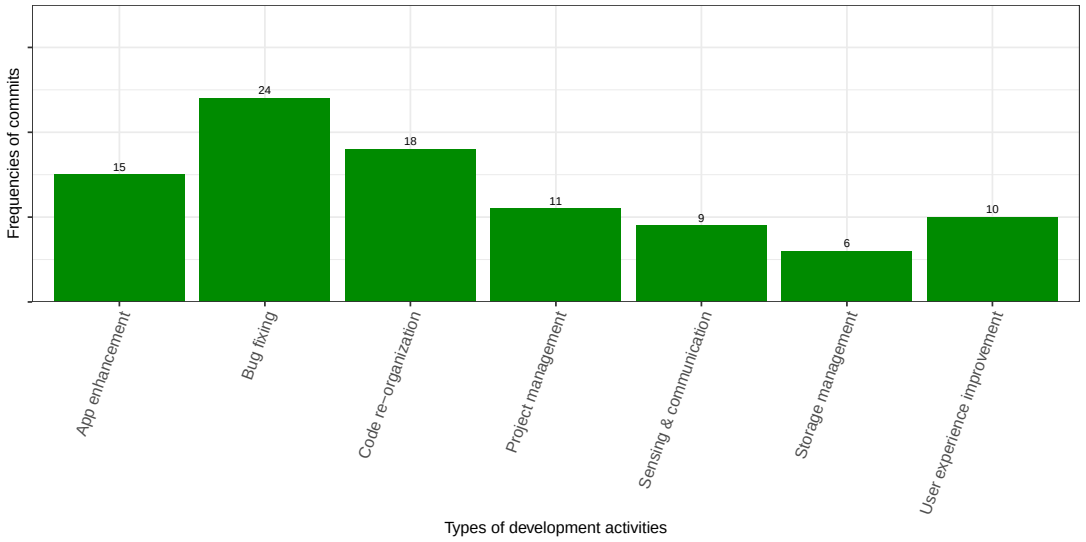
(a) Frequency of development activities in general.



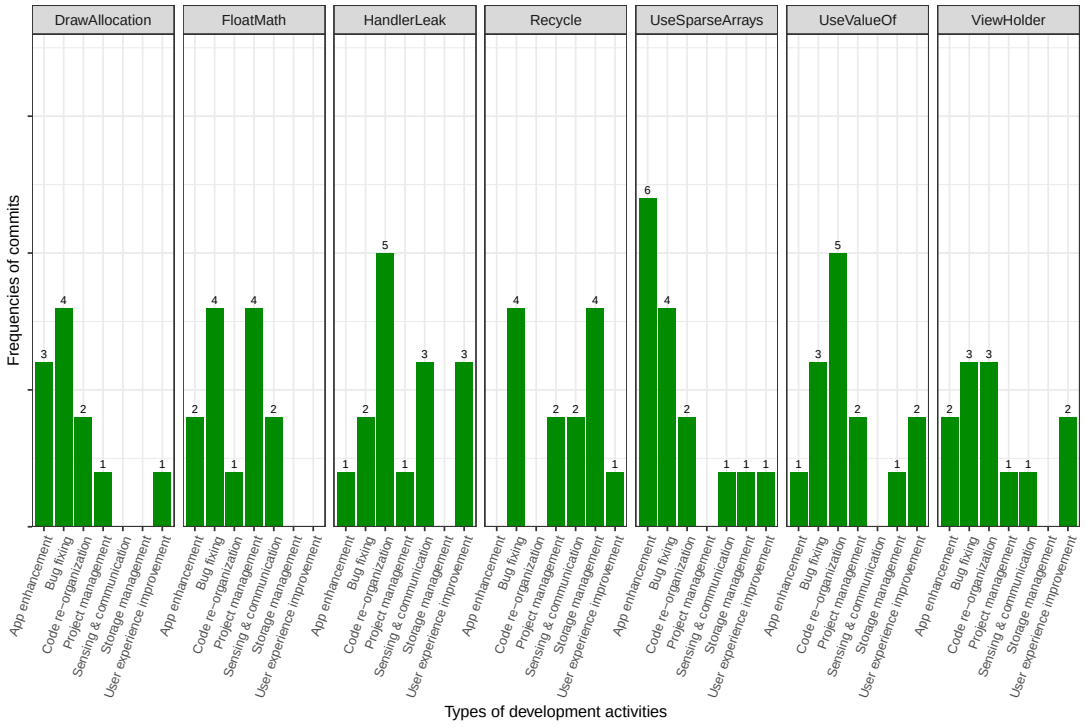
(b) Frequency of development activities across types of performance issues.

FIGURE 5.26: Development activities performed in issue-resolving commits with high code churn.

This may be an indication that Android Lint is commonly used as part of everyday development activities, potentially thanks to its default integration to the Android Studio IDE. This finding is also confirmed in [56], where many Android developers reported that they prefer to (i) use Android Lint from the project startup and (ii) try to keep the code clean by frequently considering Android Lint checks.



(a) Frequency of development activities in general.



(b) Frequency of development activities across types of performance issues.

FIGURE 5.27: Development activities performed in issue-resolving commits with low code churn.

With only 280 data points, we are aware that the performed analyses have low statistical power. Nevertheless, a more in-depth analysis about the root causes of low-high code churns and number of days before resolution is surely a worthwhile future research direction. For the interested researchers, the raw data we manually collected so far is available in the replication package of this study.

We also noted that some issues have a very short resolution time (even less than one day in some cases). We performed an additional analysis similar to the one about code churn with a focus on the top-10 commits in terms of issue resolution time for each type of performance issue. Figure 5.28 presents those developers' activities co-occurring with issue-resolution commits with extremely low (average = 2.62, median = 0.22) or high (average = 582, 568, median = 236) resolution times in days. Similarly to what we observe also for code churn, the most recurring development activities tend to co-occur with issues with both extremely high and low resolution times.

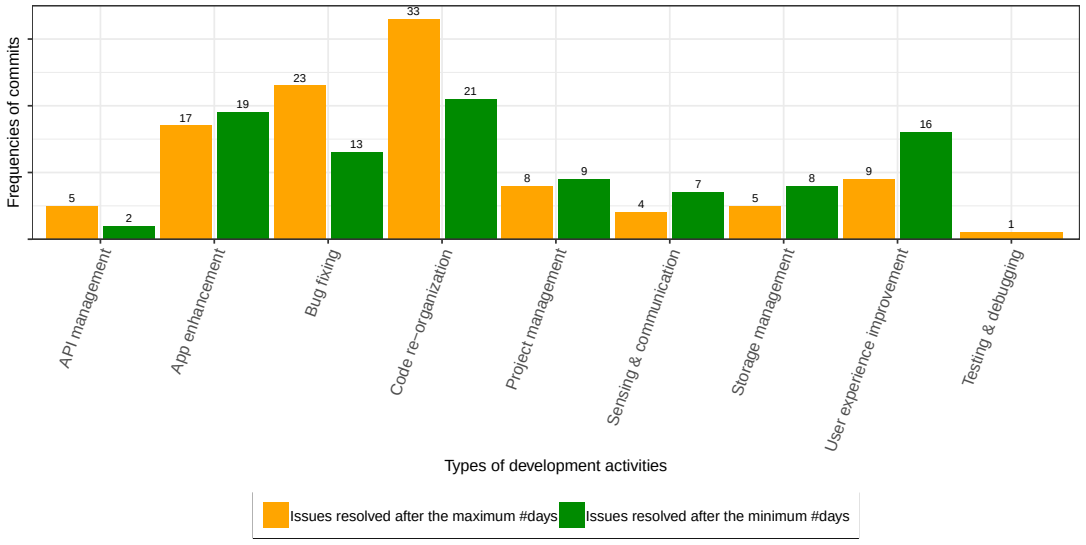


FIGURE 5.28: Development activities performed in issue-resolving commits with min and max resolution time

Finally, only in about 10% of issue resolution commits developers explicitly mention that they resolved a performance issue. This phenomenon can be seen as an indication of the fact that *the resolution of statically-detectable performance issues is embedded into other activities when developing Android apps*. One explanation of this phenomenon may be the integration of Android Lint into Android Studio, where the results of the analysis are directly integrated into the development environment, without context switches, tool configuration, etc. The same trend has been confirmed also in a recent industrial study at Google, where the results of static analyses are taken more into consideration when they are integrated into the development workflow, executed at compile time, and enabled by default for everyone [103].

5.8 Threats to Validity

This section discusses the main threats to the validity of our study and the countermeasures we applied for mitigating them.

Construct validity threats are related to the relationship between theoretical knowledge and actual observations. We detected performance issues by using only one static analysis tool, *i.e., Android Lint*. This decision involves the risk of having a mono-method bias in our study [119] since the results of the whole study are based exclusively on the types of issues supported by the current implementation of *Android Lint*. This means that other types of performance issues may be present in the analyzed apps, but are not considered in our study because they may not be supported by *Android Lint*. As discussed in Chapter 2, there is a number of other static analysis tools which are applicable to Android apps, such as *FindBugs*, *PMD*, *PerfChecker*, *Paprika*, *aDoctor*. Those tools could have been used in combination with *Android Lint*, in order to complement and cross-check its results and have a better coverage of statically-detectable Android performance issues. However, at the time of writing, *Android Lint* is the only static analysis tool which (i) is dedicated to Android-specific issues at the source code level and (ii) has a specific category for performance-related issues. We decided to use only *Android Lint* in order to do not confound the results of the study by considering issues outside the scope of the experiment (*i.e.,* statically-detectable performance issues in Android apps), such as maintainability smells, bugs, security vulnerabilities, etc.

Moreover, in this study we put ourselves in the same conditions as the developers, who generally use Android Lint because it is integrated and activated by default in Android Studio. As confirmed also by other studies, having a linter integrated in the common development workflow makes it more trustable by developers [56, 103]. This makes us reasonably confident about the representativeness of the results of our study, especially when dealing with the resolution and lifetime of the performance issues identified by Android Lint. We do not have empirical evidence about the adoption in practice of linters developed in academia, *e.g., Paprika*.

Nevertheless, the usage of Android Lint allows us to cover a relatively large set of performance-related issues, ranging from low-level issues (*e.g., UseValueOf*) to more encompassing ones (*e.g., HandlerLeak*).

When executing the experiment, Android Lint was supporting 9 performance-related checks, whereas its current latest version supports 36 performance checks (as of June 2019). In order to better understand which Android Lint checks could have been included in our study by considering the latest version of Android Lint, we analyzed the current implementation and documentation of Android Lint and found out that we could

support 4 additional performance checks, namely: *WakelockTimeout*, *StaticFieldLeak*, *LogConditional*, and *SyntheticAccessor*. However, at the time of the computation this was not considered and will be a matter of future investigation. As a way to mitigate this potential threat to validity, we make the data about all 36 performance-related checks in Android Lint publicly available in the replication package of the study.

As many other static analysis tools, *Android Lint* can suffer from the presence of false negatives, *i.e.,* performance issue not detected. Nevertheless, the number of performance issues detected by *Android Lint* is relatively large, making us reasonably confident about the considerations we give when analyzing their evolution over time. Indeed, it is out of the scope of our study to precisely identify *all* performance issues of Android apps, whereas our main objective is to characterize how statically-detectable ones evolve over the lifetime of Android apps, how much time they remain in the code base, and how developers actually resolve them. Moreover, our study may potentially suffer from the presence of false negatives, *i.e.,* performance issues actually present in the app but are not detected by Android Lint. This potential bias is mainly due to (i) the fact that we rely on the heuristics and checks implemented in *Android Lint* and (ii) that we exclusively rely on *Android Lint* for the detection of performance issues. As previously discussed, we decided to focus exclusively on *Android Lint* in order to keep the experiment as focused and realistic as possible (in terms of developers' conditions) and because *Android Lint* currently is the only static analysis tool which has a dedicated category of checks related to performance and it is specific for Android. As future work, as more static analysis tools for Android performance issues will possibly emerge, we will replicate this study and complement our results with those obtained from other tools.

As a way to complement the findings emerging from this study, in future work we will assess the actual impact of the detected issues by dynamically analyzing the considered apps. This can be useful to gain evidence-based insights about the practical consequences of the performance issues detected by Android Lint.

It is important that the toolchain for extracting the performance issues across all commits is implemented and configured correctly. We mitigated this potential threat to validity by carefully designing the whole toolchain (see Section 5.1.3), by testing each component of the toolchain in isolation via repositories for which we knew already the expected outcomes of each data extraction step, and by making the implementation of the toolchain publicly available for independent verification and replication (see the replication package of this study).

Another threat is related to the degree to which the selected apps are representative of the target population (*i.e.,* Android apps published in the Google Play store). We mitigated this threat by considering a relatively large initial set of Android apps (4,287)

and by performing an in-depth data quality assurance and filtering process (see Section 5.1.2).

When considering issue resolutions, there might have been cases of "accidental" performance issue resolutions, when a source code fragment was deleted for other reasons. Nevertheless, those cases are not jeopardizing the results of the study since our aim is to establish the lifetime of performance issues, independently of whether the resolution of the issues is conscious or not. In order to better characterize to what extent developers are consciously resolving performance issues, in RQ2.4 we report on the documented resolutions of issues only. Also, for RQ2.1, RQ2.2, and RQ2.3 we partially mitigated this potential source of bias through a line-based tracking of the issues over time via the LHDiff tool (see Section 5.1.3).

When answering RQ2.3, we consider the CDFs globally with respect to each type of performance issue, instead of considering them on a per-app basis. This may be a threat to validity since the lifetime of performance issues may vary depending on the potentially different maintenance strategies of each individual app. In this context, app-specific factors may influence the obtained distributions, potentially missing the opportunity to make a more fine-grained analysis. However, we decided to analyze the CDFs globally with respect to each type of performance issue due to the relatively limited number of data points we could have obtained when considering each app in isolation.

When answering RQ2.4, we are assuming that if a commit message contains specific keywords, it is describing the resolution of a performance issue. We are aware that such an approach may miss commits where the resolution of the performance issue is not documented. Also, we did not manually evaluated the results of 143 documented commits obtained by regular expressions. However, False positives have been avoided by performing a manual analysis of all identified issue-resolution commits.

Conclusion validity is about the relationship between treatment and outcomes of the study. We carefully took into consideration the assumptions of each applied statistical test. We minimized the possibility of misleading results by relying on non-parametric tests, such as the KS test.

The qualitative analysis we performed when answering RQ2.1 is based on the manual categorization of issue evolution plots, potentially leading to the subjective interpretation of evolution patterns. We mitigated this potential threat to validity by (i) carefully following the open card sorting methodology [107], (ii) involving three researchers, who worked both independently and collaboratively across the various analysis phases, and

(iii) statistically assessing the level of agreement between the involved researchers via the Cohen-Kappa statistics [42].

We mitigated the above mentioned potential threats to validity by preparing a full replication package of the study containing the raw data and statistical data analysis scripts, thus making the data analysis phase of this study fully reproducible.

Finally, in the preliminary analysis presented in Section 5.7.3 (implication R4), we use both code churn and issue resolution time as proxies of the difficulty of resolving issues. However, as it also emerged from that preliminary analysis, massive code churn and long issue resolution times may be due to developers performing other activities that are not related to performance issues (*e.g.,* resolving bugs or implementing new features). As a future work we will mitigate this potential threat to validity by (i) identifying the subset of commits in which developers are working exclusively on the resolution of performance-related issues and (ii) carrying out a more in-depth analysis only on those commits commits.

Internal validity is related to factors internal to our study that can influence our results. During the dataset building phase, we noticed that many potentially-relevant GitHub repositories were actually not containing apps (*e.g.,* repositories hosting only Android libraries). As discussed in Section 5.1.2, we discarded those types of repositories from our dataset. Moreover, despite all considered repositories are about the implementation of Android apps, their structure can heavily differ in terms of folders and files organization. This means that it is possible to obtain false results by considering non-app related source code in the static analysis (*e.g.,* third-party libraries, code implementing the back-end of the apps, code developed for other platforms). We mitigated this potential threat to validity by identifying, for each repository, the app's root folder containing its source code.

External validity is related to the generalizability of the obtained findings. Due to our requirement of having access to the full versioning history of the apps, this study considers exclusively Android apps whose source code is available in GitHub, which may be not representative of the population of all Android apps. However, as we study how apps evolve over time, we need access to the previous versions of the app. Mining GitHub grants access to fine-grained snapshots of each app, whereas in Google Play developers publish only the official releases of their apps. Moreover, we are interested in how performance issues are introduced and resolved by developers in the Java code of their apps; in Google Play only the binary code of the app is available, which may be structurally different from the source code produced by developers *e.g.,* because of code obfuscation. Nevertheless, the built dataset has a high heterogeneity, both in terms of apps size, number of contributors, lifetime, and categories. Moreover, in order to further

mitigate this potential threat to validity, we ensured that all considered apps are also distributed in the Google Play Store, meaning that they are real apps being actually used, and not on-time demos or toy examples.

5.9 Conclusions

This study mainly focused on the analyzing the evolution of performance issues detected by Android Lint. This empirical study is carried out to answer the second high-level research question (RQ2) of this dissertation (section 1.3), *i.e.,*

RQ2 - *How do performance issues identified by Android Lint evolve in Android apps?*

In this chapter we investigate how performance problems — detected by a static analysis tools, *i.e.,* Android Lint [5] — occur, evolve and eventually disappear in Android apps. More specifically, we analyze a set of 724 popular open source Android apps, and we found performance issues in 316 of them.

The results of the study indicate that:

1. Issues due to the lack of recycling data collections and other resources such as database cursors are the most frequently occurring ones, while Android Lint found very few instances of *ViewTag* and *WakeLock* issues.

2. Performance issues in general tend to appear suddenly in a file rather than being gradually introduced, and remain in the system for a long time (and number of commits).

3. Some issues, primarily related to the user interface and to memory management, are either resolved (in our observation period) or tend to remain in the app for a longer time, while other issues (primarily of algorithmic nature) tend to be resolved quickly (when they are resolved), possibly because there are well-known, easy solutions for them.

4. Performance issue resolutions are documented in commit messages in 10% of the cases. This may either indicate that in other cases they were (accidentally) resolved along with other changes, or that in any case their resolution was not considered as the primary goal of the commit.

Chapter 6

Automatic resolution of statically-detectable performance issues in Android apps

Due to the tremendous rise in the number of Android apps in the recent past, managing performance-related issues remains a potential challenge. The apps that trigger performance issues are seldom taken into attention by the developers' in-depth due to various reasons mentioned as follows; (i) lack of knowledge about the resolution of issues, (ii) some issues require a lot of time and effort, especially in changing the lines of codes to resolve such as *HandlerLeak* and *ViewHolder* and (iii) The presence of certain types of issues do not affect on the overall working of the app, but if they are manifested further, it could cause severe performance degradation.

Many studies have been done in the past with emphasis on the detection and categorization of performance issues in mobile apps [55][57][83][101]. Also, few studies proposed tool-based solutions such as *Paprika* [63], *PerfChecker* [83] and a catalogue tool *aDoctor* [92] which have been used to identify different types of code smells including the performance-related bad practices. Moreover, there is an Android-specific linter *Android Lint* [5] (which is officially integrated into Android studio) is utilized to detect and provide suggestions to manually resolve performance problems.

There are many state-of-the-art tools available to detect performance issues, but none of them resolve such issues automatically. However, some auto-refactoring tools such as as *Walkmod*[1], *Facebook pf*[2] and *Kadabra*[3], auto-resolve various types of code smells.

[1]WalkMod. `http://walkmod.com/`
[2]Facebook pf. `https://github.com/facebookarchive/pfff`
[3]Kadabra. `http://specs.fe.up.pt/tools/kadabra/`

AutoRefactor[4], an open-source Eclipse plugin, automatically remove bad code smells. Cruz *et al.* [45] proposed an extension of *AutoRefactor* named *Leafactor* to automatically resolve performance issues triggered by Android Lint. The main advantages of using auto-refactoring tools are: (i) developers do not require the need to spend more time on manual correction of the source code, (ii) it provides with best possible performance-related practices, and (iii) reduce the cost of development effectively.

In this chapter, we propose an auto-refactoring tool named ***ALPAR***[5]–***A****ndroid* ***L****int* ***P****erformance issues* ***A****uto* ***R****esolver*– for Android Lint performance issues in Android apps written in Java. We extended the *Leafactor* tool by implementing three new rules, extending two existing rules, and adopting two rules as they are (i.e., one from *Leafactor* and one from *AutoRefactor*). In total, seven types of Android Lint performance issues can be automatically refactored using the proposed plugin. The fixes are based on the guidelines provided by the official Lint documentation[6] and catalog of manual resolutions presented in Chapter 5.

6.1 The Proposed Tool

To automatically resolve the Android-related performance issues identified in Chapter 5, we implement a tool–***ALPAR***. We combined the static code analysis and the automatic refactoring approach to target the performance-oriented issues in Android apps (as discussed in [45]). Fig. 6.1 represents the architecture adopted from *Leafactor*) diagram of the proposed plugin in which refactoring engine is responsible for refactoring the input Java files. The ALPAR takes a single file or a complete Android project as input, then it parses all the Java files, and automatically refactor each instance of Android-specific performance issues.

ALPAR is based on AutoRefactor and Leafactor. *Autorefactor* is an open-source plugin with a list of common code practices which allow developers to automatically optimize their code. The code enhancement mechanism can be in the form of smaller unused code cleanups as well as by replacing the piece of code with an efficient one.

The backend of *AutoRefactor* uses different API's to exploit Java Abstract Syntax Trees (ASTs) [45]. Autorefactor is officially available in the Eclipse Marketplace[7] and can

[4]AutoRefactor - Eclipse plugin to automatically refactor Java codebases. `https://github.com/JnRouvignac/AutoRefactor`

[5]`https://github.com/teerath91/ALPAR`

[6]We take advantage of the fact that issues in *Android Lint* reports are tagged with a fixed set of categories (`http://tools.android.com/tips/lint-checks`) like performance, correctness, accessibility, usability, etc.

[7]Eclipse Marketplace. `https://marketplace.eclipse.org/`

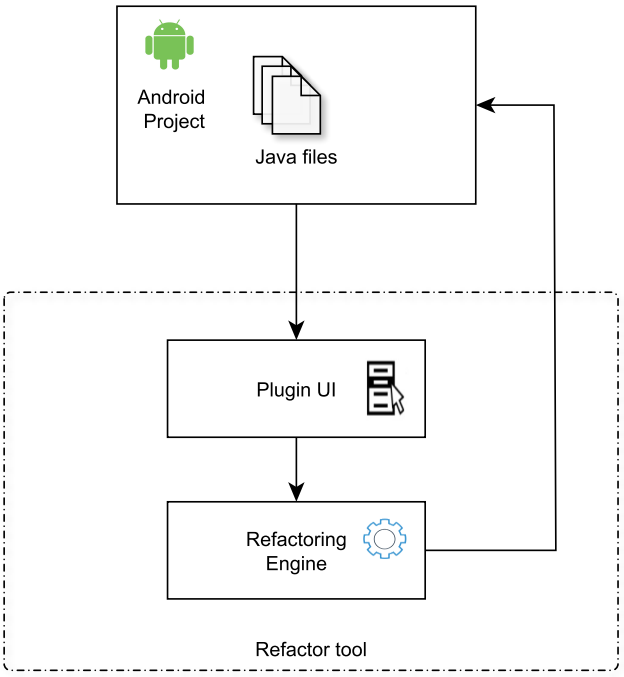


FIGURE 6.1: Overview of Automatic refactoring architecture [45].

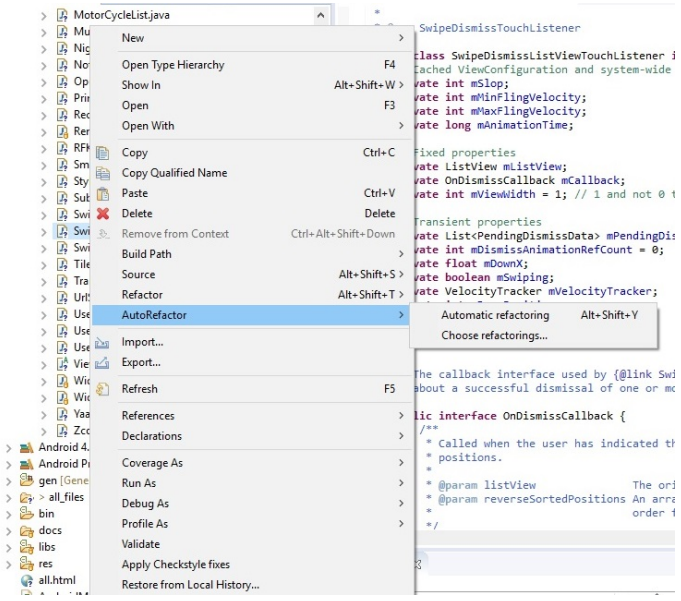


FIGURE 6.2: Screenshot about choosing the "Automatic refactoring" option.

be easily integrated into the Eclipse IDE. Fig. 6.2 shows a screenshot, in which developers can select the *Autorefactor* option and apply different types of refactoring. Likewise, *Leafactor* is used to resolve five types of performance issues of Android Lint. We extended the *Leafactor* [45] by adding the rules described in the next section (*i.e.*, Section 6.2).

6.2 Performance Refactoring Rules

Here, we present and discuss the rules implemented in the proposed plugin (ALPAR), which are designed from the catalog discussed in Chapter 5 and the recommendations provided by the Android Lint documentation [3]. To better understand about how this refactoring occurs, we report two listing examples for each type of issue (*i.e.,* before and after refactoring). Finally, we discuss some corner cases[8] for each rule that will not be considered by the our tool. The seven types of performance issues[9] supported by the developed Eclipse plugin are briefly explained below.

6.2.1 DrawAllocation: Allocations within drawing code

Performance of any application is directly inclined to its draw or layout operations. According to Android Lint documentation [3], the developer should not allocate the objects while drawing or during layout operations; but they should be declared upfront in the `onDraw/onLayout` method [10] to improve the app performance (e.g., achieving a smoother UI). However, this operation could activate the garbage collector during the drawing activity.

In the following are the two code snippets of *DrawAllocation* rule representing two versions of the code (*i.e.,* before and after refactoring).

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    Paint paint = new Paint(); ❶
    paint.setColor(Color.WHITE);
    paint.setTextSize(16);
    canvas.drawText("" + soundGenerators.get(0).getOscillator().getFrequency(),
    getWidth() / 2, getHeight() / 2, paint);
}
```

LISTING 6.1: Example of the *DrawAllocation* issue (Before Refactoring) (`marcopar/SliderSynth` - app/src/main/java/eu/flatworld/android/slider/KeyboardView.java).

[8] corner cases generally referred to the issues, that occur very rarely and therefore require relatively more research work to explore their potential solutions such as dynamic allocation cases of DrawAllocation issues.

[9] `http://tools.android.com/tips/lint-checks`

```
Paint paint = new Paint();  ❷

@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    paint.setColor(Color.WHITE);
    paint.setTextSize(16);
    canvas.drawText("" + soundGenerators.get(0).getOscillator().getFrequency(),
    getWidth() / 2, getHeight() / 2, paint);
}
```

LISTING 6.2: Example of the *DrawAllocation* issue (After Refactoring) (`marcopar/SliderSynth` - app/src/main/java/eu/flatworld/android/slider/KeyboardView.java).

❶. In Listing 6.1, the object `paint` will be allocated every time when `onDraw` method is called, thus injecting a *DrawAllocation* issue.

❷. Listing 6.2 shows the refactored version obtained by declaring the object `paint` upfront *i.e.,* before the `onDraw` method.

We extend this rule of *Leafactor* tool [45] by applying some minor changes: (i) this rule was originally checked the object allocation only in the `onDraw` method, but we extended it to analyze `onLayout` and `onMeasure` methods as well; (ii) we support additional subclasses such as *android.view.View*, *android.widget.SeekBar*, *android.widget.EditText* and *android.view.SurfaceView*; finally, (iii) we, further improved the rule for `Rect` objects. For instance, the following allocation inside onDraw method introduces a *DrawAllocation* issue.

```
Rect r = new Rect((thing.x-1) * cellWidth, (thing.y) * cellHeight + 2, thing.x
* cellWidth, (thing.y + 1) * cellHeight + 2);
```

To automatically resolve such type of case, initially, we declare the object `Rect r` outside from the `onDraw` method.

```
Rect r = new Rect();
```

Then, we defined all the points of the rectangle object inside the `onDraw()` method:

```
r.bottom = (thing.y + 1) * cellHeight + 2;

r.right = thing.x * cellWidth;

r.top = (thing.y) * cellHeight + 2;

r.left = (thing.x-1) * cellWidth;
```

Finally, to draw the `Rect`, we used the following command.

```
canvas.drawRect(r, robotBg);
```

The above solution is adopted from the catalog of developer resolution[10] discussed in Chapter 5 which will lead to resolution of the *DrawAllocation* issue.

Corner cases: As per our understanding, two types of cases are not considered for this rule; (i) BitmapFactory allocation cases, and (ii) dynamic allocation or the declarations inside the loop (*i.e., for, while*), because those allocations needed to update in every iteration inside the `onDraw` method.

6.2.2 FloatMath: Using FloatMath instead of Math

In the latest Android Version, `Math java.lang.Math` is faster than the `FloatMath` (`android.util.FloatMath`) [3] because the JIT [60] compiler has a better approach to optimize it (`java.lang.Math`). However, in the older versions of Android, it was preferred to use `android.util.FloatMath` with the intent to enhance the performance. In our proposed Eclipse plugin, we automatically refactor the *FloatMath* performance issue by transforming `FloatMath` to `Math`. As an example for *FloatMath* rule, Listing 6.3 and Listing 6.4 report before and after refactoring versions of the code respectively.

```
private static void getRotationVectorFromGyro(final float[] gyroValues, final float[]
    deltaRotationVector,
  final float timeFactor)
{
  float[] normValues = new float[3];
  // Calculate the angular speed of the sample
  float omegaMagnitude = FloatMath.sqrt(gyroValues[0] * gyroValues[0] + gyroValues[1] *
    gyroValues[1]
    + gyroValues[2] * gyroValues[2]); ❶
  // Normalize the rotation vector if it's big enough to get the axis
  if (omegaMagnitude != 0.0f)
  {
    normValues[0] = gyroValues[0] / omegaMagnitude;
    normValues[1] = gyroValues[1] / omegaMagnitude;
    normValues[2] = gyroValues[2] / omegaMagnitude;
  }
  // Integrate around this axis with the angular speed by the timestep
  // in order to get a delta rotation from this sample over the timestep
  // We will convert this axis-angle representation of the delta rotation
  // into a quaternion before turning it into the rotation matrix.
  float thetaOverTwo = omegaMagnitude * timeFactor;
```

[10] https://github.com/xxv/robotfindskitten/commit/611fc371c841b86fc7d8517e2f53b9b01d6e34e2

```
  float sinThetaOverTwo = FloatMath.sin(thetaOverTwo); ❷
  float cosThetaOverTwo = FloatMath.cos(thetaOverTwo); ❸
  deltaRotationVector[0] = sinThetaOverTwo * normValues[0];
  deltaRotationVector[1] = sinThetaOverTwo * normValues[1];
  deltaRotationVector[2] = sinThetaOverTwo * normValues[2];
  deltaRotationVector[3] = cosThetaOverTwo;
}
```

LISTING 6.3: Example of the *FloatMath* issue (Before Refactoring) (`almalence/OpenCamera` - src/com/almalence/plugins/capture/panoramaaugmented/AugmentedRotationListener.java).

```
private static void getRotationVectorFromGyro(final float[] gyroValues, final float[]
    deltaRotationVector,
  final float timeFactor)
{
  float[] normValues = new float[3];
  // Calculate the angular speed of the sample
  float omegaMagnitude = (float) Math.sqrt(gyroValues[0] * gyroValues[0] +
    gyroValues[1] * gyroValues[1]
    + gyroValues[2] * gyroValues[2]); ❹
  // Normalize the rotation vector if it's big enough to get the axis
  if (omegaMagnitude != 0.0f)
  {
    normValues[0] = gyroValues[0] / omegaMagnitude;
    normValues[1] = gyroValues[1] / omegaMagnitude;
    normValues[2] = gyroValues[2] / omegaMagnitude;
  }
  // Integrate around this axis with the angular speed by the timestep
  // in order to get a delta rotation from this sample over the timestep
  // We will convert this axis-angle representation of the delta rotation
  // into a quaternion before turning it into the rotation matrix.
  float thetaOverTwo = omegaMagnitude * timeFactor;
  float sinThetaOverTwo = (float) Math.sin(thetaOverTwo); ❺
  float cosThetaOverTwo = (float) Math.cos(thetaOverTwo); ❻
  deltaRotationVector[0] = sinThetaOverTwo * normValues[0];
  deltaRotationVector[1] = sinThetaOverTwo * normValues[1];
  deltaRotationVector[2] = sinThetaOverTwo * normValues[2];
  deltaRotationVector[3] = cosThetaOverTwo;
}
```

LISTING 6.4: Example of the *FloatMath* issue (After Refactoring) (`almalence/OpenCamera` - src/com/almalence/plugins/capture/panoramaaugmented/AugmentedRotationListener.java).

❶ ❷ ❸. It can be seen in Listing 6.3 that *FloatMath* issue was detected due to the use of `android.util.FloatMath`.

❹ ❺ ❻. In the refactored version of the code (Listing 6.4) the issue was resolved by utilizing `java.lang.Math`.

Corner cases: To the best of our knowledge, there is no corner cases left for this issue.

6.2.3 HandlerLeak: Handler reference leaks

In Java, non-static class holds an implicit reference to its outer class, which can lead to memory leaks problems in the apps [14].

In the proposed plugin, we implemented the *HandlerLeak* rule to resolve this issue automatically. The following three steps are required to fix the Handler declaration, that solve the *HandlerLeak* performance issue.

1. Declare the Handler as a static class;
2. Instantiate a `WeakReference` to the outer class and pass this object to the Handler when instantiating it;
3. Add all references to the members of the outer class using the `WeakReference` object.

Listing 6.5 (before refactoring) and Listing 6.6 (after refactoring) represent an example of *HandlerLeak* issue.

```
private Handler mHandler = new Handler() {   ❶
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case MSG_SUCCESS:
                mAdapter.notifyDataSetChanged();
                stopLoadingAnim();
                break;

            case MSG_FAILURE:
                showLoadingAnim();
                break;
        }
    }
};
```

LISTING 6.5: Example of the *HandlerLeak* issue (Before Refactoring) (`XunMengWinter/Now` - app/src/main/java/top/wefor/now/fragment/ZcoolFragment.java).

```
private static class mHandler1 extends Handler {    ❷
    private final WeakReference<ZcoolFragment> mActivity;

    public mHandler1(ZcoolFragment activity) {    ❸
        mActivity = new WeakReference<ZcoolFragment>(activity);
    }

    public void handleMessage(Message msg) {
        ZcoolFragment activity = mActivity.get();    ❹
        if (activity != null) {
            switch (msg.what) {
                case MSG_SUCCESS:
                    mAdapter.notifyDataSetChanged();
                    stopLoadingAnim();
                    break;
                case MSG_FAILURE:
                    showLoadingAnim();
                    break;
            }
        }
    }
}

private mHandler1 mHandler = new mHandler1(this);
```

LISTING 6.6: Example of the *HandlerLeak* issue (After Refactoring) (`XunMengWinter/Now` - app/src/main/java/top/wefor/now/fragment/ZcoolFragment.java).

❶. It can be observed in Listing 6.5 that *HandlerLeak* issue was detected due to the use of non-static handler.

❷ ❸ ❹. In Listing 6.6, the refactored code transformation was obtained by implementing the three steps as described above.

Corner cases: We only covered the non-static handler class cases. However, there are other cases (*i.e.,* anonymous classes) that we did not included but could lead to potential memory leaks in the apps.

6.2.4 Recycle: Missing recycle() calls

Generally, in Java, data is automatically destroyed by Garbage collector (GC), once it is used. However, there are some resources such as `TypedArrays` and `VelocityTrackers` that are different since these resources have other internal data which should be recycled

by a direct call to the `recycle()` method. This is also true for various other classes such as `Parcel`, `MotionEvent` and `DatabaseCursor` (database cursor should be freed by `close()` call).

We take this rule from the *Leafactor* with the addition of a conditional statement to check the resource before recycling and when closing the cursor to ensure that the resource is already null. For some instances, we also analyzed the case[11], where the resource is used on the return statement, and we recycled such resource by calling `recycle()`.

Listing 6.7 and Listing 6.8 are two examples coming from our dataset and show how the *Recycle* issue can be automatically resolved by ALPAR.

```
@Override
public void onStartTracking(MotionEvent event) {
  MotionEvent motionEvent = MotionEvent.obtain(event);
  motionEvent.setAction(MotionEvent.ACTION_CANCEL);  ❶
  SubjectCard.super.onTouchEvent(event);
}
```

LISTING 6.7: Example of the *Recycle* issue (Before Refactoring) (`marcioapaiva/mocos-controlator` - src/com/marcioapf/mocos/view/SubjectCard.java).

```
@Override
public void onStartTracking(MotionEvent event) {
  MotionEvent motionEvent = MotionEvent.obtain(event);
  motionEvent.setAction(MotionEvent.ACTION_CANCEL);
  if (motionEvent != null) {  ❷
    motionEvent.recycle(); ❸
  }
  SubjectCard.super.onTouchEvent(event);
}
```

LISTING 6.8: Example of the *Recycle* issue (After Refactoring) (`marcioapaiva/mocos-controlator` - src/com/marcioapf/mocos/view/SubjectCard.java).

❶. It can be noted from Listing 6.7 that the `MotionEvent` resource should be recycled so that other similar types of resource reuse them in the future.

❷ ❸. The refactored version is reported in Listing 6.8, where we initially check if the resource `MotionEvent` is not already null and then we recycle it by calling `recycle()`.

Corner cases: As per our knowledge, there are not any corner cases left for this type of issue.

[11] `https://github.com/SunyataZero/KindMind/commit/b85c68`

6.2.5 UseSparseArrays: HashMap can be replaced with SparseArray

The mapping from an integer to an object using the `HashMap` data structure is relatively slow in practice [3]. According to Android Lint documentation, this data structure should be replaced by an alternative efficient data structure such as `SparseArray`. Hence, if the key is a primitive type, then `SparseArray` can be used instead of `HashMap` [22].

We implement this rule by considering the recommendations specified in the documentation of Android Lint and the examples emerging from our dataset where developers resolved this issue. Thus, we implement the `SparseArrays` rule to automatically resolve these type of performance issue. Moreover, in this rule we also consider various types of `HashMap` to `SparseArray` case scenarios with having different types of key values. Table 6.1 reports such `HashMap` to `SparseArray` cases considered in the rule.

TABLE 6.1: `HashMap` to `SparseArray` code transformation

HashMap	**SparseArray**
HashMap<Integer, V>	*SparseArray< V>*
HashMap<Integer, Boolean>	*SparseBooleanArray*
HashMap<Integer, Integer>	*SparseIntArray*
HashMap<Integer, Long>	*SparseLongArray*
HashMap<Long, V>	*LongSparseArray< V>*
HashMap<Long, Long>	*LongSparseLongArray*

As an example from our dataset, Listing 6.9 and Listing 6.10 report two versions of the same code, before and after automatic resolution via ALPAR, respectively.

```
public class Yaaic
{
    public static Yaaic instance;
    private HashMap<Integer, Server> servers;  ❶
    private boolean serversLoaded = false;
    private Yaaic()
    {
        servers = new HashMap<Integer, Server>();  ❷
    }
    public void loadServers(Context context)
    {
        if (!serversLoaded) {
            Database db = new Database(context);
            servers = db.getServers();
            db.close();
            // context.sendBroadcast(new Intent(Broadcast.SERVER_UPDATE));
```

```
            serversLoaded = true;
        }
    }
}
```

LISTING 6.9: Example of the *UseSparseArrays* issue (Before Refactoring) (`pocmo/Yaaic` - src/org/yaaic/Yaaic.java).

```
public class Yaaic
{
    public static Yaaic instance;
    private SparseArray<Server> servers;  ❸
    private boolean serversLoaded = false;
    private Yaaic()
    {
        servers = new SparseArray<Server>();  ❹
    }
    public void loadServers(Context context)
    {
        if (!serversLoaded) {
            Database db = new Database(context);
            servers = db.getServers();
            db.close();
            // context.sendBroadcast(new Intent(Broadcast.SERVER_UPDATE));
            serversLoaded = true;
        }
    }
}
```

LISTING 6.10: Example of the *UseSparseArrays* issue (After Refactoring) (`pocmo/Yaaic` - src/org/yaaic/Yaaic.java).

❶ ❷. It can be seen in Listing 6.9, the *UseSparseArrays* issue is detected while using `HashMap` data structure with primitive data values *i.e.,* `HashMap<Integer, Server>`.

❸ ❹. The issue is automatically refactored by using `SparseArray<Server>` as shown in the Listing 6.10.

Corner cases: As per our understanding, the following corner case is not supported by this rule.

```
public static SparseArray <Integer>providerMap = new SparseArray
<Integer>();
```

Till now, we mostly covered the `HashMap` to `SparseArray` cases for auto-refactoring. However, the above-explained case[12] occur rarely and thus have minimal knowledge to resolve it and yet to be explored.

6.2.6 UseValueOf: Should use valueOf instead of new

This rule is primarily used to deal with primitive data types. Calling the constructor for wrapper classes directly is not a good programming practice *i.e.,* `new Integer()` as recommended in the official Lint documentation. Instead as an alternative the factory method `valueOf()` should be called. In other words, it is better to use `Integer.valueof()` instead of `new Integer()`. The reason is because a `new Integer` instantiates a new object each time, where as `Integer.valueOf` retrieves the cached values (`Integer.valueOf` implements a cache for the values between -128 to +127) [18]. Hence, `Integer.valueOf` is more memory friendly with respect to `new Integer`.

We derived the *UseValueOf* refactoring rule from an existing rule[13] of *AutoRefactor* emphasis on automatically converting the `new Integer` into `Integer.valueOf`. The cases considered for this rule are provided in Table 6.2.

TABLE 6.2: *UseValueOf* code transformation

Original code	**Transformed code**
new Byte()	*Byte.valueOf()*
new Character()	*Character.valueOf()*
new Double()	*Double.valueOf()*
new Float()	*Float.valueOf()*
new Integer()	*Integer.valueOf()*
new Long()	*Long.valueOf()*
new Short()	*Short.valueOf()*

In the following, we report the code snippet from our dataset (Listing 6.11 and Listing 6.12) to explain the auto-refactoring of the *UseValueOf* issue.

```
private static void initNullValues() {
  _nullValues = new Hashtable4();
    _nullValues.put(boolean.class, Boolean.FALSE);
    _nullValues.put(byte.class, new Byte((byte)0)); ❶
    _nullValues.put(short.class, new Short((short)0)); ❷
    _nullValues.put(char.class, new Character((char)0)); ❸
    _nullValues.put(int.class, new Integer(0)); ❹
    _nullValues.put(float.class, new Float(0.0)); ❺
    _nullValues.put(long.class, new Long(0)); ❻
    _nullValues.put(double.class, new Double(0.0)); ❼
```

[12] https://github.com/hwki/SimpleBitcoinWidget/commit/986b95688a1adf442fef56371e273efc506deb9b

[13] https://github.com/luiscruz/AutoRefactor/blob/master/plugin/src/main/java/org/autorefactor/refactoring/rules/PrimitiveWrapperCreationRefactoring.java

```
}
```

LISTING 6.11: Example of the *UseValueOf* issue (Before Refactoring) (`bjerva/tsp-lexikon- android - tsp-lexikon-android/src/com/db4o/internal/Platform4.java`).

```
private static void initNullValues() {
    _nullValues = new Hashtable4();
    _nullValues.put(boolean.class, Boolean.FALSE);
    _nullValues.put(byte.class, Byte.valueOf((byte)0));    ❶
    _nullValues.put(short.class, Short.valueOf((short)0));  ❷
    _nullValues.put(char.class, Character.valueOf((char)0));  ❸
    _nullValues.put(int.class, Integer.valueOf(0));        ❹
    _nullValues.put(float.class, Float.valueOf((float) 0.0)); ❺
    _nullValues.put(long.class, Long.valueOf(0));    ❻
    _nullValues.put(double.class, Double.valueOf(0.0));    ❼
}
```

LISTING 6.12: Example of the *UseValueOf* issue (After Refactoring) (`bjerva/tsp-lexikon- android - tsp-lexikon-android/src/com/db4o/internal/Platform4.java`).

❶ ❷ ❸ ❹ ❺ ❻ ❼. As shown in Listing 6.11 the *UseValueOf* issue was spotted with an instantiate new object *e.g.,* `new Byte`, `new Short`, `new Character`, and `new Integer`.

❶ ❷ ❸ ❹ ❻ ❻ ❼. Listing 6.12 shows the refactored version of code obtained after using the factory method `valueOf`.

Corner cases: As per our understanding, there is no corner case for this issue.

6.2.7 ViewHolder: View Holder Candidates

While scrolling the list view items, the `getview()` method may frequently invoke the `FindViewById()` to draw the data on the screen, which is an inefficient method and may degrade the performance of the app. As per the recommendation of the official Lint document [3] at the time of creating adapter class, it is a lousy practice to unconditionally inflate new layout. instead, developers should reuse an existing layout, if available. This can be achieved by creating a *ViewHolder pattern*, which will reuse the data from already drawn items. This approach will reduce the number of invocations to `FindViewById()` method leading to smoother scrolling.

We adopted this rule as in the *Leafactor*. Listing 6.13 and Listing 6.14 report an example of occurrence of the *ViewHolder* issue, before and after its resolution, respectively.

```
import android.app.Activity;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseAdapter;
import android.widget.SpinnerAdapter;
import android.widget.TextView;
import com.example.lexinproject.R;

public class LanguagesAdapter extends BaseAdapter implements SpinnerAdapter{
    private Activity activity;
    private Language[] list_bsl;

    public LanguagesAdapter(Activity activity, Language[] list_bsl){
        this.activity = activity;
        this.list_bsl = list_bsl;
    }
    public int getCount() {
        return list_bsl.length;
    }
    public Object getItem(int position) {
        return list_bsl[position];
    }
   /* public String getItemId(int position) {
        return list_bsl.get(position).getName();
    }*/
  @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = activity.getLayoutInflater();
        View spinView = inflater.inflate(R.layout.spin_layout, null);  ❶
        //TextView t1 = (TextView) spinView.findViewById(R.id.field1);
        TextView t2 = (TextView) spinView.findViewById(R.id.field2);  ❷
        //t1.setText(String.valueOf(list_bsl[position].getName()));
        t2.setText(list_bsl[position].getName());
        return spinView;
    }

  @Override
  public long getItemId(int position) {
    // TODO Auto-generated method stub
    return 0;
  }
}
```

Listing 6.13: Example of the `ViewHolder` issue (Before Refactoring) (`cmykola/Lexin` - src/com/example/lexinproject/data/LanguagesAdapter.java).

```
import android.app.Activity;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseAdapter;
import android.widget.SpinnerAdapter;
import android.widget.TextView;
import com.example.lexinproject.R;

public class LanguagesAdapter extends BaseAdapter implements SpinnerAdapter{
    private Activity activity;
    private Language[] list_bsl;

    public LanguagesAdapter(Activity activity, Language[] list_bsl){
        this.activity = activity;
        this.list_bsl = list_bsl;
    }
    public int getCount() {
        return list_bsl.length;
    }
    public Object getItem(int position) {
        return list_bsl[position];
    }
   static class ViewHolderItem {  ❸
    TextView t2;
  }
  /* public String getItemId(int position) {
        return list_bsl.get(position).getName();
    }*/
  @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        ViewHolderItem viewHolderItem;
        LayoutInflater inflater = activity.getLayoutInflater();
        if (convertView == null) {       ❹
            convertView = inflater.inflate(R.layout.spin_layout, null);
            viewHolderItem = new ViewHolderItem();
            viewHolderItem.t2 = (TextView) convertView.findViewById(R.id.field2);
            convertView.setTag(viewHolderItem);
        } else {
            viewHolderItem = (ViewHolderItem) convertView.getTag();
        }
        View spinView = convertView;
    //TextView t1 = (TextView) spinView.findViewById(R.id.field1);
        TextView t2 = viewHolderItem.t2;  ❺
        //t1.setText(String.valueOf(list_bsl[position].getName()));
        t2.setText(list_bsl[position].getName());
        return spinView;
```

```
    }

    @Override
    public long getItemId(int position) {
    // TODO Auto-generated method stub
        return 0;
    }
}
```

LISTING 6.14: Example of the `ViewHolder` issue (After Refactoring) (`cmykola/Lexin` - src/com/example/lexinproject/data/LanguagesAdapter.java).

❶ ❷. As it can be seen in Listing 6.13, a new `LayoutInflater` object is instantiated for each iteration when the `getview()` method, which overwrites the parametric content of `convertView` method. Further, in each iteration, a view is retrieved using the `findviewbyid()` method which depicts the text of each list item `TextView` object.

❸ ❹ ❺. Listing 6.14 shows that when a list item is built the references regarding its objects in the inner view (*i.e.,* `TextView` object) are recognized and stored in a specific data structure. This module will be executed for the first item of the list, while the next iterations will receive the `convertView` from the parameters. Therefore the `FindViewById()` method invocation will not require to fetch the TextView object every time. By implementing this pattern, the computations of new `LayoutInflater` list items as well as the fetching the inner views is avoided. Hence reducing the memory needed to build new items of the list.

Corner cases: As far as we know, there are no unmanaged corner cases for this type of issue.

6.3 Evaluation

We evaluated ALPAR with two main goals, namely: (i) to assess the developers' perception of the code refactored by ALPAR and (ii) to evaluate the impact of the resolved performance issues on the overall performance achieved at run time by Android apps.

The above mentioned goals can be refined into the following sub-research questions which aim to address the third main (high-level) research question (RQ3) of this dissertation.

***RQ*3.1** – *How do developers perceive the code automatically refactored by ALPAR?*

This research question focuses on understanding the auto-refactoring code from developers point of view. This research question can be further divided into two sub-questions.

- ***RQ*3.1.1** – *What is the perceived performance of the code automatically refactored by ALPAR?*
- ***RQ*3.1.2** – *What is the perceived understandability of the code automatically refactored by ALPAR?*

To answer those research questions, we performed a survey-based experiment described in section 6.3.1.

***RQ*3.2**– *What is the impact of statically-detectable Android performance issues during run-time?*

The aim of this research question is to quantify the run-time performance impact on Android apps and whether the performance can be improved by resolving related issues in the apps. To examine this aspect, we performed a preliminarily study by conducting a measurement-based experiment targeting a physical Android device and a set of synthetically-defined mobile apps. The details of the experiment are provided in section 6.3.2.

The **replication package** of both of the experimental studies is publicly available[14] for independent verification. The replication package includes raw data, the source code of the mobile apps, R scripts used for data analysis, and the plots used for exploring and analysing the results of the evaluation.

6.3.1 Experiment 1 - Survey-based study

Design – The goal of the study is to *analyze* the code refactoring proposed by ALPAR with the *purpose* of understanding developers' perceptions *with respect to* code performance and comprehensibility from the *viewpoint* of developers in the *context* of Android apps.

This online survey-based study involves a total of 21 Android developers to investigate the developer's perception on the code refactored by ALPAR (in terms of code performance and comprehensibility). All the participants have been sampled by convenience among the direct contacts of the author and supervisors of this dissertation. The participants of this experiment are diverse with respect to experience *i.e.,* number of apps developed, number of years associated with Android development and the number of employees in the organization. The minimum number of apps developed by a participant is at least two. The demographics show that 31.8% of participants developed more

[14]Replication Package `https://github.com/teerath91/PerformanceRefactoringStudies/`

than 10 apps and the same amount of participants lies in the range of 6-10 apps. Moreover, 81.8% of participants have 1 to 5 years of association with Android development and about 16.7% of participants have been involved for more than 10 years. Furthermore, nearly 45.5% of participants work in organizations with more than 50 employees. The overall demographics suggest that the considered participants for the survey study have a fair and close background, experience, and working environment of Android app development.

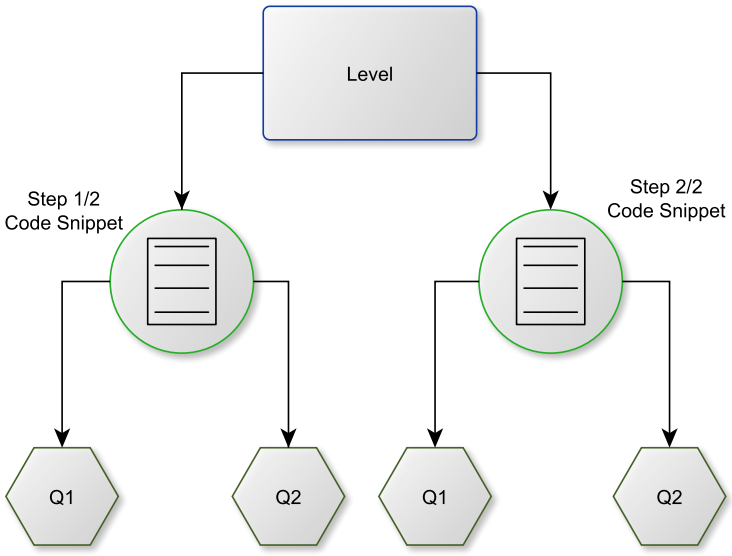


FIGURE 6.3: Logical diagram of one level of JSON file.

In the following, we discuss the design of the experiment that is composed of three batches, each of them involving an online questionnaire. Each questionnaire is composed of seven levels (*i.e.,* one level for each of the seven types of issue). Each batch is composed of seven participants. The rationale behind such allocation of the three batches is: (i) to reduce the number of questions per participant, thus promoting his/her ability to focus and (ii) to cover a wide range of refactoring cases for each type of performance issues. Each questionnaire has the following structure:

- seven levels, one level for each type of performance issue;
- each of the seven levels contains two different steps (see Figure 6.3), one for each instance of the current Android performance issue (*e.g., DrawAllocation*);
- every step shows either non-refactored code (the code with performance issue/before refactoring) or refactored code (the code after automatically resolving a previously existing issue using ALPAR);

- every snippet is accompanied by a clarification URL which points to the GitHub page for accessing the Java code of the whole file where the snippet belongs to (this is used to provide context to the question);
- for each snippet we show two questions, one about the perceived performance and another about the perceived understandability;
- for each snippet we provide an optional comment field where participants can provide any feedback.

Each questionnaire has a total of 14 steps (see figure 6.4 for a screenshot of a level of the questionniare) Moreover, to mitigate biases in the experiment, we take the following steps:

- The order of the levels is fully randomize across different instances of the questionnaire. For instance, in the first questionnaire, the *DrawAllocation* level may be in level 1, while the *DrawAllocation* level might be at level 5 for the second questionnaire, etc.
- In each of the 14 steps, we randomly provided the position of the non-refactored code and the refactored code. For example, in some steps, non-refactored code is located to the left side of the snippet, whereas, for other instances, it may be placed on the right side.

Furthermore, for each instance of the issue, the coding interface will ask two questions from developers. The text of the questions will always be the following:

> $\boldsymbol{Q_1}$ – The snippet of code to the Left executes faster than the snippet of code to the right?

As already discussed, given the random position of the snippets in each JSON file, this question evaluates how developers perceive the two versions of snippets (non-refactored vs. refactored code) in terms of execution time.

> $\boldsymbol{Q_2}$ – The snippet of code to the Left is more comprehensible than the snippet of code to the right?

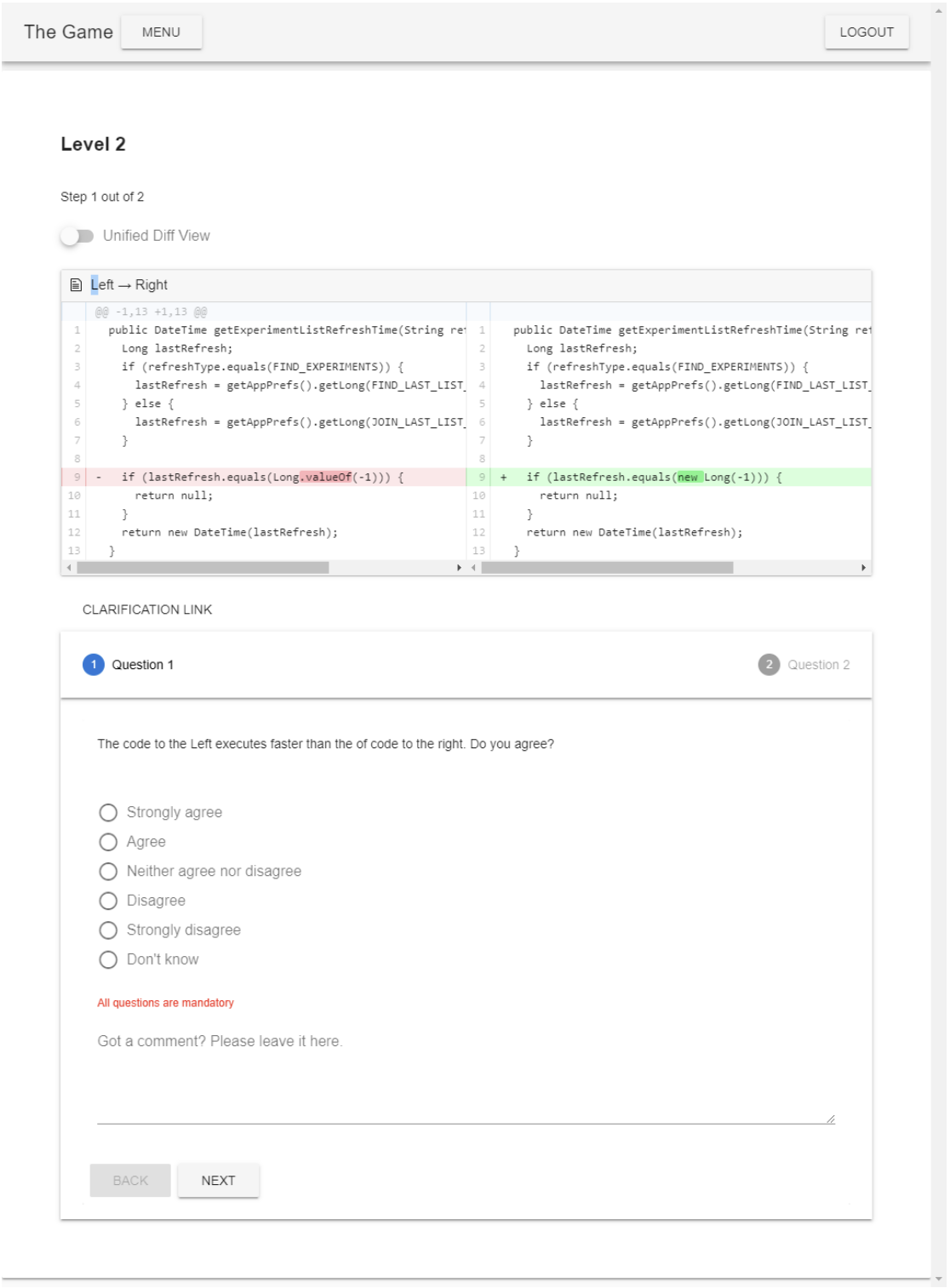


FIGURE 6.4: One level of Coding game interface.

Similarly to $\boldsymbol{Q_1}$ as the code snippets are randomly distributed, this question is dedicated to analyzing the code understandability of non-refactored and refactored code versions.

The possible responses follow a Likert scale[15] with the following structure:

[15] `https://github.com/jbryer/likert`

- Strongly agree
- Agree
- Neither agree nor disagree
- Don't know
- Disagree
- Strongly disagree

The online questionnaire has been defined by reusing an open-source project called *The coding game*[16] interface because; (i) the structure is designed in a way that is suited to our experiment *i.e., levels and steps*, and (ii) it supports diff[17] (code difference between two files) as shown in Fig. 6.4.

Data Analysis – To analyze the results, we collected the survey responses obtained from 21 Android developers and reported them using Likert plots. For each of the two questions *(**Q_1**, **Q_2**)* asked to the developers (section 6.3.1), we analyzed the responses in terms of refactored vs non-refactored code separately for **Q_1** and **Q_2**. For instance, in the case of a particular type of issue, we checked the percentage of participants responding in support of refactored and how many preferred non-refactored code. In addition to this, we also briefly described some of the developer's comments given during the analysis of the code snippets.

Results – A total of 21 developers answered all given questions of the survey. Fig. 6.5 and Fig. 6.6 summarizes the distribution of the responses obtained from the developers for **Q_1** and **Q_2**, respectively.

It can be observed from the Fig. 6.5 that regarding *DrawAllocation* types of issues, 72% participants were clearly acknowledged that non-refactored code will not execute faster than the refactored one, while on the other hand, nearly 64% participants agreed with the fact that refactoring code executes faster than non-refactoring code. This may be due to the allocation of the object during drawing or layout operation is quite sensitive for the app performance and developer may be aware of this phenomenon.

Then, there are code snippets of *UseSparseArrays* type, where participants can clearly distinguish the faster snippet. About 53% of the participants were in favour of using refactoring code over non-refactoring code for performance reasons. On the other hand, according to the 60% participants, non-refactored code will not run faster than its refactored code version.

[16] `https://github.com/S2-group/TheCodingGame`
[17] diff. `https://en.wikipedia.org/wiki/Diff`

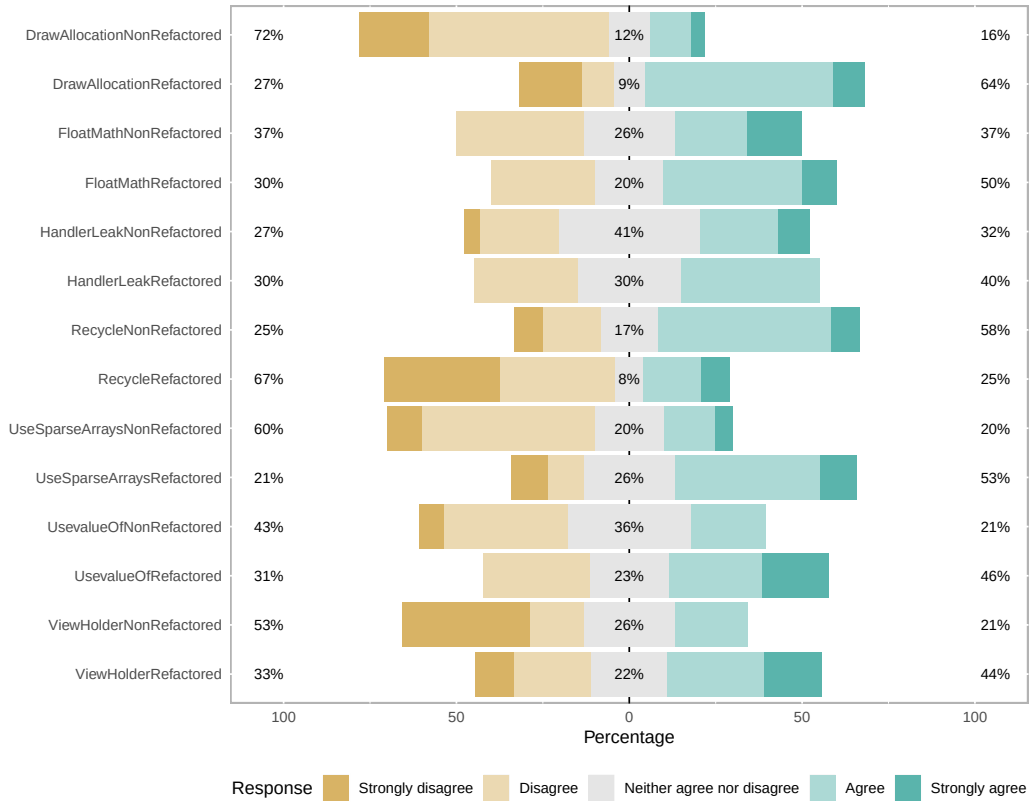


FIGURE 6.5: Code Execution Response ($\boldsymbol{Q_1}$).

It is also interesting to note that for some types of performance issues, developers remained neutral in their response regarding code execution. For instance, 30% were *neither agree nor disagree* in the *HandlerLeak* issue type. Also, 40% were neutral regarding the usage of non-refactoring code over refactoring one. This shows that for certain types of issues, it is difficult for participants to clearly distinguish between code with/without performance performance problems and this explains the importance and need of having automated tools like Android Lint and ALPAR for doing these kinds of tasks.

In the case of *Recycle* type of issues, 67% of developers were not in the support of using refactoring code snippets over non-refactoring in terms of execution. Since developers can not differentiate the refactoring version, this is one of the potential reason why we have the highest number of *Recycle* issues identified in Chapter 5.

Fig. 6.6 depicts the participants' responses regarding the code comprehension ($\boldsymbol{Q_2}$) of refactored vs non-refactored code snippets. 53% of developers feel *UseSparseArrays* refactoring code snippets are more understandable than non-refactoring code. Other than this, a large number of participants (about 50%) agreed that refactoring code version is more readable than its non-refactoring for the *DrawAllocation* type of issues.

It is reasonable to say that refactoring of code does not intend to make the code more understandable, rather it is more focus on the performance aspect.

For *ViewHolder* issue type, 60% of participants think that non-refactoring code is not comprehensive than its refactoring code version, while in case of *Recycle* type of issue, nearly 42% of participants were not convinced that non refactored is more understandable than the refactored one. The most difficult source code to comprehend by developers was about *FloatMath* issues, where 60% of the participants think that the refactored code is not more comprehensible than the non-refactored code.

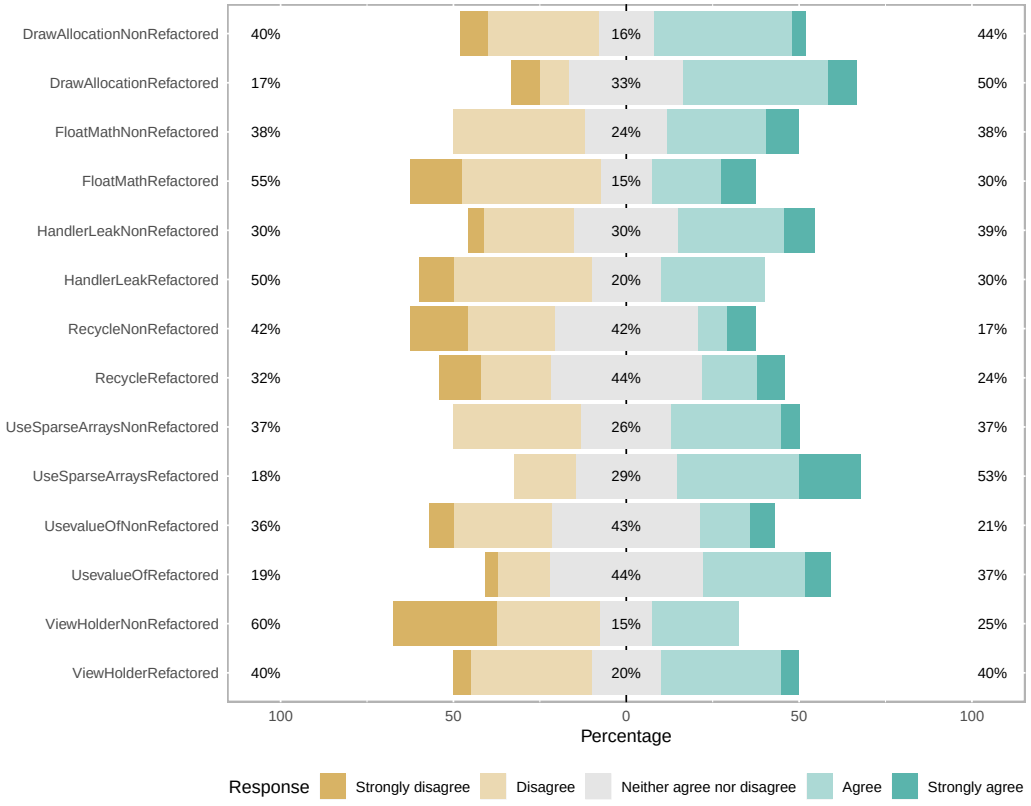


FIGURE 6.6: Code Comprehension Response ($\boldsymbol{Q_2}$).

Furthermore, in the comments field, some participants confirmed the usage of refactoring code version as more efficient. For instance one participant gave the following feedback regarding the *UseSparseArrays* issue:

"SparseArray is intended to be more memory efficient than using a HashMap to map Integers to Objects, both because it avoids auto-boxing keys and its data structure doesn't rely on an extra entry object for each mapping."

For the *HandlerLeak* type of issues, one participant provided comment regarding how to resolve this issue and what may be the consequences of this issue:

"Its not about code performance the performance addition of right side where we have a Weak Reference to activity and null check should be minimal. It's actually about memory leak."

This depicts that *HandlerLeak* issue can be avoided by adding the weak reference to the activity, as done in ALPAR.

About *UseValueOf* issue type, one of the developer given the following comment:

"As per Java documentation, Long.valueOf should be preferred to new Long(long) although in normal apps won't make any noticeable difference."

Summary – RQ3.1 – The summary of this research question is divided into two parts.
Summary – RQ3.1.1 – For $\boldsymbol{Q_1}$ (code execution), the overall trend shows that for various types of performance issues (*DrawAllocation*, *FloatMath* and *UseSparseArrays*), more than 50% participants think that the refactoring versions of code (through ALPAR) are faster with respect to their non-refactored versions and for some issue types (*i.e., UseValueOf* and *ViewHolder*), this ratio is close to 50%. It is interesting to observe that in *HandlerLeak* issue type, the developers could not distinguish between the two versions and thus opted to remain neutral.

Summary – RQ3.1.2 – Regarding the $\boldsymbol{Q_2}$ (code comprehension), the responses depicted a mix trend. For *DrawAllocation* and *UseSparseArrays*, around 50% and 53% participants (respectively) agreed that refactored code version is more comprehensive. On the other hand, about 60% of participants were convinced that non-refactored code version of *ViewHolder* is more readable than the refactored version. Finally, for *Recycle* and *UseValueOf*, a large chunk of participants preferred to be neutral.

Threats to Validity – In the following, we describe the threats to validity for this experiment and the actions we adopted to avoid them.

Transferability threats are the level to which the findings of qualitative research can be transferred or generalized to other contexts. One of the possible potential threats regarding generalisability is the sample size we considered in this study. Since the sample size of participants is not large in number, it is reasonably fair to say that it is not representing the majority of the Android developers. To mitigate this threat, we

considered the Android developers that have varied years of experience, nationalities and actively working in some organizations. However, choosing very highly professional developers may also inject new biases to our study. This is because the visualization and analysis of code snippets vary from junior to senior developers. Thus, it is always good to include more participants for the survey to decrease the biasness but this will increase workload and consume more time *e.g.,* contacting the developer whether he/she is willing to participate for the survey, wait for the participants to fill and submit the survey. Moreover, few other similar studies as our topic regarding the sample size characteristic have been done by Habchi *et al.* [56] which included 14 participants and Tomasdottir *et al.* [109] (considered 15 participants) provided good literature motivation for having sample size of 21 participants is sufficient for the study.

Another possible threat could be that we contacted the developers may use different types of Lint checks including performance checks and they may manually resolve them (*i.e.,* those who have an idea about the Lint performance issues). There may be participants who did not perform manual performance refactoring (or lack of knowledge in performance refactoring) and participated in the survey to give their opinion on two snippets of code. To alleviate this threat, we are giving a reference to study which confirmed that 97% of participants who chose linters to check issues are dependent on the Android Lint [23]. This is because Lint is by default integrated with official Android Studio and thus many developers are using it for developing the apps. Besides, Lint can provide the feature of dedicated performance categories. Therefore, we presume that Android developers have sufficient knowledge about Lint performance issues.

Credibility is about the trustworthiness of results that can affect the study. One of the most likely threat is the credibility of the obtained results. Since we surveyed the Android developers who are working in this area (*i.e.,* Android apps development), but we are not fully sure whether the answer given by their own experience or intuition or knowledge gathered from some other source. To address this threat, we always asked at the end of the survey regarding some details about the participant him/herself, such as the number of years of development experience, how many Android apps developed, and the number of employees working in the company etc. to understand the nature of work. Moreover, at the beginning of the survey, we also told participants that their identity will be kept anonymous and the survey is not aimed to judge the participants.

Confirmability in qualitative research referred to as the level at which results of our study can be confirmed or validated by other researchers. One confirmability threat may be analyzing and confirming the response independently obtained from developers. To avoid this threat, we use coding game interface to conduct an online survey. The results of the study are stored in a JSON file and anyone can check it independently.

6.3.2 Experiment 2 - Measurement-based study

Design – The goal of this study is to *analyze* the performance issues identified by ALPAR with the *purpose* of measuring the run-time impact with *respect* to CPU utilization and memory consumption from the *viewpoint* of developers and researchers in the *context* of Android apps.

To answer the second research question of this chapter defined in section 6.3, we conducted an experiment by following three steps:

1. **Creating synthetic apps with performance issue (Non-Refactored Apps):** We implemented six simple Android apps, one for each type of performance issue except *FloatMath*[18] issue. In each of the six self-developed apps, we have introduced one of each type of performance issues. Listing 6.15 reports an example of such app with *DrawAllocation* performance issue (*i.e.,* at line 26). At the end of this step, we had six apps with their performance issues.

2. **Creating synthetic apps with refactored performance issues (Refactored Apps):** Similarly to the first step, we obtained six apps without performance-related issues. As an example, Listing 6.16 represents an app without *DrawAllocation*. At the end of this step, we have refactored versions of the previously mentioned six apps.

3. **Running Android runner to measure the performance:** We ran all the synthetic apps (obtained from step 1 and step 2) 1 million times to clearly analyze the performance impact. We leverage Android runner[19] tool to measure the performance in terms of CPU utilization and memory consumption for all the total 12 apps (APKs). Android runner exploits; (i) *Monkey Runner* for replaying all the executions, and (ii) *Android Debug Bridge (ADB)*[20] to the `dumpsys` tool and gathered data about CPU utilization and memory consumption for each app.

```
package nl.vu.cs.appdrawallocation;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.util.Log;
import android.view.View;
```

[18]Since we built all the apps using API level 26, *FloatMath* issue was deprecated in that version, so to avoid the biasness, we preferred not to include it in this experiment.

[19]Android Runner - Automated experiment execution on Android devices. `https://github.com/S2-group/android-runner`

[20]`https://developer.android.com/studio/command-line/adb.html`

```
public class MyView extends View
{
    Paint paint = null;
    public MyView(Context context)
    {
        super(context);
    }
    @Override
    protected void onDraw(Canvas canvas)
    {
        Log.d("Draw method","called");
        super.onDraw(canvas);
        int x = getWidth();
        int y = getHeight();
        int radius;
        radius = 100;
        Paint p = new Paint();
    }
}
```

LISTING 6.15: Example of the *DrawAllocation* app with performance issue.

```
package nl.vu.cs.appafterdrawallocation;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.util.Log;
import android.view.View;

public class MyView extends View
{
    Paint paint = null;
    public MyView(Context context)
    {
        super(context);
    }
    Paint p = new Paint();
    @Override
    protected void onDraw(Canvas canvas)
    {
        Log.d("Draw method","called");
        super.onDraw(canvas);
        int x = getWidth();
        int y = getHeight();
        int radius;
        radius = 100;;
```

```
    }
}
```

LISTING 6.16: Example of the *DrawAllocation* app without performance issue.

The experiment was conducted on HP Laptop (i7 CPU at 2.5 GHz and 8 GB of memory), Whereas the apps were running on Galaxy A5 mobile device (64bit Octa Core Processor and 32 GB of memory) running API level 26 and Android 8.0.0. Each run of the experiment was repeated 20 times for each APK.

Data Analysis – To analyze the results; we report the statistics summary for both the CPU utilization and memory consumption across all the selected apps (along with box plots). Furthermore, to compare the distributions of different runs (refactored and non-refactored app), we apply the *Wilcoxon rank-sum test* [119] followed by *Cliff delta* with 95% of confidence intervals. [21].

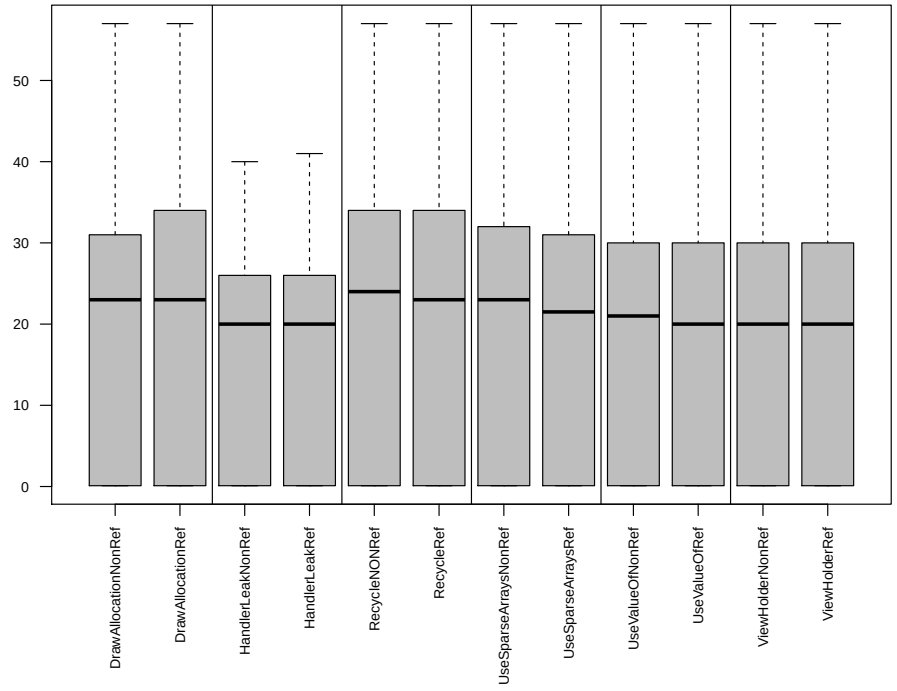


FIGURE 6.7: CPU utilization of selected apps. (in percentage)

Results – As it can be seen from Table 6.3 and Fig. 6.7, the CPU utilization for six types of performance issues have been evaluated using the descriptive statistics summary and box plots respectively. The results reveal that there were slight differences between the means of two versions of apps (*i.e.,* refactored vs. non-refactored). However, in most of the cases, CPU utilization of refactored apps were not significantly improved than their non-refactored versions. Moreover, we compared the distribution runs of CPU

TABLE 6.3: Descriptive statistics for the CPU utilization ((in percentage) of each type of selected apps (SD = standard deviation, CV = coefficient of variation).

Synthetic apps	**Min.**	**Max.**	**Median**	**Mean**	**SD**	**CV**
DrawAllocationNonRef	0.10	57.00	23.00	20.41	15.21	74.53
DrawAllocationRef	0.10	57.00	23.00	20.26	15.91	78.54
HandlerLeakNonRef	0.10	40.00	20.00	17.53	12.76	72.79
HandlerLeakRef	0.10	41.00	20.00	16.91	13.25	78.32
RecycleNonRef	0.10	57.00	24.00	20.43	15.95	78.06
RecycleRef	0.10	57.00	23.00	20.23	15.96	78.88
UseSparseArraysNonRef	0.10	57.00	23.00	19.31	15.94	82.54
UseSparseArraysRef	0.10	57.00	21.50	18.37	15.90	86.50
UseValueOfNonRef	0.10	57.00	21.00	17.44	15.84	90.82
UseValueOfRef	0.10	57.00	20.00	16.81	15.79	93.92
ViewHolderNonRef	0.10	57.00	20.00	16.76	15.62	93.21
ViewHolderRef	0.10	57.00	20.00	16.68	15.69	94.05

TABLE 6.4: Results of the Cliff Delta analysis (CPU utilization) for comparing the amount of difference between two distributions (non-refactored vs. refactored app version) across the six performance issues.

Distributions	**Delta estimate**	**Lower Bound**	**Upper Bound**
DrawAllocationNonRef VS. DrawAllocationRef	-0.008735773 (negligible)	-0.10353159	0.08621732
HandlerLeakNonRef VS. HandlerLeakRef	0.01819155 (negligible)	-0.1247598	0.1604030
RecycleNonRef VS. RecycleRef	0.004071171 (negligible)	-0.07362021	0.08171343
UseSparseArraysNonRef VS. UseSparseArraysRef	0.03148089 (negligible)	-0.03600335	0.09867931
UseValueOfNonRef VS. UseValueOfRef	0.02214067 (negligible)	-0.03808772	0.08220878
ViewHolderNonRef VS. ViewHolderRef	0.001344259 (negligible)	-0.05302900	0.05570957

utilization for both the versions (across all types of issues) using two-tailed *Wilcoxon rank-sum test*. In all the cases, we obtained *p-values* much greater than 0.05 (*p-values* are insignificant), meaning there is a very high probability that the differences between the medians of two versions are not significant. Further, we calculated the *Cliff delta* to quantify the amount of difference between the two distributions. Since the *Wilcoxon test* is not significant, so the cliff delta values are not significant, that is also confirmed in Table 6.6 that the difference was *negligible* between two versions of apps across each issue types.

Regarding the memory consumption, it can be visualized from Table 6.5 and Fig. 6.8 that there are no noticeable differences between the medians and means of two versions of the app (refactored vs. non-refactored) across all six types of issues. Moreover, for certain types of issues, non-refactored versions of the app consume a little more memory than their refactored versions (with respect to mean). Further, *Wilcoxon rank-sum test*

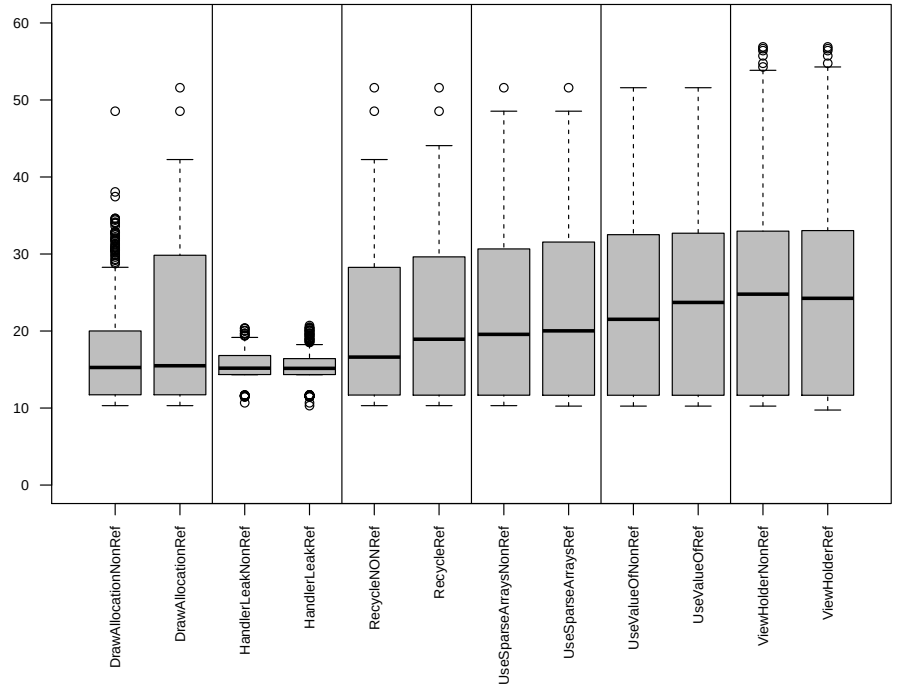


FIGURE 6.8: Memory consumption of selected apps (in Mb).

TABLE 6.5: Descriptive statistics for the memory consumptions (in Mb) of each type of selected apps (SD = standard deviation, CV = coefficient of variation).

Synthetic apps	**Min.**	**Max.**	**Median**	**Mean**	**SD**	**CV**
DrawAllocationNonRef	10.31	48.54	15.27	18.30	7.49	40.92
DrawAllocationRef	10.31	51.59	15.49	20.51	9.02	43.97
HandlerLeakNonRef	10.68	20.39	15.17	15.31	2.76	18.03
HandlerLeakRef	10.31	20.71	15.15	15.18	2.73	17.98
RecycleNonRef	10.31	51.59	16.61	20.47	8.64	42.20
RecycleRef	10.31	51.59	18.93	21.17	8.86	41.87
UseSparseArraysNonRef	10.31	51.59	19.57	22.10	9.83	44.49
UseSparseArraysRef	10.25	51.59	20.02	22.73	10.38	45.68
UseValueOfNonRef	10.25	51.59	21.52	23.34	10.85	46.48
UseValueOfRef	10.25	51.59	23.71	23.77	11.10	46.69
ViewHolderNonRef	10.25	184.21	24.79	27.57	22.60	81.96
ViewHolderRef	9.74	184.21	24.25	27.41	22.16	80.86

indicates that there is considerably less difference between the median of two versions of all types of issues. Finally, Table 6.5 shows the *Cliff Delta* estimation on the difference of two distribution. In each case, we have *negligible* amount of difference between the distributions. With these simple analyses. effect sizes are really negligible, however, if one tries to compose many smells, the performance degradation may not be negligible anymore.

TABLE 6.6: Results of the Cliff Delta analysis (Memory Consumption) for comparing the amount of difference between two distributions (non-refactored vs. refactored app version) across the six performance issues.

Distributions	**Delta estimate**	**Lower Bound**	**Upper Bound**
DrawAllocationNonRef VS. DrawAllocationRef	-0.1028614 (negligible)	-0.197011408	-0.006831514
HandlerLeakNonRef VS. HandlerLeakRef	-0.007490637 (negligible)	-0.1554826	0.1408302
RecycleNonRef VS. RecycleRef	-0.03521593 (negligible)	-0.11397668	0.04398471
UseSparseArraysNonRef VS. UseSparseArraysRef	-0.02678459 (negligible)	-0.09570758	0.04239400
UseValueOfNonRef VS. UseValueOfRef	-0.02065733 (negligible)	-0.08302316	0.04186969
ViewHolderNonRef VS. ViewHolderRef	0.002224764 (negligible)	-0.05436961	0.05880489

Summary – RQ3.2 –The results of the experiment showed that there is no significant trend in terms of differences, both for the CPU usage and memory consumption across all types of considered performance issues. This may be due to the fact that the context of this study is relatively small and simple (as they are not real world apps but instead self-developed)

Threats to Validity – In the following, we report the threats to validity related for the experiment and the adopted precautions we took to avoid them.

External validity. One of the possible threat could be selecting the real apps from Google Play Store, that include six types of performance issues. Thus, this possibly restricts us to generalize the results of our experiment to represent all the set of real Android apps. However, at the same time, the impact of smaller issue types in real apps may not be noticeable when the size of the code file or project is large. Therefore, to avoid this threat, we created synthetic apps for each type of performance issues and executed 1 million times inside each app to clearly observe the impact of performance issues.

Construct validity. In this experiment, we did not consider *FloatMath* issue type due to its depreciation, which limit us to analyze the run time impact of remaining six types of performance-related issues. We took such a decision to reduce the bias of the experiment and maintain the same configurations across all the apps in terms of API level and Android version.

Conclusion validity. The experiment setup can be affected by random events such as Android OS scheduling, and background tasks that can affect the measurements of the apps. To mitigate this threat, we took the following proactive measures: (i) performed

the experiment on brand new Android phone (Galaxy A5) and double-checked that no task was running in background, (ii) to reduce the chance of random events, we repeated the experiment with the same treatments of twenty runs for each considered APKs.

6.4 Discussion

ALPAR can be used by developers to resolve seven types of Android performance-related issues. Since Android Lint is the de facto static analysis tool for Android studio, so developers can make most of this plugin by resolving the issues to ensure the error-prone app development process (*i.e.,* can improve the performance of their apps). Moreover, this plugin can provide a base (direction) for developers to implement the rules for various other types of potential performance issues identified from the tools mentioned in the state-of-the-art (Chapter 3).

From the survey-based study, the majority of the participants (64%) can easily recognize the refactoring code version of *DrawAllocation*, meaning that developers have sufficient knowledge about this issue. This confirms the results achieved by Cruz *et al.* [45], where *DrawAllocation* was not found throughout the evaluation of their study on 140 open-source Android apps which shows developers may tend to pro-actively resolve the issue at the time of app development.

Refactored version of *Recycle* issue type, were not recognized by the majority of developers. Our extensive study conducted in Chapter 5 shows that *Recycle* (550, 22.84%) is the most frequently identified issues. Hence, the results from the survey about different types of performance issues can be used as base to resolve such issues which were not received the due attention from developers' *e.g., Recycle* and *UseValueOf* issue types.

About code understandability, notice that here we are not aiming to make the code more comprehensible, rather the goal of refactoring is mainly performance improvement. However, the results of the survey clearly indicate that some refactored solutions such as *UseSparseArrays* and *DrawAllocation* are more comprehensible. Therefore, refactoring is not detrimental for code comprehensibility.

The measurement-based experiment showed that there is a very small difference in terms of CPU utilization and memory consumptions between original and refactored code for all types of performance issues. This demonstrates that at least refactoring does not hinder the performance. Though the context of this empirical study is not representative of real open-source and industrial apps, therefore, considering it only a preliminary study, developers can explore the run-time performance on real world apps.

6.5 Conclusions

As the mobile apps are becoming more and more popular, developers and researchers are coming with new tools to address the performance concerns in Android apps, especially in the domain of detecting the performance issues. However, there are a few tools available which focus on providing a solution to automatically resolve those issues.

This chapter emphasizes on presenting possible automatic solution for performance smells and investigating performance-related code auto-refactoring for Android apps. The proposed study is carried out to answer the third high-level research question of this dissertation (mentioned in section 1.3), *i.e.,*

RQ3 - *Is it possible to automatically resolve statically-detectable performance issues in Android apps?*

In this chapter, we introduced an Eclipse plugin –**ALPAR**– to auto-refactor the performance-related problems discussed in Chapter 5. By building on the *Leafactor*, a total of seven types of issues can be auto-resolved by our proposed tool. These issue types include: *DrawAllocation, FloatMath, HandlerLeak, Recycle, UseValueOf, UseSparseArrays*, and *ViewHolder*.

Further, we took a step forward to understand the developer's mindset on the refactoring and non-refactoring code by conducting a survey-based experimental study. According to obtained responses, refactoring versions of *DrawAllocation* and *UseSparseArrays* code snippets were seen to execute faster than their non-refactored code snippets. The overall trend showed that in various issue types about (near to or above) 50% of participants inclined towards the usage of refactoring code over non-refactoribg in terms of code execution.

Regarding the comprehensibility of the code, a majority of developers (53%) feel that refactored version of *UseSparseArrays* code is more understandable than its non-refactored one, whereas 60% of participants disagree that non-refactored code is more readable than refactored code version for *ViewHolder* issue type. Finally, in issue types such as *Recycle* and *UseValueOf*, a large number of participants remained neutral. This represents a mix trend for the code comprehensibility.

Finally, we performed a measurement-based study to investigate the run time performance impact of the apps with/without performance issues. The findings of the preliminarily study for CPU utilization depicted that there was very less difference between various versions of the refactored and non-refactored apps . However, for some issues types, CPU utilization for refactored apps were very slightly improved as compared to

its non-refactored versions, such as *UseSparseArrays*. While, in terms of memory consumption, we have not found much difference between the two versions of apps across all issue types. Only refactored app of *HandlerLeak* consumes relatively less memory than its non-refactored app version.

Chapter 7

Conclusions

This section summarizes the main contributions of the dissertation, in the light of the research questions formulated in the thesis. In addition to presenting the contributions, we also reviewed the future prospects related to our research work.

7.1 Main Contributions

The main goal of this thesis is to investigate the performance-related issues in Android apps and provide an in-depth understanding to Android researchers and developers. Thus, the essence of this thesis would provide added value to the researcher community in terms of identification, evolution and resolution of various performance issues at the time of development.

The main contributions of this thesis are summarized explicitly with the obtained results in each main (high-level) research questions of this dissertation defined in Chapter 1.

RQ1 *Which are the most recurrent types of performance-related issues observed in the developers commits for Android apps?*

In Chapter 4, a preliminary study is conducted to qualitatively and quantitatively analyze the performance-related commits in the Android apps. The work is aimed to add the following contributions:

1. An investigation on a total of 457 commits distributed over 180 open-source Android apps. As per our knowledge, this is the first study to evaluate performance issue.

2. A taxonomy of main types of performance issues consists of 10 different categories obtained using card sorting technique is provided. [107]. The most recurrent performance issues observed in commits is about the app user interface (UI). We also found frequent commits pointing the issues related to code smells, app logic, network connection, file system, and local databases etc.. This study provides to developers a checklist to discover all potential performance issues during app development.

RQ2 *How do performance issues identified by Android Lint evolve in Android apps?*

In Chapter 5, an empirically study was conducted with the aim of highlighting the occurrence of performance issues in 316 Android apps by running Android Lint. This analysis resulted in the identification of total 2,408 performance issues. The major outcomes of this research question are briefly described below:

1. According to our findings, *Recycle* issues are very frequent in our dataset (550, 22.84%), they occurred due to lack of recycling data collections and other resources such as database cursors and TypedArrays. While *WakeLock* issues were very uncommon ones (3, 0.12%).
2. Then, we traced the evolution patterns of identified issues by considering the apps' evolution history. Five different evolution patterns emerged by analyzing their frequency by using open card sorting. The STICK evolution pattern is the most immediate one in our dataset *i.e.,* 209 times in 316 apps showing that the issue injected in the app tends to remain for several commits. Finally, a taxonomy is drawn based on these evolution patterns.
3. Furthermore, the empirical research of this work examines to what extent Android developers do not resolve performance issues. The findings showed that there are a total 45.25% of identified issues that remained unresolved in our dataset. *ViewHolder* (101/180, 56.11%) type of issues were not resolved in the huge amount, which can profoundly impact the performance aspect because this issue is mainly deal with smoother scrolling of the `ListView` items. most of the *FloatMath* issues were resolved (52/188, 27.66%).
4. Our investigation analyzed the survival time for different types of identified performance issues as well. We did such analysis to know whether certain types of issues tend to resolved immediately or some of them did not get much attention by developers. Results of this study showed that the issues can be able to survive relatively longer in the Android apps (with an average of 137 days and a median of 21 days) during the whole span of the projects.
5. We also investigated the extent to which developers acknowledge the resolution of the detected performance issues and documented in their commit

messages. We observed that overall 143 commits out of 1,314 (10.88%) were documented by developers. Among these 143 issue resolution commits, *Recycle* issues are frequently documented (57, 39.9%), while *UseSparseArrays* (6, 4.2%) issues are rarely documented by developers. Finally, a catalog of manual corrections for all types of performance-related issues is provided to support early fixing.

RQ3 *Is it possible to automatically resolve statically-detectable performance issues in Android apps?*

1. We proposed an Eclipse plugin named ***ALPAR*** to automatically resolve seven types of Android-specific performance issues discussed in Chapter 5. We extended *Leafactor* [45] with three new rules, modifying two existing rules and utilizing one rule as it was. While, one rule we derived from *AutoRefactor* tool. The issue types covered by the proposed plugin includes: *DrawAllocation, FloatMath, HandlerLeak, Recycle, UseValueOf, UseSparseArrays*, and *ViewHolder*.
2. Moreover, we performed a survey-based experiment to understand the self-accessed performance refactoring from the developers' perspective. This study provides an overview of how developers can perceive the refactoring in terms of code execution and readability. According to obtained results, *DrawAllocation* and *UseSparseArrays* refactoring code executed faster as compared to their non-refactoring code. The overall trend for various types of performance issues like *DrawAllocation*, *FloatMath* and *UseSparseArrays* showed that majority of participants (more than 50%) were inclined towards usage of refactored code over non-refactored one in terms of execution. While, in case of some issue types such as *UseValueOf* and *ViewHolder*, this number was even close to 50%.
3. Regarding code comprehensibility, the overall trend seemed to be mixed *i.e.,* for certain types of performance issues like *DrawAllocation* and *UseSparseArrays*, the majority of participants felt that refactored versions of code are more comprehensive than their non-refactored code versions. Whereas for some other types of issues such as *ViewHolder*, a large group of participants agreed that non-refactored code versions are more readable than refactored ones. For some issues such as *UseValueOf* and *Recycle*, the participants opted to remain neutral.
4. Finally, as a preliminarily study, we conducted an measurement-based experiment to observe the run-time performance improvements in the refactored

and non-refactored version of code. Results showed that there were no noticeable difference in terms of CPU utilization and memory consumption across all types of performance issues.

7.2 Future Research Directions

There are various future research directions that can be pursued starting from this dissertation. We describe them as follows.

- ***Identifying and analyzing performance issue through self-admitted comments*** – Future work is needed to analyze the performance issues from different aspects, such as investigating self-admitted performance comments in the source code, to understand what types of performance issues are considered by developers. More analysis of source code are needed to understand performance regression. Investigating the phenomena observed in Chapter 4 on other mobile platforms like Apple iOS or the Web is interesting as well.
- ***Exploring other types of potential performance issues through static analysis tools*** – We investigated the occurrence and evolution of performance issues using Android Lint. Future research is needed to replicate this study using other popular static analysis tools such as *FindBugs* and *PerfChecker* and compare the evolution of performance issues identified by Android Lint and other static analysis tools. Analyzing performance issues from other static analysis tools could help to discover other performance issues which can be potentially threat the app. Moreover, an extensive qualitative study is required to understand the evolution patterns emerged in this thesis. This investigation includes studying the rationale of the patterns which can heavily decrease app performance. A more fine-grained analysis is required to analyze the solutions recommended to developers in terms of understandability, effort, and change impact.
- ***Extending the auto-refactoring tool by implementing rules for other types of potential performance issue*** – The refactoring tool presented in this dissertation is able to detect seven types of performance issues. Future work will extend the tool to fix other performance issues already listed in the official Lint document [3]. This includes rules for both Java and XML files. Future efforts should implement automatic refactoring rules for the performance smells obtained from other tools that could potentially threat the app performance.

- ***Analyzing and Measuring the run-time performance in real-world Android apps*** – In this dissertation, we conducted a survey-based study to understand the performance and readability of code snippets affected by performance smells with respect to their refactored counterparts. However, future work should analyze the ALPAR more in details by providing it to industrial developers.

  Finally, we conducted a preliminarily study to investigate the impact of statically-detectable performance issues on both CPU usage and memory consumption. It would be interesting to replicate the experiment in the wild, while considering mobile apps developed by third-party developers and with a real user base.

Bibliography

[1] Android - application components. https://www.tutorialspoint.com/android/android_application_components.htm.

[2] Android floatmath documentation. https://developer.android.com/reference/android/util/FloatMath.

[3] Android lint checks. http://tools.android.com/tips/lint-checks.

[4] Android now the world's most popular operating system. https://www.csoonline.com/article/3187011/mobile-wireless/android-is-now-the-worlds-most-popular-operating-system.html.

[5] Android studio project site. http://tools.android.com/tips/lint.

[6] Basic concepts of experimental and quasi-experimental research. https://writing.colostate.edu/guides/page.cfm?pageid=1361&guideid=64.

[7] Basic parts of android application. http://www.wideskills.com/android/overview-android/principal-ingredients-android.

[8] Card sorting. https://www.usability.gov/how-to-and-tools/methods/card-sorting.html.

[9] Crawler4j website. https://github.com/yasserg/crawler4j.

[10] Drawallocation: Memory allocations within drawing code. https://stackoverflow.com/questions/27717093/drawallocation-memory-allocations-within-drawing-code.

[11] Eslint - the pluggable linting utility for javascript and jsx. https://eslint.org/.

[12] Global mobile app revenue 2021. https://www.statista.com/statistics/269024/distribution-of-the-worldwide-mobile-app-revenue-by-category/.

[13] Google play. https://en.wikipedia.org/wiki/Google_Play.

[14] How to leak a context: Handlers & inner classes. https://www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html.

[15] Kadabra - a java-to-java compilation tool for code instrumentation and transformations controlled by the lara language. http://specs.fe.up.pt/tools/kadabra/.

[16] Mining software repositories. https://en.wikipedia.org/wiki/Mining_software_repositories.

[17] Mobile operating system market share worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide.

[18] New integer vs valueof. https://stackoverflow.com/questions/2974561/new-integer-vs-valueof.

[19] pfff is a set of tools and apis to perform static analysis. https://github.com/facebookarchive/pfff.

[20] Pmd (software) - wikipedia. https://en.wikipedia.org/wiki/PMD_(software).

[21] R package effsize. https://cran.r-project.org/web/packages/effsize/effsize.pdf.

[22] Sparsearray vs hashmap. https://stackoverflow.com/questions/25560629/sparsearray-vs-hashmap.

[23] Technical report. https://zenodo.org/record/1320453#.XQ-hJOgzaM8.

[24] Thomas phd thesis. http://cs.queensu.ca/~sthomas/data/Thomas_PhdThesis.pdf.

[25] Walkmod - open source tool to fix java coding style issues. http://walkmod.com/.

[26] What are the popular types and categories of apps. https://thinkmobiles.com/blog/popular-types-of-apps/.

[27] 22 mobile stats everyone should know, 2016.

[28] Mobile app retention challenge: 75[report], 2016.

[29] Fdroid, 2017.

[30] Wikipedia page on open-source Android apps, 2017.

[31] S. Adolph, W. Hall, and P. Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, 2011.

[32] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang. Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: an experience report. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 1–12. ACM, 2016.

[33] H. Alrubaye. How does API migration impact software quality and comprehension? an empirical study. *CoRR*, abs/1907.07724, 2019.

[34] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta. Lhdiff: A language-independent hybrid approach for tracking source code lines. In *29th IEEE International Conference on Software Maintenance*, pages 230–239. IEEE, 2013.

[35] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta. Lhdiff: Tracking source code lines to support software maintenance activities. In *29th IEEE International Conference on Software Maintenance*, pages 484–487. IEEE, 2013.

[36] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[37] V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[38] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[39] G. Canfora, M. Ceccarelli, L. Cerulo, and M. D. Penta. How long does a bug survive? an empirical study. In *18th Working Conference on Reverse Engineering*, pages 191–200, 2011.

[40] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of code smells in object-oriented systems. *ISSE*, 10(1):3–18, 2014.

[41] M. Christakis and C. Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343, 2016.

[42] J. Cohen. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological bulletin*, 70(4):213, 1968.

[43] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, 21:241–257, 2011.

[44] L. Cruz and R. Abreu. Performance-based guidelines for energy efficient mobile applications. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 46–57, 2017.

[45] L. Cruz and R. Abreu. Using automatic refactoring to improve energy efficiency of android apps. *arXiv preprint arXiv:1803.05889*, 2018.

[46] L. Cruz and R. Abreu. Catalog of energy patterns for mobile applications. *Empirical Software Engineering*, 24(4):2209–2235, 2019.

[47] Dan Crawley. VentureBeat report, 2014. `http://venturebeat.com/2014/10/15/google-play-downloads-60-percent`.

[48] T. Das, M. D. Penta, and I. Malavolta. A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, pages 443–448. IEEE, 2016.

[49] O. Dunn. Multiple comparisons using rank sums. *Technometrics*, (6):241–252, 1964.

[50] B. Fling. *Mobile design and development: Practical concepts and techniques for creating mobile sites and Web apps*. O'Reilly Media, Inc., 2009.

[51] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 159–168. IEEE Press, 2015.

[52] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[53] I. Gartner. Gartner says free apps will account for nearly 90 percent of total mobile app store downloads in 2012, 2012. `http://www.gartner.com/newsroom/id/2153215`.

[54] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier. Mining test repositories for automatic detection of UI performance regressions in android apps. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 13–24, 2016.

[55] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in android applications. In *28th IEEE/ACM International Conference on Automated Software Engineering*, pages 389–398, 2013.

[56] S. Habchi, X. Blanc, and R. Rouvoy. On adopting linters to deal with performance concerns in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 6–16, 2018.

[57] S. Habchi, G. Hecht, R. Rouvoy, and N. Moha. Code smells in ios apps: How do they compare to android? In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 110–121, 2017.

[58] S. Habchi, R. Rouvoy, and N. Moha. On the survival of android code smells in the wild. In *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems*, pages 87–98, 2019.

[59] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 38–47. IEEE, 2017.

[60] M. Hataba, A. El-Mahdy, and K. Ueda. Generation of efficient obfuscated code through just-in-time compilation. *IEICE Transactions*, 102-D(3):645–649, 2019.

[61] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien. Tracking the software quality of android applications along their evolution (t). In *30th IEEE/ACM International Conference on Automated Software Engineering*, pages 236–247. IEEE, 2015.

[62] G. Hecht, N. Moha, and R. Rouvoy. An empirical study of the performance impacts of android code smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 59–69, 2016.

[63] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. Detecting antipatterns in android apps. In *2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 148–149, 2015.

[64] S. Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal on Statistics*, 6:65–70, 1979.

[65] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[66] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering*, pages 672–681, 2013.

[67] W. D. Jones. Reliability models for very large software systems in industry. In *Second International Symposium on Software Reliability Engineering*, pages 35–42, 1991.

[68] P. Jonsson, S. Carson, K. h. Per Lindberg, I. Sorlie, R. Queirs, L. E. Frank Muller, M. Arvedson, and A. Carlsson. Ericsson mobility report. November 2018.

[69] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, 2013.

[70] H. H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 19(2):77–131, 2007.

[71] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016.

[72] H. Khalid, M. Nagappan, and A. E. Hassan. Examining the relationship between findbugs warnings and app ratings. *IEEE Software*, 33(4):34–39, 2016.

[73] S. Kim and M. D. Ernst. Which warnings should I fix first? In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 45–54, 2007.

[74] S. Kim and E. J. W. Jr. How long did it take to fix bugs? In *Proceedings of the International Workshop on Mining Software Repositories*, pages 173–174, 2006.

[75] W. H. Kruskal and A. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, (47):583–621, 1952.

[76] M. Lamothe and W. Shang. Exploring the use of automated API migrating techniques in practice: an experience report on android. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 503–514, 2018.

[77] D. Leffingwell. *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley Professional, 2010.

[78] A. Lella and A. Lipsman. The 2017 u.s. mobile app report. *comScore*, 2017.

[79] A. Lella, A. Lipsman, and B. Martin. The Global Mobile Report: How Multi-Platform Audiences and Engagement Compare in the US, Canada, UK and Beyond, 2015. comsCore white paper.

[80] D. Li, Y. Lyu, J. Gui, and W. G. Halfond. Automated Energy Optimization of HTTP Requests for Mobile Applications. In *Proceedings of the 38th International Conference on Software Engineering*, May 2016.

[81] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.

[82] W. Lidwell, K. Holden, and J. Butler. *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design.* Rockport Publishers, 2nd edition, January 2010.

[83] Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *36th International Conference on Software Engineering*, pages 1013–1024, 2014.

[84] Y. Liu, C. Xu, and S. C. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *IEEE International Conference on Pervasive Computing and Communications*, pages 2–10, 2013.

[85] I. Malavolta, S. Ruberto, V. Terragni, and T. Soru. End users perception of hybrid mobile apps in the google play store. In *IEEE International Conference on Mobile Services*, pages 25–32. (IEEE), June 2015.

[86] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the International Conference on Software Engineering*, pages 1012–1021. IEEE Press, 2013.

[87] M. Mandel and E. Long. The app economy in europe: Leading countries and cities. *Practically pragmatic*, 2017.

[88] A. Mazuera-Rozo, C. Trubiani, M. Linares-Vsquez, and G. Bavota. Investigating types and survivability of performance bugs in mobile apps. *Empirical Software Engineering*, to be appear.

[89] I. Moura, G. Pinto, F. Ebert, and F. Castor. Mining energy-aware commits. In *12th IEEE/ACM Working Conference on Mining Software Repositories*, pages 56–67, 2015.

[90] J. C. Munson and S. G. Elbaum. Code churn: A measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance*, pages 24–31. IEEE, 1998.

[91] A. Nistor and L. Ravindranath. Suncat: helping developers understand and predict performance problems in smartphone applications. In *International Symposium on Software Testing and Analysis*, pages 282–292, 2014.

[92] F. Palomba, D. D. Nucci, A. Panichella, A. Zaidman, and A. D. Lucia. Lightweight detection of android-specific code smells: The adoctor project. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, pages 487–491, 2017.

[93] F. Palomba, D. D. Nucci, A. Panichella, A. Zaidman, and A. D. Lucia. On the impact of code smells on the energy consumption of mobile applications. *Information & Software Technology*, 105:43–55, 2019.

[94] L. Pascarella, F.-X. Geiger, F. Palomba, D. Di Nucci, I. Malavolta, and A. Bacchelli. Self-Reported Activities of Android Developers. In *5th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, page to appear. ACM, May 2018.

[95] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Tenth ACM Workshop on Hot Topics in Networks (HotNets-X)*, page 5, 2011.

[96] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *The 10th International Conference on Mobile Systems, Applications, and Services*, pages 267–280, 2012.

[97] M. D. Penta, L. Cerulo, and L. Aversano. The life and death of statically detected vulnerabilities: An empirical study. *Information & Software Technology*, 51(10):1469–1484, 2009.

[98] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 321–334, 2011.

[99] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *10th USENIX Symposium on Operating Systems Design and Implementation*, pages 107–120, 2012.

[100] B. Ray, D. Posnett, V. Filkov, and P. T. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165, 2014.

[101] J. Reimann, M. Brylski, and U. Aßmann. A tool-supported quality smell catalogue for android developers. *Softwaretechnik-Trends*, 34(2), 2014.

[102] J. Rosenberg. Statistical methods and measurement. In *Guide to Advanced Empirical Software Engineering*, pages 155–184. Springer, 2008.

[103] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[104] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 61–72. ACM, 2016.

[105] F. Shull, J. Singer, and D. I. Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2007.

[106] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136. ACM, 2006.

[107] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[108] M. Sulír and J. Porubän. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–25. ACM, 2016.

[109] K. F. Tómasdóttir, M. F. Aniche, and A. van Deursen. Why and how javascript developers use linters. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 578–589, 2017.

[110] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Software Eng.*, 43(11):1063–1088, 2017.

[111] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, Sept. 2019. DOI: 10.1145/3340544.

[112] M. L. Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. D. Penta, and D. Poshyvanyk. Mining energy-greedy API usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11, 2014.

[113] M. L. Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *IEEE International Conference on Software Maintenance and Evolution*, pages 352–361, 2015.

[114] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli. A large-scale empirical exploration on refactoring activities in open source software projects. *Sci. Comput. Program.*, 180:1–15, 2019.

[115] C. Vassallo, S. Proksch, H. C. Gall, and M. D. Penta. Automated reporting of anti-patterns and decay in continuous integration. In *Proceedings of the 41st International Conference on Software Engineering*, pages 105–115, 2019.

[116] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal. Towards verifying android apps for the absence of no-sleep energy bugs. In *Workshop on Power-Aware Computing Systems*, 2012.

[117] M. Wedel, U. Jensen, and P. Göhner. Mining software code repositories and bug databases using survival analysis models. In *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement*, pages 282–284, 2008.

[118] F. Wedyan, D. Alrmuny, and J. M. Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *International Conference on Software Testing Verification and Validation*, pages 141–150. IEEE, 2009.

[119] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Computer Science. Springer, 2012.

[120] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th Working Conference on Mining Software Repositories*, pages 199–208. IEEE Press, 2012.

[121] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. A. Dinda, and L. Yang. ADEL: an automatic detector of energy leaks for smartphone applications. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 363–372, 2012.