Understanding Mobile-Specific Code Smells

Sarra Habchi

▶ To cite this version:

Sarra Habchi. Understanding Mobile-Specific Code Smells. Software Engineering [cs.SE]. Université de Lille, 2019. English. NNT : . tel-02414928

HAL Id: tel-02414928
https://theses.hal.science/tel-02414928v1

Submitted on 6 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Understanding Mobile-Specific Code Smells

Sarra Habchi

Supervisor: Prof. Romain Rouvoy

Inria Lille
University of Lille

Jury:
Referees: Prof. Andy Zaidman and Prof. Yann Gaël Guéhéneuc.
Examiners: Prof. Jean-Christophe Routier, Prof. Naouel Moha, and Dr. Tegawendé Bissyande.

December 4, 2019

Comprendre les défauts de code spécifiques aux applications mobiles

Sarra Habchi

Directeur de thèse: Pr. Romain Rouvoy

Inria Lille
Université de Lille

Jury:
Rapporteurs: Pr. Andy Zaidman et Pr. Yann Gaël Guéhéneuc.
Examinateurs: Pr. Jean-Christophe Routier (président), Pr. Naouel Moha, et Dr. Tegawendé Bissyande.

Thèse soutenue le 4 décembre 2019

Acknowledgements

I would like to express my gratitude to all the people that have contributed to the realization of this thesis.

Foremost, I would like to thank my advisor, Romain Rouvoy, for offering me the opportunity to carry out this doctoral work. Thank you for the many advices and discussions that helped me achieve this work. I also want to thank for your moral support that helped me overcome the PhD challenges. I truly appreciate all your efforts to make these three years an enjoyable experience for me.

Besides my advisor, I would like to thank the members of my thesis committee : Yann-Gaël Guéheneuc, Andy Zaidman, Tegawendé Bissyande, and Naouel Moha for their time and feedback. I want to particularly thank the reviewers, Andy and Yann-Gaël, for their detailed reviews and their recommendations that helped in improving the quality of my manuscript.

I wish to show my gratitude to the current and former members of the Spirals team. Thanks for the witty conversations at lunch, the generous pots, and the friendliness. I want to particularly thank Antoine Veuiller for his enormous contribution in building the Sniffer toolkit and his engineering support. Thanks to our team leader, Lionel Seinturier, for his excellent direction of the team and his disposal to help us. Also, I wish to thank Laurence Duchien for her wise career advices, encouragement, and above all her kindness.

I am also grateful to all the researchers that worked with me during this thesis. Thanks to Xavier Blanc for his interesting insights and critical views that helped in my qualitative study. Thanks to Naouel Moha for the collaboration and for welcoming me during two research visits.

Last but by no means least, thanks to my family and friends who have provided me through moral and emotional support in my life.

Abstract

Object-oriented code smells are well-known concepts in software engineering. They refer to bad design and development practices commonly observed in software systems. With the emergence of mobile apps, new classes of code smells have been identified to refer to bad development practices that are specific to mobile platforms. These mobile-specific code smells differ from object-oriented ones by pertaining to performance issues reported in the documentation or developer guidelines. Since their identification, many research works studied mobile-specific code smells to propose detection tools and study them. Most of these studies focused on measuring the performance impact of such code smells and did not provide any insights about their motives and potential solutions. In particular, we lack knowledge about (i) the rationales behind the accrual of mobile code smells, (ii) the developers' perception of mobile code smells, and (iii) the generalizability of code smells across different mobile platforms. These lacks hinder the understanding of mobile code smells and consequently prevent the design of adequate solutions for them.

Therefore, in this thesis we conduct a series of empirical studies to better understand mobile code smells. First, we study the prevalence of code smells in different mobile platforms. Then, we conduct a large-scale study to analyze the change history of mobile apps and discern the factors that favor the introduction and survival of code smells. To consolidate these studies, we also perform a user study to investigate developers' perception of code smells and the adequacy of static analyzers as a solution for coping with them. Finally, we perform a qualitative study to question the established foundation about the definition and detection of mobile code smells. The results of these studies revealed important research findings. Notably, we showed that pragmatism, prioritization, and individual attitudes are not relevant factors for the accrual of mobile code smells. The problem is rather caused by ignorance and oversight, which are prevalent among mobile developers. Furthermore, we highlighted several flaws in the code smell definitions currently adopted by the research community. These results allowed us to elaborate some recommendations for researchers and tool makers to design detection and refactoring tools for mobile code smells. Our results opened perspectives for research works about the identification of mobile code smells and development practices in general.

Abstract

Au cours des dernières années, les applications mobiles sont devenues indispensables dans notre vie quotidienne. Ces applications ont pour particularité de fonctionner sur des téléphones mobiles, souvent limités en ressources (mémoire, puissance de calcul, batterie, etc). Ainsi, il est impératif de surveiller leur code source afin de s'assurer de l'absence de défauts de code, c.à.d., des pratiques de développement qui dégradent la performance. Plusieurs études ont été menées pour analyser les défauts de code des applications mobile et évaluer leur présence et leur impact sur la performance et l'efficience énergétique. Néanmoins, ces études n'ont pas examiné les caractéristiques et les motifs de ces défauts alors que ces éléments sont nécessaires pour la compréhension et la résolution de ce phénomène.

L'objectif de cette thèse est de répondre à ce besoin en apportant des connaissances sur les défauts de code mobile. Pour cela, nous avons mené diverses études empiriques à caractère quantitatif et qualitatif pour comprendre ce phénomène et ses possibles résolutions. Les résultats de ces études montrent que les défauts de code mobile ne sont pas dus au pragmatisme, à la priorisation, ou aux comportements individuels des développeurs. En effet, l'essentielle raison derrière l'accumulation des défauts de code est l'ignorance générale de ces pratiques parmi les développeurs. Nos études ont aussi montré que le linter peut être un outil adéquat pour l'analyse des défauts de code et l'amélioration des performances. Cependant, certaines défauts nécessitent une analyse dynamique pour une détection plus précise. Enfin, nous avons montré qu'il existe un écart entre les défauts de code étudiés par la communauté scientifique et les réelles mauvaises pratiques adoptées par les développeurs d'applications mobiles. Pour remédier à cet écart, nous recommandons à la communauté d'impliquer les développeurs dans le processus d'identification de défauts de code.

Contents

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Context . . . 1

1.2 Problem Statement . . . 2

1.2.1 Problem 1: Code Smells in Different Mobile Platforms . . . 4

1.2.2 Problem 2: The Motives of Mobile-Specific Code Smells . . . 4

1.2.3 Problem 3: Developers' Perception of Mobile-Specific Code Smells . . 4

1.3 Contributions . . . 5

1.3.1 Contribution 1: Study of the iOS Platform . . . 5

1.3.2 Contribution 2: Discerning the Motives of Mobile-Specific Code Smells 5

1.3.3 Contribution 3: Study of Developers Perception of Mobile-Specific Code Smells . . . 6

1.3.4 Contribution 4: A Reevaluation of Mobile-Specific Code Smells 6

1.4 Outline . . . 7

1.5 Publications . . . 8

1.5.1 Published . . . 8

1.5.2 Under Evaluation . . . 8

2 State of the Art 9

2.1 Code Smells in Mobile Apps . . . 10

2.1.1 Identification of Mobile-Specific Code Smells . . . 10

2.1.2 Detection of Mobile-Specific Code Smells . . . 14

2.1.3 Studies About Mobile-Specific Code Smells . . . 17

2.1.4 Studies About OO Code Smells . . . 19

2.1.5 Summary . . . 21

2.2 Management in Practice . . . 21

2.2.1 Performance Management . . . 21

2.2.2 Linters . . . 22
2.2.3 Linter Studies . . . 24
2.2.4 Summary . . . 24
2.3 Conclusion . . . 24

3 Code Smells in iOS Apps 27
3.1 Background on iOS Apps Analysis . . . 28
3.1.1 Dynamic Analysis . . . 28
3.1.2 Static Analysis of Binaries . . . 28
3.1.3 Static Analysis of Source Code . . . 29
3.2 Code Smells in iOS Apps . . . 29
3.2.1 Code Smells Identification Process . . . 29
3.2.2 Catalog of iOS Code Smells . . . 29
3.2.3 Similarities with Android Code Smells . . . 32
3.3 Sumac . . . 32
3.3.1 Code Analysis . . . 32
3.3.2 Sumac Model for iOS . . . 33
3.3.3 Storing the iOS Model . . . 33
3.3.4 Code Smell Queries . . . 33
3.4 Study of Code Smells in iOS . . . 35
3.4.1 Objects . . . 35
3.4.2 iOS Dataset and Inclusion Criteria . . . 36
3.4.3 Hypotheses and Variables . . . 36
3.4.4 Analysis Method . . . 36
3.4.5 Results . . . 37
3.4.6 Threats to Validity . . . 41
3.5 Comparative Study Between iOS and Android . . . 42
3.5.1 Android Dataset and Inclusion Criteria . . . 42
3.5.2 Variables and Hypotheses . . . 42
3.5.3 Analysis Method . . . 43
3.5.4 Results . . . 43
3.5.5 Threats to Validity . . . 46
3.6 Summary . . . 47

4 Code Smells in the Change History 49
4.1 Study Design . . . 51
4.1.1 Context Selection . . . 51
4.1.2 Dataset and Selection Criteria . . . 53
4.2 Data Extraction: Sniffer . . . 54

4.2.1 Validation . . . 59
4.3 Evolution Study . . . 60
4.3.1 Data Analysis . . . 60
4.3.2 Study Results . . . 64
4.3.3 Discussion & Threats . . . 72
4.3.4 Study Summary . . . 74
4.4 Developers Study . . . 74
4.4.1 Data Analysis . . . 75
4.4.2 Study Results . . . 78
4.4.3 Discussion . . . 84
4.4.4 Study Summary . . . 86
4.5 Survival Study . . . 86
4.5.1 Data Analysis . . . 87
4.5.2 Study Results . . . 90
4.5.3 Discussion . . . 98
4.5.4 Study Summary . . . 100
4.6 Common Threats to Validity . . . 100
4.7 Summary . . . 101

5 Developers' Perception of Mobile Code Smells 103
5.1 Methodology . . . 105
5.1.1 Interviews . . . 105
5.1.2 Participants . . . 106
5.1.3 Analysis . . . 108
5.2 Results . . . 111
5.2.1 Why Do Android Developers Use Linters for Performance Purposes? . 111
5.2.2 How Do Android Developers Use Linters for Performance Purposes? . 115
5.2.3 What Are the Constraints of Using Linters for Performance Purposes? 118
5.3 Implications . . . 121
5.3.1 For Developers . . . 121
5.3.2 For Researchers . . . 122
5.3.3 For Tool Creators . . . 122
5.4 Threats To Validity . . . 123
5.5 Summary . . . 124

6 A Second Thought About Mobile Code Smells 127
6.1 Study Design . . . 128
6.1.1 Data Selection . . . 128
6.1.2 Data Analysis . . . 129

6.2 Study Results 132
6.2.1 Leaking Inner Class (LIC) 132
6.2.2 Member Ignoring Method (MIM) 135
6.2.3 No Low Memory Resolver (NLMR) 137
6.2.4 HashMap Usage (HMU) 139
6.2.5 UI Overdraw (UIO) 140
6.2.6 Unsupported Hardware Acceleration (UHA) 142
6.2.7 Init OnDraw (IOD) 143
6.2.8 Unsuited LRU Cache Size (UCS) 144
6.2.9 Common Theory 146
6.3 Discussion 147
6.3.1 Implications 147
6.3.2 Threats to Validity 148
6.4 Summary 148

7 Conclusion and Perspectives 151
7.1 Summary of Contributions 151
7.2 Short-Term Perspectives 153
7.2.1 Rectify Code Smell Definitions 153
7.2.2 Improve Code Smell Detection 153
7.2.3 Reevaluate the Impact of Code Smells 154
7.2.4 Integrate a Code Smell Detector in the Development Environment . . 154
7.2.5 Communicate About Code Smells 154
7.3 Long-Term Perspectives 155
7.3.1 Involve Developers in Code Smell Identification 155
7.3.2 Leverage Big Code in Code Smell Identification 156
7.4 Final Words 157

Bibliography 159

Appendix A Sumac: A Code Smell Detector for iOS Apps 173

Appendix B A Preliminary Survey for the Linter Study 177

List of Figures

2.1 Structure of our state of the art. 9

4.1 Overview of the Sniffer toolkit. 54
4.2 An example of a commit tree. 56
4.3 Commits ordered in different settings. 57
4.4 Tracking code smell renamings. 58
4.5 Distribution of code smell %diffuseness in studied apps. 65
4.6 The evolution of code smells in the Seafile Android client app. 66
4.7 The number of code smell introductions and removals per commit in the last 100 commits before release. 67
4.8 The density function of code smell introductions and removals one day, one week, and one month before releasing. 67
4.9 Intersection between introducers and removers. 79
4.10 The distribution of the number of commits among the studied developers. . . . 79
4.11 The distribution of the number of followers among studied developers. 83
4.12 Survival curves in days. 91
4.13 Code smell lifetime in terms of days. 91
4.14 Survival curves in effective commits. 93
4.15 Code smell lifetime in terms of effective commits. 93
4.16 The estimated hazard coefficients for #Commits, #Developers, and #Classes. 95
4.17 The hazard coefficient of #Releases and Cycle. 96
4.18 The impact of the presence in the linter on survival. 96
4.19 The impact of linter priority on code smell survival. 97
4.20 The impact of granularity on code smell survival. 98

6.1 Design of the qualitative analysis. 130
6.2 A sunburst of *LIC* instances in practice. 133

B.1 The form used to prepare the linter study (Part 1). 178
B.2 The form used to prepare the linter study (Part 2). 179

B.3 Responses about the linter usage. 180
B.4 Responses about the adopted used Android linters. 180
B.5 Responses about the reasons for using an Android linter. 181
B.6 The respondents' experience in Android. 181
B.7 The respondents' expertise in Android. 182
B.8 The context of app development. 182
B.9 Comments about the linter. 183

List of Tables

2.1 The code smells detailed by Reimann *et al.* [169]. 11
2.2 Identification of mobile-specific issues 13
2.3 Tools and techniques for detecting mobile-specific issues 16
2.4 Studies about mobile-specific code smells. 19

3.1 Mapping Cohen's d to Cliff's δ. 37
3.2 Percentage of apps affected by code smells. 38
3.3 Ratios comparison between Objective-C and Swift. 39
3.4 Metrics comparison between Objective-C and Swift. 41
3.5 Percentage of Android apps affected by smells. 43
3.6 Ratios comparison between Android and iOS 44
3.7 Metrics comparison between iOS and Android. 46

4.1 Studied code smells. 53
4.2 Repository metadata. 55
4.3 Validation of SNIFFER. 59
4.4 Study metrics. 61
4.6 Numbers of code smell introductions. 64
4.7 Compare #commit-introductions in commits authored one day, one week, and one month before releasing. 68
4.8 Compare #commit-removals in commits authored one day, one week, and one month before releasing. 68
4.9 Number and percentage of code smell removals. 69
4.10 Developer metrics. 76
4.12 Distribution of developers according to code smell introductions and removals. 78
4.13 Correlation between introductions and removals per code smell type. 81
4.14 Comparison between newcomers and regular developers in terms of introduction and removal rates. 81
4.15 The numbers of followers among the studied developers. 82

4.16 Comparison between infamous and famous developers in term of code smell introduction and removal rates. . . . 83
4.17 Metrics for *RQ2*. . . . 87
4.19 The values of *Linted*, *Popularity*, and *Granularity* per code smell type. . . . 89
4.20 Kaplan Meier survival in days. . . . 92
4.21 Kaplan Meier survival in effective commits. . . . 93

5.1 Partcipants' code name, years of experience in Android and Android Lint, and the type of projects they work on 107
5.2 Results of the coding process. . . . 110

6.1 Studied code smells. . . . 129
6.2 Distribution of *MIM* instances according to the method content. . . . 136
6.3 Distribution of *HMU* instances according to HashMap key type. . . . 139
6.4 Alternative means used in *UIO* instances to avoid unnecessary drawings. . . . 141
6.5 Common flaws in code smell definitions. . . . 146

7.1 The short and long-term perspectives of our thesis 157

A.1 The entities used by SUMAC 173
A.2 The metrics used by SUMAC for analyzing Objective-C apps. . . . 174
A.3 The metrics used by SUMAC for analyzing Swift apps. . . . 175

Chapter 1

Introduction

1.1 Context

In 2007, the market of mobile phones was shaken by the unveiling of the original iPhone. Smartphones already existed and detained a small share of the mobile phone market before the iPhone. Examples of these smartphones are the BlackBerry, Nokia E62, and Motorola Q, which were mainly intended for business professionals. However, the iPhone was considered as a *game-changer* thanks to its revolutionary hardware and software features. In particular, it replaced the physical hardware buttons and the stylus by a touchscreen and proposed new built-in software, like a Safari browser and a YouTube client. These features allowed smartphones to reach a wider market. Indeed, the original iPhone sold 6.1 million units and launched an unprecedented growth in the market of mobile devices [202]. Tens of mobile devices were launched afterwards to offer new capabilities and choices to end-users. Notably, in 2008, the HTC Dream was announced as the first smartphone to feature an Android OS, and in 2010, the first iPad was unveiled. These continuous improvements allowed mobile devices to successfully conquer the market and retain the interest of end-users. During 2018 alone, 1.4 billion smartphones were sold and 173 million tablets were shipped [33].

This market expansion led to the development of a new software ecosystem to accompany this new generation of mobile devices. In particular, several operating systems were developed to run on mobile devices, notably, Android and iOS, but also other systems like Windows 10 Mobile, Fuchsia, and HarmonyOS. Android and iOS provided their own *software development kits* (SDK) and app markets to facilitate the creation of mobile apps for their operating systems. These SDKs provide the essential components to build a mobile app that meets the needs of a mobile user. On top of classical software development tasks, these components facilitate the management of the *user interface* (UI), user inputs, and sensors. The app markets allow developers to easily commercialize their apps and reach end-users. Thanks to

these facilities, Google Play Store and Apple App Store count today more than 4.4 million apps and 245 billion app downloads [180–182].

These numbers offered a wide variety of choice for end users and also created a competitive atmosphere among apps. Developers must constantly ship new and better apps or features to gain users and keep them satisfied. Reports suggest that satisfying mobile users can be challenging as 75% of the mobile apps are uninstalled within three months [62]. To face this challenge, developers have to pay careful attention to the quality of their mobile apps. In particular, as mobile apps run on mobile devices, they are constrained by hardware specificities:

- Mobile devices have limited capacities in terms of memory, storage, and battery compared to personal computers. Some aspects of this limitation are disappearing as manufacturers keep improving their devices. The current flagship devices propose $32GB$ RAM and $1TB$ storage [162, 163]. This advancement faces the increasing requirements of mobile users who expect to use their devices for more sophisticated tasks all day long;
- Another hardware particularity is the sensors integrated in the mobile device. These sensors allow the system and apps to collect data about the environment of the device like motion, orientation, and localization. Today, on average, a mobile device is equipped with 14 sensors [68]. The use of these sensors can be very costly especially in terms of battery.

These hardware specificities set the bar high for mobile apps in terms of performance. Specifically, apps are expected to perform complex tasks on devices with limited capacities, manage battery-draining sensors, and remain fluid and efficient to satisfy users. Hence, performance can be considered as a key challenge of mobile development. Indeed, a user survey showed that the second top reason of app uninstall is poor performance [6].

1.2 Problem Statement

The research community showed an increasing interest for the topic of performance in mobile apps. Many studies focused on identifying the development practices that hinder the app performance. These studies covered practices related to network operations, sensor utilization, and UI management [126, 159, 169, 208]. They also followed different approaches to identify bad practices. Some studies relied on the manual analysis of source code, while others relied on the documentation and blogs. In this regard, the study of Reimann *et al.* [169] stands out by its reliance on diverse data sources and its resulting catalog, which included 30 bad practices. These practices were dedicated to the Android platform and were qualified as *mobile-specific code smells*.

The concept of code smells is well-known in *Object-oriented* (OO) software systems. Specifically, OO code smells describe poor design or coding practices that negatively impact

software maintainability and cause long-term problems. These code smells were identified by Fowler [69] in 1999, and since then the research community developed an important knowledge about them. Specifically, studies investigated and quantified their presence in the source code and proposed tools to detect them [26, 139]. Research works also inspected their impact on different maintainability aspects, like change proneness, and showed that they represent a real concern for software systems [2, 67, 116]. Furthermore, researchers analyzed the evolution of code smells in the change history (project commits) to identify the rationales behind their introduction and removal in the source code [161, 196]. This inspection was complemented by qualitative studies that highlighted the developers' stance about code smells [155, 203]. These studies helped in improving our understanding of code smells and highlighted notable facts, like developers' lack of awareness about code smells. This understanding allowed studies to design solutions and tools adequate for OO code smells [193, 195].

Mobile-specific code smells are different from OO code smells, since they often refer to a misuse of the mobile platform SDK and they are more performance-oriented [169]. According to the guidelines of mobile frameworks, they may hinder memory, CPU, and energy performances [7, 9, 137, 143]. Hence, our existing knowledge about OO code smells may not apply to mobile-specific code smells. Aware of this difference, research studies inspected some aspects of mobile-specific code smells. In particular, different research and community tools were proposed to detect these code smells in mobile apps [9, 109, 156, 169]. Using these tools, researchers quantified the presence of mobile-specific code smells and showed that they are very common in Android apps [108, 157]. Empirical studies also assessed the impact of these code smells on app performance, demonstrating that their refactoring can significantly improve performance and energy efficiency [40, 107, 140, 157].

Apart from performance, other aspects of mobile-specific code smells remained unaddressed. Specifically, we still lack knowledge about their motives—*i.e.*, the rationales behind their appearance and accrual in the source code. We also lack qualitative understanding of code smells and insights from developers about mobile code smells. Similarly to OO code smells, understanding these aspects is important for designing adequate solutions and tools for mobile-specific code smells. Another aspect of mobile-specific code smells that remains unaddressed is the platform impact. Mobile-specific code smells are tightly related to mobile platforms and their SDK. Nevertheless, these platforms manifest several differences in their development frameworks, documentations, and communities. Hence, it is important to inspect multiple mobile platforms and study their relations with mobile-specific code smells.

In this thesis, we aim to address these lacks and build a better understanding of mobile-specific code smells. We believe that this understanding is helpful for the research community working on mobile apps and their performance, but also to the research community of code smells and technical debt in general. Moreover, understanding the motives of code

smells is necessary for tool makers who aspire to build tools that meet the needs of software developers.

The objective of this thesis is to build an understanding of mobile-specific code smells that helps in the design of adequate tools and solutions for them.

To build this understanding, we address in this thesis different research problems. We detail in the following these problems and the motivation behind our focus on them.

1.2.1 Problem 1: Code Smells in Different Mobile Platforms

All the studies of mobile-specific code smells focused on the Android platform. This focus may be justified as this platform is popular and easy to study compared to iOS. Nonetheless, mobile platforms manifest important differences in their development frameworks and runtime environments. These differences may impact the nature of code smells but also their extent in the source code. Hence, comparative studies between different mobile platforms can be helpful in understanding mobile code smells. On top of that, these studies can help in identifying common code smells between multiple mobile platforms and therefore allow us to generalize future studies and tools.

1.2.2 Problem 2: The Motives of Mobile-Specific Code Smells

The existing research studies did not explain the motives behind the accrual of mobile-specific code smells. One of the objectives of this thesis is to provide this explanation. In this regard, the literature of technical debt suggests that the potential motives behind code smells are (i) internal and external factors like pragmatism and prioritization, (ii) development processes, and (iii) developer motives like attitudes, ignorance, and oversight. We aim to assess the relevance and impact of these motives in the case of mobile-specific code smells. This assessment is important for all future approaches and tools to cope with these code smells.

1.2.3 Problem 3: Developers' Perception of Mobile-Specific Code Smells

Aware of the importance of developers in the topic of code smells, previous studies captured developers' perception of OO code smells. Notably, the survey of Palomba *et al.* [155] showed that intuitive code smells that are related to complex or long source code are generally perceived as an important threat by developers. On the other hand, code smells that are related to OO concepts rather than explicit complexity are less perceived by developers. Interestingly, mobile-specific code smells may not be intuitive, as they rely on technical practices related to the SDK. Hence, mobile developers may be unaware of these code smells and thus introduce them in their software systems. Nevertheless, research studies about mobile-specific code smells did not investigate this hypothesis. Therefore, in

this thesis we aim to understand how developers perceive code smells that are specific to mobile platforms.

1.3 Contributions

To address the aforementioned problems, we conduct in this thesis multiple studies that contribute to the research about mobile apps and code smells. We summarize these contributions in the following subsections.

1.3.1 Contribution 1: Study of the iOS Platform

The first contribution of this thesis is a study of code smells in the iOS platform. In this study, we propose six iOS-specific code smells that we identified from developers' discussions and the platform's official documentation. We summarize these code smells in a structured catalog and we emphasize the similarities between our iOS code smells and other Android-specific smells. We also propose SUMAC, the first code smell detector for iOS apps. SUMAC is able to detect seven code smells in apps written in Objective-C or Swift. This study covers 279 iOS apps and 1,551 Android apps and highlights several findings. In particular, it shows that some code smells can be generalized to both mobile platforms Android and iOS. Moreover, it suggests that the presence of code smells is more related to the mobile platform than the programming language.

1.3.2 Contribution 2: Discerning the Motives of Mobile-Specific Code Smells

The second contribution of our thesis is a series of empirical studies that analyze the change history to discern the motives behind mobile-specific code smells. To perform these studies, we design and build SNIFFER, a novel toolkit that tracks the full history of Android-specific code smells. Our toolkit tackles many issues raised by the Git mining community, like branch and renaming tracking [120]. Using SNIFFER, we perform three empirical studies:

— **Evolution study:** This study analyzes the introductions and removals of mobile-specific code smells aiming to understand how prioritization factors impact them. The study also analyzes the actions leading to code smell removals to inspect motives like pragmatism, ignorance, and oversight;

— **Developers study:** This study intends to understand the role played by developers in the accrual of mobile-specific code smells. This study covers the three motives that are developer-oriented, namely attitude, ignorance, and oversight;

— **Survival study:** This study investigates the survival of mobile-specific code smells in the change history. In particular, it analyzes the impact of processes and prioritization factors on the persistence of code smells in the codebase.

These studies cover eight Android code smells, 324 Android apps, 255k commits, and contributions from 4,525 developers. The results of these studies shed light on many potential rationales behind the accrual of Android code smells. In particular, they show that:

— Pragmatism and prioritization choices are not relevant motives in the case of Android code smells;
— Individual attitudes of Android developers are not the reason behind the prevalence of code smells. The problem is rather caused by the ignorance and oversight, which seem to be common among the studied developers;
— The processes adopted by development teams (*e.g.*, the use of tools and their configuration) can help in limiting the presence of code smells.

1.3.3 Contribution 3: Study of Developers Perception of Mobile-Specific Code Smells

To address the lack of comprehension of developers' perception of mobile-specific code smells, we conduct a qualitative study about the usage of Android Lint by developers. We focus on Android Lint because it is a widely adopted tool that can be considered as a medium between developers and code smells. Indeed, it is the most used linter for Android, and it detects a large set of mobile-specific code smells. Our study includes interviews with 14 experienced Android developers. The results of these interviews provide insights about developers' perception of Android code smells:

— There is a prevalent ignorance of mobile-specific code smells among developers;
— The mindset of managing performance reactively is very common. Hence, future studies should adapt to this mindset and focus on reactive approaches and tools for dealing with code smells like profilers;
— Some developers challenge the relevance and impact of performance bad practices.

Our results also provide many indications about the usage of linters for performance:

— Linters can be useful for anticipating and debugging performance bottlenecks and they can also be used to raise performance culture among developers;
— Besides the intuitive proactive approach for using linters, developers can also use linters reactively in performance sprints;
— Linters should be more clear and explicit about the categories of checks. Clarity is also required in the explanations of the potential impact of performance checks.

1.3.4 Contribution 4: A Reevaluation of Mobile-Specific Code Smells

The last contribution of our thesis is an exploratory study inspecting the characteristics of mobile-specific code smell instances. In this study, we rely on the well founded approach of Grounded Theory [90], which allows us to build a theory about mobile-specific code smells. This theory provides second thoughts about the established foundation about code smell

definition and detection. Specifically, it highlights the following flaws in the definitions of Android code smells:

— The code smell definitions for *No Low Memory Resolver*, *UI Overdraw*, and *HashMap Usage* omit critical technical details and generate many false positives;
— The current definitions of *Leaking Inner Class* and *Member Ignoring Method* consider prevalent practices related to prototyping and the usage of listeners as performance issues. This leads to confusing code smell instances that may be hard to refactor for developers;
— The definition of *Unsupported Hardware Acceleration* considers `Path` drawing, which does not have exact alternatives in Android, as a bad practice. This generates confusing code smell instances that may seem inevitable for developers.

The theory also shows that the techniques currently used by research tools for detecting *UI Overdraw* and *Unsupported Hardware Acceleration* exhibit many flaws that can hinder the study of these code smells.

1.4 Outline

The remainder of this dissertation is composed of five chapters as follows:

Chapter 2: State of the Art: This chapter presents the most relevant works for our topic. This includes works about the identification, detection, and impact of mobile-specific code smells. It covers studies related to the management of performance in software systems.

Chapter 3: Code Smells in iOS Apps: This chapter presents a study of code smells in the iOS platform. This chapter is a revised version of the following paper:

— Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. Code smells in iOS apps: How do they compare to Android? In *Proceedings of the 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2017.

Chapter 4: Code Smells in the Change History: This chapter synthesizes our investigation of mobile-specific code smells in the change history to understand their motives. This chapter is a revised version of the following papers:

— Sarra Habchi, Romain Rouvoy, and Naouel Moha. Android Code Smells: From Birth to Death, and Beyond. In *Journal of Systems and Software*, 2019—Under evaluation.
— Sarra Habchi, Naouel Moha, and Romain Rouvoy. The rise of Android code smells: Who is to blame? In *Proceedings of the 16th International Conference on Mining Software Repositories*, 2019.

— Sarra Habchi, Romain Rouvoy, and Naouel Moha. On the survival of Android code smells in the wild. In *Proceedings of the 6th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2019.

Chapter 5: Developers' Perception of Mobile Code Smells: This chapter presents a qualitative study about developers' perception of code smells and the adequacy of linters as a solution for them. This chapter is a revised version of the following paper:

— Sarra Habchi, Xavier Blanc, and Romain Rouvoy. On adopting linters to deal with performance concerns in Android apps. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018.

Chapter 6: A Second Thought About Mobile Code Smells: This chapter presents an exploratory study that questions the established definitions and detection techniques for mobile-specific code smells.

1.5 Publications

1.5.1 Published

— Sarra Habchi, Xavier Blanc, and Romain Rouvoy. **On adopting linters to deal with performance concerns in Android apps.** In *ASE18-Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018.

— Sarra Habchi, Naouel Moha, and Romain Rouvoy. **The rise of Android code smells: Who is to blame?** In *MSR2019-Proceedings of the 16th International Conference on Mining Software Repositories*, 2019.

— Sarra Habchi, Romain Rouvoy, and Naouel Moha. **On the survival of Android code smells in the wild.** In *MOBILESoft19-Proceedings of the 6th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2019.

— Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. **Code smells in iOS apps: How do they compare to Android?** In *MOBILESoft17-Proceedings of the 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2017.

1.5.2 Under Evaluation

— Sarra Habchi, Romain Rouvoy, and Naouel Moha. **Android Code Smells: From Birth to Death, and Beyond.** In *Journal of Systems and Software*, 2019.

Chapter 2

State of the Art

In this chapter, we review the works that are closely related to our research topic. While the field of mobile-specific code smells is relatively young—the first publication was in 2013, we found many research works that can be insightful in our context. We synthesized the topics of these research works in the diagram depicted in Figure 2.1. Based on this synthesis, we organize our state of the art in two sections. Section 2.1 presents all topics related to code smells in mobile apps, including the identification, detection, and empirical studies. This section mainly focuses on mobile-specific code smells, but also presents some works about mobile-specific issues that are not explicitly presented as code smells. Moreover, the section covers studies about OO code smells in mobile apps, and other studies of OO code smells that can serve our research purposes. Section 2.2 presents works related to the topic of managing mobile-specific code smells in practice. This includes works about performance management in mobile apps and other software systems. This section also covers the topic of mobile-specific linters and studies about linters in general.

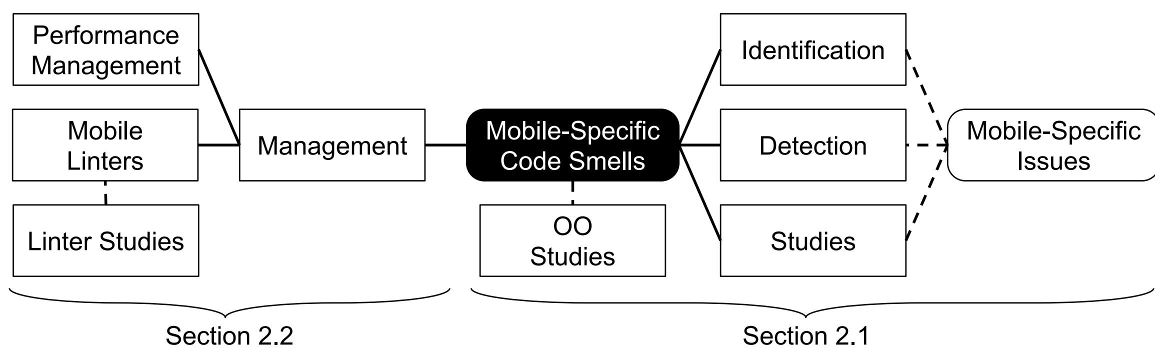

Figure 2.1 Structure of our state of the art.

2.1 Code Smells in Mobile Apps

We present in this section research works related to the identification, detection, and empirical study of mobile-specific code smells. We also briefly present some studies about OO code smells in mobile apps and other software systems.

2.1.1 Identification of Mobile-Specific Code Smells

The works that identified mobile-specific code smells laid the foundation for all other studies about mobile code smells. The main work in this category was conducted by Reimann *et al.* [169], who proposed a catalog of mobile-specific code smells.

The Catalog of Reimann *et al.*

Reimann *et al.* identified 30 quality smells dedicated to Android based on the following information providers:

— **Documentation**: including official ones, like Android Developer Documentation [1] and semi-official ones like Google I/O talks; [2]

— **Blogs**: this includes blogs from developers at Google or other companies who explain development issues with Android;

— **Discussions**: this includes questions and answers forums, like the Stack Exchange network, [3] bug tracker discussions, [4] and other developer discussion forums.

To identify code smells from these resources, Reimann *et al.* opted for a semi-automated approach. First, they automatically crawled and queried (when possible) the information providers and gathered their data into a local database. Afterwards, they filtered this data using keywords like "energy efficiency", "memory", and "performance", paired with issue keywords like "slow", "bad", "leak", and "overhead". The results of this filtering were again persisted in a database. Finally, the authors read the collected resources and extracted a catalog of code smells dedicated to Android.

While the catalog included 30 different quality code smells, many of them were only *"descriptive and abstract"*. That is, Reimann *et al.* only provided a detailed specification for nine code smells. The other 21 were not characterized using a catalog schema and did not have a precise definition. Interestingly, later studies provided detailed definitions for some of these code smells to study them. We will present this in the upcoming sections. Reimann *et al.* used the following schema to present the nine defined code smells:

— Name;

— Context: generally UI, implementation, or network;

1. `https://developer.android.com/`
2. `https://events.google.com/io/`
3. `https://stackexchange.com/`
4. https://source.android.com/setup/contribute/report-bugs

— Affected qualities: the aspect of performance or user experience that may be affected by the quality smell;
— Roles of Interest: the source code entity that hosts the code smell;
— Description;
— Refactorings;
— References;
— Related Quality Smells;

In Table 2.1, we present the nine detailed code smells with their affected qualities.

Code smell	***Affected qualities***
Interruption from Background	User Expectation, User Experience, User Conformance
Data Transmission Without Compression	Energy Efficiency
Dropped Data	User Experience, User Conformity
Durable WakeLock	Energy Efficiency
Internal Use of Getters/Setters	Performance
No Low Memory Resolver	Memory Efficiency, Stability, User Experience
Rigid AlarmManager	Energy Efficiency, Performance
Unclosed Closeable	Memory Efficiency
Untouchable	User Experience, Accessibility, User Expectation

Table 2.1 The code smells detailed by Reimann *et al.* [169].

From Table 2.1, we observe that even though the authors did not explicitly tag their code smells as performance-oriented, most of them affect performance related issues. This is due to the keywords initially used for filtering the data, which were mainly around performance and energy issues.

The Identification of Other Mobile-Specific Issues

Many other works identified mobile-specific practices that hinder performance without referring to them as code smells. In this category, the closest work to our topic is the one of Afjehei *et al.* [4], which identified four performance anti-patterns that are specific to the iOS platform. To identify these anti-patterns, the authors first collected performance-related issue reports from the repositories of four iOS apps written in Swift. The reports were selected based on keywords like “slow”, “performance”, “thread” and “memory”. Then, they manually analyzed 225 reports to identify the root causes of performance issues. They found that the redundant issues are inefficient UI design, memory issues, and inefficient thread handling. Afterwards, they derived from the performance reports four anti-patterns that they presented using the following schema: name, description, example of an instance, example of developer

awareness, and possible solutions. It is worth noting that this study was conducted after our work and publication about iOS-specific code smells.

Pathak *et al.* [159] proposed a taxonomy of mobile errors that cause high energy consumption and identified them as *ebugs*. The ebugs can occur at any level of the mobile system—*i.e.*, apps, operating system, hardware, firmware, external conditions, etc. Pathak *et al.* built their taxonomy using a semi-automated approach. First, they collected 1.2 million posts discussing energy issues from bug repositories and user discussion forums. Then, they filtered and clustered these posts to build an initial classification of ebugs. Finally, they manually analyzed the top groups to build a taxonomy. This taxonomy showed the origins of energy issues faced by users, however it did not characterize the development practices that led to these issues. The only bugs presented in this study that are relevant to our work are *No-Sleep*, *Loop*, and *Immortality*. Interestingly, the *No-Sleep* bug is similar to the code smell *Durable Wakelock* from the catalog of Reimann *et al.* [169].

Liu *et al.* [127] identified three anti-patterns from the performance bugs of Android apps. Their identification started by collecting 70 performance-labeled issues from the bug-tracking systems of 29 Android apps. Afterwards, they manually analyzed the bugs and their related discussions to classify them according to their consequences. They found that performance bugs can be attributed to three groups: GUI lagging, energy leaks, and memory bloat. To identify the patterns that characterize the performance bugs, Liu *et al.* manually analyzed the reports, patches, commit logs, and patch reviews looking for the root causes of the bugs. They successfully identified the root causes of 52 performance bugs and found that for 21 of them there were three common anti-patterns causing them. These anti-patterns are *Lengthy operations in main threads*, *Wasted computation for invisible GUI*, and *Frequently invoked heavy-weight callbacks*. Interestingly, the anti-pattern *Lengthy operations in main threads* is related to the *Heavy* code smells, which are detected by Paprika [106]. Indeed, they all describe heavy operations that block the main thread and degrade the user experience.

Guo *et al.* [92] characterized resource leaks in Android apps. First, they collected data from Android discussion forums to identify three classes of Android resources: exclusive resources, memory-consuming resources, and energy consuming resources. Then, they followed a semi-automated approach where they relied on their tool Relda and on manual analysis to identify resource leaks in 98 Android apps. They found that leaks generally occur for four main reasons: (i) the resource release is in a handler, (ii) the resource release is forgotten, (iii) the resource release is in a callback, or (iv) the Android app lifecycle is misunderstood.

Zhang *et al.* [208] identified four Android development practices that lead to energy leaks. The authors profiled 15 Android apps and detected energy leaks caused by network activity. Afterwards, they manually examined the apps looking for the root causes of these leaks. They found that the root practices are API misuse, bad downloading scheme, repetitive downloads, and aggressive prefetching.

Linarez *et al.* [123] identified API calls and usages that lead to high energy consumption in Android apps. They followed an approach that combined quantitative and qualitative analyses. First, they automatically analyzed the API calls in 55 Android apps to measure their energy consumption values. The measurement highlighted 131 API calls that consumed significantly more energy than the average API calls. Then, they manually analyzed these 131 API calls to identify their patterns. They found that the energy greedy API calls are mostly a manipulation of GUI and images or database calls.

Synthesis

Authors	***Platform***	***Resources***	***Process***	***Focus***	***#Smells***
Reimann *et al.* [169]	Android	documentation blogs discussions	data crawling manual analysis	performance	30
Afjehei *et al.* [4]	iOS	issue reports	manual analysis	performance	4
Pathak *et al.* [159]	Android	issue reports discussions	data crawling manual analysis	performance	3
Liu *et al.* [127]	Android	issue reports	manual analysis	performance	3
Guo *et al.* [92]	Android	source code	profiling manual analysis	performance (resource leaks)	4
Zhang *et al.* [208]	Android	source code	profiling manual analysis	energy leaks	4
Linarez *et al.* [123]	Android	source code	profiling manual analysis	energy efficiency	-

Table 2.2 Identification of mobile-specific issues

We summarize the works that identified mobile-specific issues in Table 2.2. As highlighted in the table, all these works focused on the performance-aspect and some of them were even more specific by investigating narrower issues like resource leaks or energy efficiency. The main difference between these works resides in the resources and processes leveraged in the identification process. The works of Guo *et al.* [92], Zhang *et al.* [208], and Linarez *et al.* [123] relied on the profiling of source code to detect performance bottlenecks and used manual analysis afterwards to identify the root causes of these bottlenecks. The works of Afjehei *et al.* [4] and Liu *et al.* [127] relied only on manual analysis to directly extract bad practices from reports of performance issues. The works that diversified their resources the most are the ones of Reimann *et al.* [169] and Pathak *et al.* [159]. They both collected and filtered large amounts of data from online discussions before manually analyzing it. Interestingly, in addition to the discussions, Reimann *et al.* also collected data from documentation and developer blogs. The work of Reimann *et al.* also stands out with the number of identified issues (30), which tops all other identification studies. Later, this work became a reference in

the field of mobile-specific code smells as many studies rely on it to build detection tools and conduct studies.

2.1.2 Detection of Mobile-Specific Code Smells

In this section, we review the techniques and tools proposed for detecting mobile-specific code smells. In this regard, there are two categories of works: (i) tools and techniques that detect code smells from the catalog of Reimann *et al.* [169] and (ii) tools and techniques that detect other mobile-specific issues.

Tools and Techniques Covering the Catalog of Reimann *et al.*

Refactory: A tool proposed by Reimann *et al.* [169] for the detection and correction of code smells. The tool is implemented as a set of plugins extending the Eclipse platform. It uses an *Eclipse Modeling Framework* (EMF) to detect the nine precisely characterized quality smells from the catalog of Reimann *et al.*. Unfortunately, the source code of Refactory is not currently available and we could not check its performance on Android apps.

Paprika: A tooled approach proposed by Hecht *et al.* [108] to detect OO and Android code smells in Android apps. Paprika takes as input the *Android packages* (APK) and uses Soot [199] and its Dexpler module [34] to convert the Dalvik bytecode into an abstract model of the source code. This model is stored in a graph database, and the code smells are expressed as queries that can be applied into this database to detect code smell instances. Paprika initially detected four Android-specific and three OO code smells. Later, queries were added for new code smells, and it currently detects 13 Android code smells. Paprika is open-source [72] and we could apply it on multiple apps. Hecht *et al.* report that their tool has an F1-score of 0.9 [106].

aDoctor: A tool designed by Palomba *et al.* [156] to detect Android code smells. aDoctor takes the source code as input and relies on the *Eclipse Java Development Toolkit* (JDT) to statically analyze it and detect code smells. aDoctor can detect 15 Android code smells from the catalog of Reimann *et al.* [169]. We recall that the catalog of Reimann *et al.* initially characterized only nine code smells. Hence, Palomba *et al.* had to provide precise definitions for other six code smells to form their set of 15 code smells. It is possible to use aDoctor via both CLI, for large scale analysis, and GUI. The source code of this tool is publicly available, however we could not apply it on Android apps due to several crashes. Nonetheless, Palomba *et al.* report that their tool has a 98% precision and a 98% recall.

MOGP technique: A *Multi-Objective Genetic Programming* technique proposed by Kessentini and Ouni [115] to generate rules that detect Android code smells. MOGP

takes as input examples of Android code smells and uses a multi-objective optimization algorithm to generate a set of rules that maximize both precision and recall. The technique was used to generate rules for six Android code smells from the catalog of Reimann *et al.* [169]. Kessentini and Ouni report that their technique has on average a 81% correctness and a 77% relevance, based on the feedback of 27 developers. MOGP was proposed more as a technique than a tool and its source code is, to the best of our knowledge, not publicly available.

Tools and Techniques Covering Other Mobile-Specific Issues

We present here tools that were proposed for detecting mobile-specific issues—other than the ones present in the catalog of Reimann *et al.* [169]. Most of these tools focus on issues identified by their very same authors and usually in the same studies.

iPerfDetector: A tool proposed by Afjehei *et al.* [4] to detect performance anti-patterns in iOS apps. iPerfDetector takes as input the Swift source code and uses SwiftAST [205] to parse it and generate an AST. Then, using visitors it analyzes the generated tree to detect anti-patterns. iPerfDetector detects four anti-patterns that were identified by the same authors and its source code is open-source.

Relda: A tool proposed by Guo *et al.* [92] to detect resource leaks in Android apps. Relda takes as input the APK and applies reverse engineering on it to extract the Dalvik bytecode. Afterwards, it explores the Dalvik bytecode in sequential order to build a modified *Function Control Graph* (FCG). This modified FCG includes the Android framework callbacks in order to handle the features of event-driven mobile programming. After building the FCG, Relda performs a depth-first search on it to detect unreleased resources. To the best of our knowledge, Relda is not currently open-source.

PerfChecker: A tool proposed by Liu *et al.* [127] for detecting performance anti-patterns in Android apps. PerfChecker takes Java bytecode as input and relies on Soot to perform static analysis on it. The main components of PerfChecker's detection are class hierarchy, call graph, and dependency graph analyses. PerfChecker detects two anti-patterns that are identified by the same authors, namely *Lengthy operations in main threads* and *Frequently invoked heavy-weight callbacks*. Currently, the source code of PerfChecker is not publicly available.

GreenDroid: An approach proposed by Liu *et al.* [126] to detect two energy inefficiencies, namely *Sensor listener misusage* and *Sensory data underutilization*. GreenDroid takes as input the Java bytecode and the configuration files of the Android app (XML files). It uses these files to execute the app on a Java virtual machine. Then, during the execution, it monitors sensor operations—*i.e.*, registration and unregistration, and feeds the app with

mocked sensory data. Afterwards, it tracks the propagation of these data and compares its utilization at different states. This comparison allows GreenDroid to identify app states where sensory data are underutilized, and thus detect *Sensory data underutilization.* The approach detects sensor listeners that have not been unregistered by the end of execution and reports them as instances of *Sensor listener misusage.*

ADEL: For *Automatic Detector of Energy Leaks*, a tool proposed by Zhang *et al.* [208] to help detecting energy leaks in network communications of Android apps. ADEL takes as input Dalvik bytecode and performs dynamic taint-tracking analysis on it to detect energy leaks. During the analysis, the tool labels all the downloaded data objects with a tag to enable their propagation tracking. This tracking follows also all new data objects that are derived from the originally labeled (tainted) ones. Hence, the tool can map the system inputs with the downloaded objects. This mapping can provide developers with insights about unnecessary network communications. Zhang *et al.* claim that these insights can ease the identification of design flaws resulting in energy waste. ADEL was not designed to detect specific energy issues, however, as explained in the previous section, its usage allowed the authors to identify four causes of energy leaks.

Synthesis

	Tool/Technique	***Platform***	***#Issues***	***Analysis***	***Input***	***Source***
Reimann *et al.*	Refactory	Android	9	static	Java code	private
	Paprika	Android	13	static	APK	public
	aDoctor	Android	15	static	Java code	public
	MOGP	Android	6	static	Java code	private
Other issues	iPerfDetector	iOS	4	static	Swift code	public
	Relda	Android	-	static	APK	private
	PerfChecker	Android	2	static	Java bytecode	private
	GreenDroid	Android	2	dynamic	Java bytecode	-
	ADEL	Android	-	dynamic	Dalvik bytecode	-

Table 2.3 Tools and techniques for detecting mobile-specific issues

We summarize in Table 2.3 the tools and techniques that were proposed for detecting mobile-specific issues. The iPerfDetector tool stands out by targeting the iOS platform while all other tools opted for Android. Also, we notice that GreenDroid and ADEL are the only tools that relied on dynamic analysis, while the prevalent choice is static analysis. The use of static analysis is reasonable in this case since mobile-specific issues are usually

expressed as rules describing the source code. It is worth noting that we do not discuss in this synthesis the validation and performance of these tools. Only few studies reported their tool performances and unfortunately they followed different processes (manual validation, developer validation, etc).

For the tools that addressed the catalog of Reimann *et al.* [169], only Paprika and aDoctor are open-source. The two tools also detect more code smells than other tools, 13 and 15 respectively. They differ as Paprika detects code smells directly from the APK, while aDoctor relies on the Java source code.

2.1.3 Studies About Mobile-Specific Code Smells

The studies of mobile-specific code smells fall into two categories: (i) quality studies and (ii) performance studies.

Quality Studies

The closest study to our research topic is the one of Hecht *et al.* [108] in which they used code smells as a metric to track the quality of Android apps. The study analyzed four Android-specific and three OO code smells and covered 3,568 versions of 106 Android apps. The four studied Android-specific code smells included two smells from the catalog of Reimann *et al.* [169], namely *Leaking Inner Class* and *Member Ignoring Method*, and two new code smells called *UI Overdraw* and *Heavy Broadcast Receiver*. The authors described these code smells based on the Android documentation, but did not provide any details about their identification process. The authors relied on Paprika to detect these code smells and their results showed that the quality of Android apps generally follow five patterns: constant decline, constant rise, stability, sudden decline, and sudden rise. Nevertheless, these patterns were built by combining both OO and Android-specific code smells, thus they cannot provide fine-grained insights about the evolution of Android-specific code smells. Interestingly, in the results discussion, the authors suggested that the rise of the quality metric is due to refactoring operations but they did not provide any elements to justify this hypothesis.

Another close study was conducted by Mateus and Martinez [134], who used OO and Android-specific code smells as a metric to study the impact of the Kotlin programming language on app quality. Their study included four OO and six Android-specific code smells and covered 2,167 Android apps. They found that three of the studied OO smells and three of the Android ones affected proportionally more apps with Kotlin code, but the differences were generally small. They also found that the introduction of Kotlin in apps originally written in Java produced an increase in the quality scores in at least 50% of the studied apps.

Another close study is the one of Gjoshevski and Schweighofer [87], which did not study explicitly mobile-specific code smells but relied on Android Lint rules. The study included 140 Lint rules and 30 open-source Android apps. The aim of this study was to identify

the most prevalent issues in Android apps. They found that the most common issues are *Visibility modifier*, *Avoid commented-out lines of code*, and *Magic number*.

Performance Studies

Many studies focused on assessing the performance impact of mobile code smells [40, 107, 157]. For instance, Palomba *et al.* [157] studied the impact of Android-specific code smells on energy consumption. The study is, to the best of our knowledge, the largest in the category of performance assessment. Indeed, it considered nine Android-specific code smells from the catalog of Reimann *et al.* [169] and 60 Android apps. By analyzing the most energy-consuming methods, the authors found that 94% of them were affected by at least one code smell type. The authors also compared smelly methods with refactored ones and showed that methods that represent a co-occurrence of *Internal Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*, consume 87 times more energy than refactored methods.

Hecht *et al.* [107] conducted an empirical study about the individual and combined impact of Android-specific code smells on performance. The study covered three code smells from the catalog of Reimann *et al.* [169]: *Internal Getter Setter*, *Member Ignoring Method*, and *HashMap Usage*. To assess the impact of these code smells, they measured the performance of two apps with and without code smells using the following metrics: frame time, number of delayed frames, memory usage, and number of garbage collection calls. The results of these measurements did not always show an important impact of the studied code smells. The most compelling observations suggested that in the SoundWaves app, refactoring *HashMap Usage* reduced the number of garbage collection calls by 3.6% and the refactoring of *Member Ignoring Method* reduced the number of delayed frames by 12.4%.

Carette *et al.* [40] studied the same code smells as Hecht *et al.*, but focused on their energy impact. They detected code smell instances in five open-source Android apps and then derived four versions of them: three by refactoring each code smell type separately, and one by refactoring all of them at once. Afterwards, they used user-based scenarios to compare the energy impact of the five app versions. Depending on the app, the measurement differences were not always significant. Moreover, in some cases, the refactoring of code smells like *Member Ignoring Method* slightly increased energy consumption (max +0.29%). The biggest observed impact was on the Todo app where refactoring the three code smells reduced the global energy consumption by 4.83%.

Morales *et al.* [140] conducted a study to show the impact of code smells on the energy efficiency of Android apps. The study included five OO and three Android-specific code smells and covered 20 Android apps. The included Android code smells are *Internal Getter Setter* and *HashMap Usage*, from the catalog of Reimann *et al.* [169], and *Binding Resources Too Early*. The results of their analysis of Android code smells showed that refactoring *Internal Getter Setter* and *Binding Resources Too Early* can reduce energy consumption in some

cases. As for OO code smells, they found that two of them had a negative impact on energy efficiency, while the others did not. Also, the authors reported that code smell refactoring did not increase energy consumption. Morales *et al.* also proposed EARMO, for *Energy-Aware Refactoring approach for MObile apps*. EARMO is a multi-objective approach that refactors code smells in mobile apps while considering their impact on energy consumption.

Synthesis

We summarize in Table 2.4 the main research works that studied mobile-specific code smells. We observe that most of these studies focused on assessing the impact of code smells on performance aspects like memory, UI, and energy. From all these studies, the one of Palomba *et al.* [157] stands out because it covers more apps and code smells and consequently has more generalizable results. The other studies that did not focus on performance, namely Hecht *et al.* [108], Mateus and Martinez [134], and Gjoshevski and Schweighofer [87], used mobile-specific code smells as a metric to asses app quality and did not effectively study them. Specifically, they did not investigate the grounds of these code smells and did not provide any qualitative insights about them.

Authors	***Study focus***	***#Apps***	***#Smells***
Hecht *et al.* [108]	quality evolution	106	4
Mateus and Martinez [134]	Kotlin quality	2167	6
Gjoshevski and Schweighofer [87]	technical debt	30	140
Palomba *et al.* [157]	energy efficiency	60	9
Hecht *et al.* [107]	performance (memory & UI)	2	3
Carette *et al.* [40]	energy efficiency	5	3
Morales *et al.* [140]	energy efficiency	20	3

Table 2.4 Studies about mobile-specific code smells.

2.1.4 Studies About OO Code Smells

In this section, we present a few studies that did not address mobile-specific code smells but investigated a close concern. This includes works on (i) OO code smells in mobile apps and also (ii) studies of OO code smells in general.

OO Code Smells in Mobile Apps

Mannan *et al.* [129] compared the presence of well-known OO code smells in 500 Android apps and 750 desktop apps written in Java. They did not observe significant differences

between these two types of systems in terms of density of code smells. However, they observed that the distribution of code smells for Android is more diversified than desktop applications. Their study also showed that *Internal and external duplications* are the most common code smells in desktop applications.

Linares-Vásquez *et al.* [124] used DECOR [139] to detect OO code smells in mobile apps built using *Java Mobile Edition* (J2ME) [152]. The study included 1,343 apps and 18 different OO code smells. Their results showed that the presence of code smells negatively impacts software quality metrics, especially those related to fault-proneness. They also found that some code smells are more common in certain categories of Java mobile apps.

Verloop [200] used popular Java refactoring tools, such as PMD [164] and JDEODORANT [194] to detect code smells, like *large class* and *long method* in open-source Android apps. They found that code smells tend to appear at different frequencies in core classes—*i.e.,* classes that inherit from the Android framework—compared to non-core classes. For example, *long method* was detected twice more in core classes than in non-core ones.

OO Software

In our review of OO code smells, we focused on studies that attempted to understand code smells and their motives. In this regard, the closest study was conducted by Palomba *et al.* [155] who inspected developers' perception of OO code smells. The study included instances of 12 different code smells, three open-source projects, and 34 developers. First, they manually identified smelly and non-smelly code snippets and presented them to the participants. Afterwards, they surveyed the participants to check if they perceived any problems in the code snippets. The results of this survey showed that code smells that are related to complex or long source code—*e.g.*, *Blob*, *Long Method*, and *Complex Class*, are generally perceived as an important threat by developers. On the other hand, code smells that are related to OO practices, *e.g.*, coupling, rather than explicit complexity are less perceived by developers.

Other studies aiming to understand code smells opted for analyzing the evolution of code smells through the change history. For instance, Tufano *et al.* [197] analyzed the change history of 200 open-source projects to understand when and why code smells are introduced and for how long they survive. They observed that most of code smell instances are introduced when files are created and not due to evolution process. They also found that new features and enhancement activities are responsible for most smell introductions, and newcomers are not necessarily more prone to introducing new smells. Moreover, their manual analysis of code smell removals showed that only 9% of code smells are removed with specific refactoring operations.

Peters and Zaidman [161] conducted a case study on seven open-source systems to investigate the lifespan of code smells and the refactoring behavior of developers. They found

that, on average, code smell instances have a lifespan of approximately 50% of the examined revisions. Moreover they noticed that usually one or two developers refactor more than the others, however the difference is not large. Finally, they observed that the main refactoring rationales are cleaning up dead or obsolete code, dedicated refactoring, and maintenance activities.

2.1.5 Summary

We reviewed in this section research works related to the topic of code smells in mobile apps. This review showed that many studies attempted to identify performance issues that are specific to mobile platforms. For our topic, the most relevant study is the one of Reimann *et al.* [169] that proposed a catalog of 30 Android-specific code smells. Many studies relied on this catalog to propose detection tools and conduct empirical studies. The tools that stand out are aDoctor and Paprika since they are open-source and detect a large set of code smells. As for the studies, our review showed that they mainly focused on assessing the performance impact of mobile-specific code smells. Moreover, studies that did not cover performance used mobile-specific code smells as a metric to measure the quality of mobile apps and did not study them specifically. By comparing the scope of these studies to the existing works about OO code smells, we claim that we need more empirical studies to enable understanding mobile-specific code smells.

After our analysis of studies about mobile-specific code smells, we review in the upcoming section another facet of our topic, which is the management of mobile-specific code smells in practice.

2.2 Management in Practice

As there are no studies about the management of mobile-specific code smells in practice, we present in this section studies that addressed similar concerns. In this regard, the closest studies are the ones investigating performance management in mobile apps and software systems in general. Another relevant topic for this section is mobile-specific linters, which represent the equivalents of code smell detection tools in practice. There are no studies about the usage of linters for managing mobile-specific code smells or performance in general. Therefore, we present other studies that provide qualitative insights about linters and can be useful for our research.

2.2.1 Performance Management

There are two main works about performance management in mobile apps. First, the study of Linarez *et al.* [125] that surveyed developers to identify the common practices and tools for detecting and fixing performance issues in open-source Android apps. Based on 485 answers,

they deducted that most of developers rely on reviews and manual testing for detecting performance bottlenecks. When asked about tools, developers reported using profilers and framework tools and only five of them mentioned using static analyzers. Developers were also openly questioned about the targets of their performance improvement practices. From 72 answers, the study established the following categories: GUI lagging, memory bloats, energy leaks, general performance, and unclear benefits.

Another study was conducted by Liu *et al.* [127] who investigated the management of performance bugs in mobile apps. Specifically, they analyzed the issue reports and source code of 29 Android apps to inspect, among other questions, the manifestation of performance bugs and the efforts needed to fix them. They found that, contrarily to desktop apps, small scale data is enough to manifest performance bugs in mobile apps. However, they observed that more than one third of these performance bugs required user interaction to manifest. We consider that this observation is reasonable since mobile apps are known to be very event-driven. The study results also showed that some performance bugs require specific software or hardware platforms to manifest. As for the efforts, the study suggested that fixing performance bugs is more difficult as it requires more time and discussions and larger patches than fixing non-performance bugs. By manually analyzing the discussions and patches, the authors showed that during the debugging process, information provided by stacktrace was less helpful than information provided by profilers and performance measurement tools. Nonetheless, the study reports that these tools still need enhancement to visualize simplified runtime profiles.

Performance management has also been addressed in the pre-mobile era through many studies. Notably, Zaman *et al.* [207] manually analyzed a random sample of 400 performance and non-performance bug reports from Mozilla Firefox and Google Chrome. The objective was to understand the practices and shortcomings of reproducing, tracking, and fixing performance bugs. Among other interesting findings, they realized that fixing performance bugs is a more collaborative task than for non-performance bugs. However, performance bugs show more regression proneness and they are not well tracked.

2.2.2 Linters

As Android and iOS are the most used mobile platforms, the available linters are mainly focused on these two platforms.

Android: ANDROID LINT [9] is the mainstream linter for Android, and the only one that detects Android-specific issues. It is integrated in Android Studio, the official IDE for Android. It can be run on Android projects from the command line or in Android Studio interactively. It scans the code to identify structural code problems that can affect the quality and performance of Android apps. Lint targets 339 issues related to correctness, security,

performance, usability, accessibility, and internationalization. The category performance includes 36 checks,[5] *a.k.a.* rules. As an example, we explain the rule *HandlerLeak* that checks if a `Handler` is used as a non-static inner class or not. This situation is problematic because the `Handler` holds a reference to the outer class. Thus, as long as the `Handler` is alive the outer class cannot be garbage collected, thus causing memory leaks. A similar issue has been addressed in research studies as a code smell named *Leaking Inner Class* [108]. Android Lint reports each problem with a brief description message, a priority, and a severity level. The priority is a number from 1 to 10, and the severity has three levels: `ignore`, `warning`, and `error`. All the Android Lint checks have a default priority and severity. The severity is a factor that can be configured by developers to classify the problems on which they want to focus. Lint also offers the possibility of adding new rules, however the addition of rules requires the user to develop the detection algorithms in Java.

Other linters like PMD [164], CheckStyle [45], Infer [112], and FindBugs [66] can be used to analyze Android apps. Nonetheless, they only detect issues related to either Java or Kotlin and they do not consider issues or practices specific to the Android SDK. For instance, PMD detects programming flaws like dead code. CheckStyle checks that coding conventions are respected and FindBugs analyzes the Java bytecode to detect potential bugs.

For the programming language Kotlin, Detekt [28] is a linter that can be used to compute source code complexity and identify some code smells, while Ktlint [175] is a linter that focuses on checking code conventions.

iOS: With regard to iOS apps, there are only few tools that support the analysis of code quality and to the best of our knowledge, none of them supports iOS-specific code smells. For instance, OCLint [147] is a static analyzer that inspects C, C++ and Objective-C code, searching for code smells and possible bugs. The detected issues are relevant for the programming language and are not dedicated to the iOS platform. Another tool is Infer [112], which uses static analysis to detect potential bugs in Objective-C, C, and Android programs. Infer can detect possible memory leaks in iOS and *null pointer* exceptions and resource leaks in Android. Other tools, like Clang [49], SonarQube [177] and Faux Pas [65] are also static analyzers that can be used to detect potential bugs in Objective-C, but they do not support iOS-specific smells.

As for the programming language Swift, the main code analyzers are Tailor [188], Swift Lint [167] and Linter Swiftc [30]. Tailor is a static analyzer that checks styling consistency and helps avoiding bugs. Swift Lint enforces Swift style and conventions, and Linter Swiftc is a Linter plugin for syntax analysis.

5. As for September 2019.

2.2.3 Linter Studies

The objective of this short section is not to provide an extensive literature review of linter studies. Our aim is to rather highlight qualitative studies that investigated developers' usage of linters. These studies can be an inspiration for our inspection of the adequacy of linters for managing mobile-specific code smells.

The first study that drew our attention is the one of Tómasdóttir *et al.* [192] who investigated the benefits of using linters in a dynamic programming language. They interviewed 15 developers to understand why and how JavaScript developers use ESLint [64] in open-source software. They found that linters can be used to (i) augment test suites, (ii) spare newcomers' feelings when making their first contributions, and (iii) save time that goes into discussing code styling.

Christakis and Bird [47] conducted an empirical study combining interviews and surveys to investigate the needs of developers from static analysis. Among other results, they found that performance issues are the second most severe code issues that require an immediate intervention from developers. Performance issues were also in the top four needs of developers.

Johnson *et al.* [114] conducted 20 interviews to understand why developers do not use static analysis tools, like FindBugs [66], to detect bugs. They found that all the participants are convinced by the benefits of static analysis tools. However, false positives and warning presentation are the main barriers to developers in adopting static analyzers.

2.2.4 Summary

In this section, we reviewed performance management studies and tools used for code smell detection in practice. This review showed that many linters are available for mobile developers in both platforms Android and iOS. These tools can be considered as the equivalents to detectors of mobile-specific code smells like aDoctor and Paprika. Indeed, some of these linters, *e.g.*, Android Lint, can detect performance issues that are similar to the Android-specific code smells studied by researchers. Our review also covered studies about the usage of tools for mobile development. Moreover, as there are no studies about the management of mobile-specific code smells in practice, we showed different empirical studies that can be a source of inspiration for any qualitative inspection.

2.3 Conclusion

Based on our literature review, we observe that we still lack knowledge about various aspects of mobile-specific code smells:

— **Other mobile platforms:** Most of the reviewed research works only studied the Android platform. It is important to explore other mobile platforms and study their

specific code smells to enable an effective discussion of generalized mobile-specific code smells.

— **Motives:** The reviewed studies did not explain the reasons behind the accrual of mobile-specific code smells. Some studies hypothesized that it is the result of releasing pressure and developers' incompetence [106, 108], however they did not provide elements to support such hypothesis. It is necessary to analyze the history of code smells to investigate the factors that impact their introduction and removal, and thus understand their motives;

— **Developers' perception:** Mobile-specific code smells are substantially different from classical OO code smells. They are performance-oriented and more related to the details of their development frameworks. Consequently, developers' perception of these smells can be different from what have been discussed in previous OO studies. Thus, we need studies to capture developers' knowledge and consideration of mobile-specific code smells;

— **Adequate tools for managing mobile-specific code smells:** Our tool review showed that many static analyzers are proposed to developers that aim to preserve the performance of their mobile apps. However, the studies of performance management in practice showed that mobile developers rely substantially on profilers and use rarely static analyzers for performance concerns [125]. This observation highlights a gap between the tools proposed by the research community for handling performance concerns, which are mainly static analyzers, and the tools used in practice, which are mainly profilers. This gap incites us to question the adequacy of static analyzers for managing performance-related issues. Indeed, this adequacy and the management of mobile-specific code smells in general, have never been investigated by research works.

In our thesis, we address these knowledge lacks and provide necessary elements to understand mobile-specific code smells. We start by addressing the lack of studies about mobile platforms. We present in the upcoming chapter an empirical study that explores mobile-specific code smells in the iOS platform.

Chapter 3

Code Smells in iOS Apps

As demonstrated in the state of the art, the research community focused mainly on Android in its study of code smells in mobile apps. This focus may be justified as (i) Android is the most used mobile operating system, (ii) more Android apps are available in open-source platforms, and (iii) Android apps are relatively easy to analyze (compared to iOS). Nonetheless, the study of other mobile platforms is necessary as the latter have different development frameworks and runtime environments. These differences may impact the nature of code smells and also their prevalence in the source code. Hence, comparative studies between different mobile platforms can be helpful in understanding mobile code smells and their motives. It is also important to look for common code smells between different mobile platforms to help generalizing future studies and tools.

For these reasons, we explore in this chapter code smells in the iOS platform. In particular, we identify six new iOS-specific code smells from developers discussions and the platform official documentation. We summarize these code smells in a catalog similar to the one of Reimann *et al.* [169] and we emphasize the similarities between our iOS code smells and other Android-specific smells. Afterwards, we propose SUMAC, a tool that detects code smells from our iOS-catalog and other OO catalogs. SUMAC is open-source [97] and it supports iOS apps written in Objective-C and Swift.

Using SUMAC and our catalog of iOS-specific code smells, we conduct an empirical study with the aim of answering the following research questions:

— **RQ1:** Are OO and iOS smells present in Swift and Objective-C with the same proportion?

— **RQ2:** Are code smells present in iOS and Android with the same proportion?

In this study, we first assess the extent of code smells in iOS apps and investigate the impact of programming languages on it. Then, we conduct a comparative analysis to highlight the difference between iOS and Android in terms of code smells. Our study covers 103 Objective-C, 176 Swift, and 1,551 Android apps.

Our study results show that, despite the programming language differences, code smells tend to appear with the same proportions in iOS apps written in Objective-C and Swift. Our results also show that for all code smells, at the exception of Swiss Army Knife, Android apps tend to have significantly more code smells.

This chapter is organized as follows. Section 3.1 presents concepts related to the analysis of iOS source code and binaries. Then, we introduce, in Section 3.2 our catalog of iOS-specific smells, and in Section 3.3 our novel tool SUMAC. Section 3.4 reports the results of our empirical investigation for RQ1 and Section 3.5 presents our study for RQ2. Finally, Section 3.6 summarizes our work and outlines further works to perform on this research topic.

This chapter is a revised version of a paper published in the proceedings of MOBILE-Soft'17 [100].

3.1 Background on iOS Apps Analysis

This section reports on the ongoing state of the art in terms of static and dynamic analyses for iOS apps.

3.1.1 Dynamic Analysis

Most of the existing dynamic analysis techniques work on applications and systems running on x86 architectures. While this architecture is widely used for desktop operating systems, mobile software systems rather use the ARM architecture, which is different [187]. Consequently, the existing dynamic analysis techniques cannot be used for analyzing iOS apps. Another constraint for the dynamic analysis of iOS applications is the general adoption of event-based graphic user interfaces, which makes app features depend on the events triggered by the user. Studies showed that because of these constraints, a finite number of automatic runs of an iOS app may be unable to cover all the execution paths [187]. Therefore, a relevant dynamic analysis of iOS apps necessarily requires an interaction with the graphic interface.

3.1.2 Static Analysis of Binaries

Although Objective-C is a strict super set of the C language, its binaries differ from the C/C++ ones by the use of messages to support the interactions between objects. All these messages are routed by the routine `objc_msgSend`. This means that every method call from an object to another goes through a call to the method `objc_msgSend` [63]. The routing and the additional calls impact significantly the semantics of control flow graphs, and thus the results of the static analysis. Hence, resolving the real method calls in this case requires manipulations at the level of the CPU registers and type information tracking [63].

3.1.3 Static Analysis of Source Code

iOS applications are made available in the App Store as IPA files (iOS App Store Package). With the IPA extension, the code source files are encrypted with FairPlay, a technique for numerical rights management adopted by Apple, and compressed with the ZIP format [149]. As a result, the access to the source code through these files requires decryption and reverse-engineering, which is prohibited. This implies that source code analysis can only be applied on open-source apps.

Synthesis: Regarding the current state of practices, constraints, and tools available in the iOS ecosystem, we consider that the most relevant solution to study code smells in iOS consists in analyzing the source code of apps published as open-source software.

3.2 Code Smells in iOS Apps

This section introduces our first contribution: a catalog of iOS-specific code smells. We first explain the process we followed to identify these code smells, then we describe the identified smells as a catalog, and finally we highlight the similarities between iOS and Android code smells.

3.2.1 Code Smells Identification Process

As code smells in iOS have not been addressed by the state of the art, we relied on the platform official documentation and the community's knowledge to identify these smells. The sources we considered relevant for our task are:
— The Apple developer guide [25],
— Well-known web references for iOS development: RayWenderlich [166] and Objc.io [146].

Using a grounded theory approach [185], we manually parsed the posts of these sources and identified the problems raised by developers, as well as the critically recommended practices. Thereafter, we selected the most relevant bad practices and detailed them from data of:
— Developers blogs,
— Q&A forums like Stack Overflow [179].

Lastly, we structured the problems using the mini-antipattern template proposed by Brown *et al.* [38].

3.2.2 Catalog of iOS Code Smells

Name: *Singleton Abuse (SA)*

Category: `Design`
Problem: Singleton is one of the design patterns recommended by Apple for the development of iOS apps [23]. Developers interact often with this pattern through platform classes, like `UIApplication` and `NSFileManager`. Moreover, the XCode IDE has a default code to easily generate a Singleton instance. However, this tends to encourage the use of this pattern in inappropriate situations.
Example: A recurrent example of abusive use of Singleton is when the singleton class is used for storing global variables of the application—*e.g.*, user data that are accessed by all the app classes. Having a global data in the application makes it stateful, and thus difficult to understand and debug [183].
References: [5, 53, 110, 122, 183]

Name: *Massive View Controller (MaVC)*
Category: `Design`
Problem: Most of the iOS apps are designed using MVC, a design/architectural pattern that splits the application into three layers: *model*, *view*, and *controller*, and where the default role of the controller is to link the two other layers [135]. However, in iOS, app controllers tend to carry much more responsibilities than connecting the model to the views [135]. In particular, the controller handles UI events, like clicks and swipes, because it is a part of the response chain. It also receives the system warnings regarding the memory state and it has to perform the necessary operations for managing the memory occupied by the app. All these additional responsibilities make the controllers massive, complex, and difficult to maintain [113].
References: [29, 46, 133, 135, 178, 165]

Name: *Heavy Enter-Background Tasks (HEBT)*
Category: `Performance`
Problem: Since version 4.0 of iOS, apps are allowed to execute tasks in the background. This possibility requires from apps specific adjustments, like freeing the memory space or stopping some tasks, to manage the transition from front to background. These adjustments must be applied to the method `applicationDidEnterBackground:` from the `AppDelegate` class, which is called when the app moves to the background [24]. However, a problem occurs when the operations of this method last for a long time, thus exhausting the allocated execution time, and causing the completion handler to be called in order to suspend the app.
References: [24]

Name: *Ignoring Low-Memory Warning (ILMW)*
Category: `Performance`
Problem: In iOS, when the system requires more memory to perform its tasks, it sends

low-memory warnings to the apps holding important memory spaces via the method `didReceiveMemoryWarning:` of the `UIViewController` class. Every view controller should implement this method to free the unused memory space—*e.g.*, view elements that are not currently visible. However, when the method is not implemented, the view controller cannot react to the system warnings, and if the application is holding an important memory space, it will be killed by the system [21].
References: [21, 130]

Name: *Blocking The Main Thread (BTMT)*
Category: `Performance`
Problem: In iOS, the UIKit is directly tied to the main thread, so all the graphical user interface interactions are in this thread and are impacted by the execution time of the other operations sharing it [22]. Therefore, every heavy processing in the main thread makes the UI completely unresponsive. The operations that often block the main thread are [131]:

- The synchronous access to the network,
- The access to the disk, especially for reading or writing large files,
- The execution of long tasks, like animations or complex data processing.

References: [22, 131]

Name: *Download Abuse (DA)*
Category: `Performance`
Problem: Mobile apps rely increasingly on online data storage to save their content without impacting the user's limited internal storage. Despite the positive impact of this practice on user experience, it may also lead to performance problems if used inappropriately. The *Download Abuse* code smell describes the case where the online data are downloaded with abuse, the most explicit case of this may be downloading the same data repeatedly without using the cache memory. Since the access to online data requires more energy than the access to internal disk [41], this practice negatively impacts the battery autonomy. Furthermore, depending on the network state it may also impact the execution time and it can be costly if the user is connected to a GSM network.
References: [22, 91, 132]

3.2.3 Similarities with Android Code Smells

After building our catalog, we noticed that some of the identified iOS-specific code smells are similar to Android code smells studied in previous works [91, 106, 169]. We present in this section the similar Android code smells and their commonalities with our catalog.

No Low Memory Resolver (NLMR): When the Android system is running low on memory, the system calls the method `onLowMemory()` of every running activity. This method is responsible of trimming the memory usage of the activity. If this method is not implemented by the activity, the Android system automatically kills the process of the activity to free memory, which may lead to an unexpected program termination [106, 169]. This smell is analogous to *Ignoring Low Memory Warning* in iOS.

Heavy Main Thread Method (HEAVY): This code smell is a composition of three similar Android smells: *Heavy Service Start*, *Heavy BroadcastReceiver*, and *Heavy AsyncTask* [106, 108, 169]. The three code smells are defined as Android methods that contain heavy processing and are tied to the main-thread. Consequently, the three code smells lead to freezing the UI and make the app unresponsive. The main concept of the three code smells is analogous to *Blocking the main-thread* in iOS.

3.3 Sumac

To study code smells in iOS apps, we built Sumac, an open-source tool that analyzes iOS apps and detects OO and iOS-specific code smells [97]. Sumac extends the existing tool Paprika, which so far only supports Android apps written with the Java programming language. We devote the following subsections to explain the changes thet we applied to Paprika in order to build Sumac and support the iOS framework.

3.3.1 Code Analysis

From Parsing Android Apps

Paprika uses the Soot framework [199] and its Dexpler module [34] to analyze APK artifacts. Soot converts the Dalvik bytecode of Android apps into a Soot internal representation, which is similar to the Java language, and also generates the call graph of the app. The representation and the graph are used by Paprika to build a model of the code (including classes, methods, attributes) as a graph annotated with a set of raw quality metrics. Paprika enriches this model with metadata extracted from the APK file (*e.g.*, app name, package) and the Google Play Store (*e.g.*, rating, number of downloads).

To Parsing iOS Apps

As Soot does not support Objective-C nor Swift, we had to find an alternative solution. The analysis of iOS binaries is challenging due to the issues exposed in Section 3.1, thus we opted for source code analysis. We built two parsers for Objective-C and Swift using ANTLR4 grammars [189, 190] and the ANTLR parser generator. Reading the source code, the parsers build an *Abstract Syntax Tree* (AST) of the analyzed program. We built a custom visitor for each parser to extract from the AST all the necessary attributes and metrics to feed Sumac and to build the associated graph model. Since we are retrieving apps from open-source repositories (*e.g.*, GitHub) instead of the official online stores, we cannot retrieve the complementary metadata, but this does not impact the detection process and the results we obtain.

3.3.2 Sumac Model for iOS

The model built from the code analysis phase is converted into a graph where the nodes are entities, the edges are relationships, and the nodes attributes are properties or quality metrics. The graph model was originally designed for Paprika to reflect the core components of any Android app, and consequently it is based on the Java language elements. While Objective-C and Swift share the same core concepts of Java (classes, methods and attributes), they have also features of the procedural programming like functions and global variables. Moreover, they introduce new concepts like extensions, structs, and protocols. To cope with these particularities, we included in our Sumac model the new concepts of Objective-C and Swift and we removed the ones that are only relevant for Java. The exhaustive list of entities and metrics used for iOS is provided in Appendix A.

3.3.3 Storing the iOS Model

We store the Sumac model generated from the app analysis in a Neo4j database [142]. Neo4j is a flexible graph database that offers good performances on large-scale datasets especially when combined with the Cypher [141] query language. We therefore reuse the storage layer of Paprika as it is scalable and efficient and allows to easily store and query the graph model.

3.3.4 Code Smell Queries

We use Cypher [141] queries for expressing the code smells. When applied on the graph database, these queries detect code smell instances. We kept the Paprika original queries for the OO smells and we only updated their metrics thresholds. The thresholds are computed using the Boxplot technique [198] by considering all the dataset apps and thus, we have different thresholds in Objective-C and Swift.

As an example, we report below the Cypher query of the OO code smell *Complex Class* (CC):

Listing 3.1 *Complex Class* detection query using Cypher.

```
MATCH (cl:Class)
WHERE
  AND cl.class_complexity > very_high_complexity
RETURN cl
```

This query detects all classes with a complexity higher than the *very high* threshold of complexity.

For iOS-specific smells, we followed the same process defined by Paprika for transforming the literal definition of the code smell into a query. The following two examples illustrate the queries for detecting *ILMW* and *MaVC*.

Listing 3.2 *Ignoring Low-Memory Warning* query.

```
MATCH (cl:Class)
WHERE HAS(cl.is_view_controller)
  AND NOT (cl:Class)-[:CLASS_OWNS_METHOD]->
    (:Method{name:'didReceiveMemoryWarning'})
RETURN cl
```

The ILMW query looks for the view controllers missing an implementation of the method `didReceiveMemoryWarning`, which is required to react to the system memory warnings.

Listing 3.3 *Massive View Controller* query.

```
MATCH (cl:Class)
WHERE HAS(cl.is_view_controller)
  AND cl.number_of_methods > very_high_nom
  AND cl.number_of_attributes > very_high_noa
  AND cl.number_of_lines > very_high_nol
RETURN cl
```

For the *Massive View Controller* query, classes are identified as smell instances whenever the metrics number_of_methods (`nom`), numbers_of_attributes (`noa`), and number_of_lines (`nol`) are *very high*.

It is worth noting that translating code smell definitions into queries is not always possible. This can be due to the complexity of the code smell itself, as for *Singleton Abuse*, which is contextual and requires some intelligence to spot it. Another obstacle for translating the code smells is the limits of static analysis. For example, dynamic analysis is required to measure the execution time to detect the *Heavy Enter Background Task* code smell. Also, data flow analysis is needed to determine whether the downloaded data is being reused or

not and detect the code smell *Download Abuse*. Therefore, we cannot include the code smells *Singleton Abuse*, *Heavy Enter Background Task*, and *Download Abuse* in our tool SUMAC.

Handling the Uncertainty

Many code smell detection tools rely on standard thresholds to identify outliers. However, these approaches are limited as they can only report boolean values. In order to deliver results that are closer to human reasoning, we adopt fuzzy logic [206]. In particular, SUMAC uses jFuzzyLogic [48] to compute the fuzzy value between the two extreme cases of truth (0 and 1). For each metric, the *very high* value is considered as an extreme case of truth. These fuzzy values represent the degree of truth or certainty of the detected code smell instance.

3.4 Study of Code Smells in iOS

This section focuses on our first research question:

RQ1: Are OO and iOS smells present in Swift and Objective-C with the same proportions?

Using SUMAC, we study the presence of OO and iOS code smells in a dataset of Objective-C and Swift apps. The following subsections present the design of this study, its results discussion, and its threats to validity.

3.4.1 Objects

The objects of our study are the iOS and OO smells detectable by SUMAC. As mentioned previously, SUMAC can detect three iOS smells, namely *Massive View Controller*, *Ignoring Low Memory Warning*, and *Blocking the Main Thread*. In addition to these iOS smells, SUMAC can also detect four well-known OO smells, *Blob*, *Swiss Army Knife*, *Long Method*, and *Complex Class* [38, 69]. To clarify all the study objects, we present here a brief description of these OO code smells. More detailed definitions can be found in previous works [38, 69].

Blob Class (BLOB): A Blob class, also known as *God class*, is a class with a large number of attributes and/or methods [38]. The Blob class handles a lot of responsibilities compared to other classes that only hold data or execute simple processes.

Swiss Army Knife (SAK): A *Swiss Army Knife* is a very complex class interface containing a lot of methods. This interface is often designed to handle a wide diversity of abstractions or to meet all possible needs in a single interface [38].

Long Method (LM): *Long Methods* are implemented with much more lines of code than the average. They are often very complex, and thus hard to understand and maintain. These methods can usually be split into smaller methods to fix the problem [69].

Complex Class (CC): A *Complex Class* is a class containing complex methods. Again, these classes are hard to understand and maintain and need to be refactored [69]. The class complexity can be estimated by summing the internal complexity of each of its methods.

3.4.2 iOS Dataset and Inclusion Criteria

Following the technical choices we adopted for SUMAC, our code smell detection requires the source code of the app. Thus, our dataset is exclusively composed of open-source apps. We chose to consider a collaborative list of apps available from GitHub [57] that is, as far as we know, the largest repository of open-source iOS apps. It gathers currently 605 apps of diverse categories and authored by different developers. Almost 47% of these apps are published in the App Store.

The repository apps are written with Objective-C, Swift, Xamarin, and other web languages, such as HTML and JavaScript. We considered only the native apps written with Objective-C and Swift. We also excluded demo and tutorial apps since they are very light and not relevant for our study. As a result, we included 103 Objective-C apps and 176 Swift apps within this study. The exhaustive list of apps used in our study is available online [1].

3.4.3 Hypotheses and Variables

Independent variable: The independent variable of this study is the programming language of the app: Objective-C (*ObjC*) or *Swift*.

Dependent variables: The dependent variables are the proportions of code smells in the analyzed apps.

Hypothesis: To compare the presence of the different code smells ($CS \in \{Blob, LM, CC, SAK, MaVC, ILMW, BTMT\}$) in the two languages ($L \in \{ObjC, Swift\}$), we formulate the following null hypothesis:

HR^{CS}: There is no difference between the *proportions* of code smells for the apps written with Objective-C or Swift.

3.4.4 Analysis Method

First, we analyze the Objective-C and Swift datasets with SUMAC to detect the OO and iOS code smells. Then, we compute, for each app, the ratio between the number of code smells and the number of concerned entities—*e.g.* the ratio of ILMW is normalized with the number of view controllers. Afterwards, we compute the median and the interquartile range of the ratios. We also compute Cliff's δ [170] effect size to quantify the importance of the difference in proportions of the smells.

1. `http://sofa.uqam.ca/paprika/paprika_ios.php`

Cliff's δ is reported to be more robust and reliable than Cohen's d [51]. It is a non-parametric effect sizes measure—*i.e.*, it makes no assumptions of a particular distribution—which represents the degree of overlap between two sample distributions [170]. It ranges from -1 (if all the selected values in the first group are larger than the ones of the second group) to $+1$ (if all the selected values in the first group are smaller than the second group). It evaluates to zero when two sample distributions are identical [50]:

$$\text{Cliff's } \delta = \begin{cases} +1, & \text{Group 1} > \text{Group 2;} \\ -1, & \text{Group 1} < \text{Group 2;} \\ 0, & \text{Group 1} = \text{Group 2.} \end{cases}$$

Cohen's d is mapped to Cliff's δ via the percentage of non-overlap, as shown in Table 3.1 [170]. Cohen [52] states that a medium effect size represents a difference likely to be visible to a careful observer, while a large effect is noticeably larger than medium.

Cohen's Standard	Cohen's d	% of non-overlap	Cliff's δ
small	0.20	14.7 %	0.147
medium	0.50	33.0 %	0.330
large	0.80	47.4 %	0.474

Table 3.1 Mapping Cohen's d to Cliff's δ.

We chose to use the median and the interquartile range because our variables are not normally distributed. Likewise, we opted for the Cliff's δ test since it is suitable for non-normal distributions. Moreover, this test is recommended for comparing samples of different sizes [128].

3.4.5 Results

This section reports and discusses the results we obtained to answer our first research question.

Overview of the Results

Table 3.2 summarizes the percentages of apps affected by each code smell for the two datasets. For the iOS-specific code smells, we observe that *ILMW* affects most of the studies apps (78 % and 85 %). *MaVC* is also relatively common, as it appears in more than 10 % of the apps in the two datasets. *BTMT* is the least recurrent code smell. It appears in only 6 % of Objective-C apps, and it does not appear at all in Swift ones. Generally, we observe that iOS code smells tend to be more present in the Objective-C apps than the Swift ones. However, this insight needs to be consolidated with statistical tests.

	Lang	***BLOB***	***LM***	***CC***	***SAK***	***MaVC***	***ILMW***	***BTMT***
Apps %	*ObjC*	58.82	96.08	76.47	24.51	33.33	85.29	6.86
	Swift	40.34	87.50	69.32	14.77	10.23	78.41	0.00

Table 3.2 Percentage of apps affected by code smells.

Regarding the OO code smells, *LM* is the most recurrent one. It appears in approximately 90% of the apps in the two datasets. *BLOB* and *CC* are also very common with percentages ranging from 40% to 76%. *SAK* is a less prevalent code smell, but its percentage is still significant (14% and 24%). Similarly to iOS-specific code smells, the table shows that Objective-C apps are more prone to OO code smells than Swift apps.

Discussion: The surprisingly high prevalence of *ILMW* shows that the method `applicationDidReceiveMemoryWarning:` is not implemented in most of the view controllers. Missing the implementation of this method can be justified when absolutely no resource can be freed by the app. However, it is unlikely that such a large proportion of view controllers belongs to this case. We therefore hypothesize that developers are not aware of the benefits of implementing this method.

Comparison between Objective-C and Swift

Table 3.3 shows, for each code smell type and programming language, the median (med) and interquartile range (IQR) of the code smell ratios. Also, it reports on the Cliff's δ effect size between the ratios in the two datasets.

For each application a, the ratio of a code smell type s is defined by:

$$ratio_s(a) = \frac{fuzzy_value_s(a)}{number_of_entities_s(a)}$$

where $fuzzy_value_s(a)$ is the sum of the fuzzy values of the detected instances of the smell s in the app a and $number_of_entities_s(a)$ is the number of the entities concerned by the smell s in the app a. As a reminder, for BLOB, CC and BTMT, the concerned entity is the *class*, while for LM it is the *method*. For MAVC and ILMW, it is the *view controller* and for SAK it is the *interface*.

Table 3.3 shows that the median ratio for *SAK*, *MaVC* and *BTMT* is null. This is consistent with the results of Table 3.2, since these code smells appear in less than 35% of the apps, which means that in more than 70% of the apps the ratio is null. Consequently, the first quartile and the median for their ratios are null and the IQR is very small or almost null.

Smell	***Lang***	***Med***	***IQR***	***Cliff's*** δ
BLOB	*ObjC*	0.004	0.020	0.304(S)
	Swift	0.000	0.004	
LM	*ObjC*	0.060	0.055	0.135(I)
	Swift	0.048	0.059	
CC	*ObjC*	0.033	0.071	0.062(I)
	Swift	0.026	0.074	
SAK	*ObjC*	0.000	0.000	0.115(I)
	Swift	0.000	0.000	
MAVC	*ObjC*	0.000	0.015	0.181(S)
	Swift	0.000	0.000	
ILMW	*ObjC*	0.905	0.634	0.156(S)
	Swift	0.583	0.833	
BTMT	*ObjC*	0.000	0.000	0.069(I)
	Swift	0.000	0.000	

I: INSIGNIFICANT DIFFERENCE.
S: SMALL DIFFERENCE.

Table 3.3 Ratios comparison between Objective-C and Swift.

We also observe that the IQR is very small for all code smells except *ILMW*. These weak values show that these code smells are present in the apps with consistent ratios. As for *ILMW*, the high IQR value indicates that this smell is abundant in some apps, but nearly absent in others, and this strengthens our previous hypothesis about the *ILMW* origin. The majority of developers are not aware of the memory warnings importance and the *ILMW* ratio is high in their apps, while few other developers are aware of the issue and the *ILMW* ratio is very low in their apps.

The cliff's δ values show that the difference between the smell ratios in Objective-C and Swift is insignificant for *LM*, *CC*, *SAK*, and *BTMT*. Moreover, the median and the IQR for these smells are very close in the two datasets. Therefore, we deduct that these code smells are present in the two datasets with the same proportions and we cannot reject the study hypothesis HR^{CS} for $CS \in \{LM, CC, SAK, BTMT\}$.

For *BLOB*, *MaVC* and *ILMW*, there are *small* differences in the smell ratios in the two languages. However, these differences are not large enough to reject the hypothesis HR^{CS} for $CS \in \{BLOB, MaVC, ILMW\}$.

Discussion: The similar proportions of *ILMW*, *MaVC* and *BTMT* between Objective-C and Swift are quite expected, as these code smells are related to the platform concepts, which are shared by the two datasets regardless of the used programming language. Concerning OO smells, we expected significant differences between the two programming languages. Indeed, the metrics used to detect OO code smells (*e.g.*, complexity and number of lines) are dependent of the language structure and features. Thus, we expected that programming features that are present in Swift and not in Objective-C would result in a significant difference in code smell presence. However, our results confirm the absence of such difference.

To further investigate this point, we compared the metrics related to OO code smells in the two languages Objective-C and Swift. We computed for each metric the median, first and third quartile, and the standard deviation. We also used the Wilcoxon–Mann–Whitney test [174] to check if the distributions of our metrics in both datasets are identical, so that there is a 50% probability that an observation from a value randomly selected from the Objective-C dataset will be identical to an observation randomly selected from the Swift dataset. To perform this comparison, we used a 99% confidence level—*i.e.*, p-value < 0.01. Table 3.4 depicts the computed values for the metrics: *class complexity* (CLC), *number of attributes* (NOA), *number of methods* (NOM), *number of lines in a method* (NOL_M), *number of lines in a class* (NOL_C), and *number of methods in an interface* (NOM_I).

For the Wilcoxon–Mann–Whitney test, since the p-values for all the metrics are less than the threshold ($p = 0.01$), we can reject the hypothesis of similarity between the two distributions. Moreover, the results indicate that the quartile values in Objective-C are considerably higher than Swift, with also a much higher standard deviation. This means that the metrics range is wider, but also that there are more outliers in Objective-C.

These results show that the apps written in Objective-C and Swift are different in terms of OO metrics. Compared to Swift, Objective-C apps tend to have longer and more complex methods and classes, with more attributes and methods in classes and interfaces. This is likely due to the features introduced in Swift that help developers to better structure their apps. As an example, the *Extensions* to existing types, the *Structs*, and the advanced *Enums* are features that encourage developers to implement lighter classes in Swift.

Following this deduction, we can affirm that the presence of OO smells with the same proportions in iOS apps is not due to the similarity between the languages Objective-C and Swift. We rather hypothesize that it is due to the framework, which dictates the architecture/design and the development practices. In fact, the development community is almost the same for the two programming languages.

Metric	*Lang*	*Q1*	*MED*	*Q3*	*SD*	*p-value*
CLC	*ObjC*	2	6	16	22.34	0
	Swift	0	2	5	11.72	
NOA	*ObjC*	0	3	6	6.84	0
	Swift	0	0	1	2.58	
NOM	*ObjC*	1	4	8	8.57	0
	Swift	0	1	3	8.41	
NOL_M	*ObjC*	3	6	13	20.70	1.7e-39
	Swift	2	5	11	26.09	
NOL_C	*ObjC*	12	39	108	153.84	2.6e-196
	Swift	5	17	43	143.19	
NOM_I	*ObjC*	1	1	3	2.87	1.1e-227
	Swift	0	1	2	2.61	

Table 3.4 Metrics comparison between Objective-C and Swift.

3.4.6 Threats to Validity

We discuss here the main issues that may have threatened the validity of the validation study, by considering the classification of threats proposed in [54]:

Internal Validity: The threats to internal validity are relevant in those studies that attempt to establish causal relationships. In this case, the internal validity might be affected by the detection strategy of SUMAC. We relied on a robust set of standard metrics to evaluate the presence of well-defined code smells. However, for the threshold-based code smells, we computed the thresholds from the analysis of the whole dataset, to avoid influencing the detection results.

External Validity: This refers to the approximate truth of conclusions involving generalizations within different contexts. The main threat to external validity is the representativeness of the results. We used sets of 103 and 176 open-source Objective-C and Swift apps. It would have been preferable to consider proprietary apps to build a bigger and more diverse dataset. However, this option was not possible as the analysis of iOS binaries is restricted. We believe that our dataset still allows us to generalize our results to open-source iOS apps, but that further studies are needed to extend our results to all iOS apps.

Construct Validity: The threats to construct validity concern the relation between theory and observations. In this study, these threats could be due to errors during the analysis

process. We were very careful when collecting and presenting the results and drawing conclusions based on these results.

Conclusion Validity: This threat refers to the relation between the treatment and the outcome. The main threat to the conclusion validity in this study is the validity of the statistical tests applied. We alleviated this threat by applying a set of commonly accepted tests employed in the empirical software engineering community [136]. We paid attention not to make conclusions that cannot be validated with the presented results.

3.5 Comparative Study Between iOS and Android

In this section, we present our comparative study between iOS and Android apps. This study aims to answer our second research question:

RQ2: Is there a difference between iOS and Android in terms of code smells presence?

3.5.1 Android Dataset and Inclusion Criteria

To compare iOS and Android apps, we used our iOS dataset described in Section 3.4.5 and we built a new dataset of Android apps. We created the dataset by first collecting all apps available in the F-Droid repository [2] in April 2016. Afterwards, we excluded all the apps that could not be analyzed by PAPRIKA for bugs or crashes. This resulted in a set of 1,551 open-source Android apps. The full set can be found with our artifacts online. [3] Our dataset apps differ both in internal attributes, such as their size, and external attributes from the perspective of end users, such as user ranking and number of downloads.

3.5.2 Variables and Hypotheses

Independent variables: The nature of the app, Android or iOS, is the independent variable of our study.

Dependent variable: The dependent variables correspond to:

- The average proportion of the OO smells: BLOB, LM, CC, SAK;
- The average proportion of the similar iOS and Android smells: NLMR/ILMW, HEAVY/BTMT.

Hypotheses: To answer our second research question, we formulate the following null hypothesis, which we applied to the iOS and Android datasets ($CS \in \{BLOB, LM, CC, SAK, NLMR/ILMW, HEAVY/BTMT\}$):

HR^{CS}: There is no difference between the *proportions* of code smells in Android and iOS apps.

2. F-Droid: `https://f-droid.org`
3. `http://sofa.uqam.ca/paprika/paprika_ios.php`

3.5.3 Analysis Method

To compare our results, we used the median and the interquartile range. We also computed the Cliff's δ effect size to quantify the impact of the platform on code smells presence. Later, to discuss and explain the obtained results we compare the metric values in the two platforms using the Wilcoxon and Mann-Whitney test. As mentioned previously, Cliff's δ and Mann-Whitney make no assumption about the assessed variables distribution, and both of them are suitable for comparing datasets of different sizes.

3.5.4 Results

This section reports and discusses the results of our comparative study.

Overview of Android Results

	BLOB	***LM***	***CC***	***SAK***	***HEAVY***	***NLMR***
Apps %	78.40	98.19	86.59	12.31	41.84	98.32

Table 3.5 Percentage of Android apps affected by smells.

We can observe in Table 3.5 that *BLOB*, *LM* and *CC* appear in most of the apps (more than 75% in all cases). Indeed, most apps tend to contain at least one of these code smells. On the other side, just like in iOS, *SAK* is uncommon and only appears in 12% of the analyzed apps. Concerning Android-specific code smells, the *HEAVY* code smells appear in almost half of the apps. After inspection, we observed that, usually, there is only one instance of these code smells in each app. *NLMR* is our most common code smell with more than 98% of the apps being affected by it. Here, we also notice that there is often only one activity affected per app.

Comparison of Code Smell Proportions Between iOS and Android

We compared the ratio between the number of code smells and the number of concerned entities for each code smell in every app. The medians, IQR and Cliff's δ effect sizes of the distributions obtained are presented in Table 3.6. The Cliff's δ values column presents for each smell the Cliff test for Android with Objective-C, then for Android and Swift, respectively.

Table 3.6 shows that we have a significant difference between the proportions of code smells in Android compared to Objective-C and Swift for all code smells except *SAK*. For OO smells, Cliff's δ always gives a large difference for *BLOB*, *CC* and *LM*. Furthermore, this is confirmed by the greater means in Android apps for these smells. Thus, we can reject the hypothesis HR^{CS} for all OO code smells *BLOB*, *CC* and *LM*. On the other hand, we

Smell	***Lang***	***Med***	***IQR***	***Cliff's δ***
BLOB	*ObjC*	0.004	0.020	0.495(L) 0.648(L)
	Android	0.052	0.129	
	Swift	0.000	0.004	
LM	*ObjC*	0.060	0.055	0.710(L) 0.727(L)
	Android	0.156	0.289	
	Swift	0.048	0.059	
CC	*ObjC*	0.033	0.071	0.509(L) 0.518(L)
	Android	0.122	0.317	
	Swift	0.026	0.074	
SAK	*ObjC*	0.000	0.000	-0.121(I) -0.015(I)
	Android	0.000	0.000	
	Swift	0.000	0.000	
NLMR/ILMW	*ObjC*	0.905	0.634	0.397(M) 0.512(L)
	Android	1.000	0.000	
	Swift	0.583	0.833	
HEAVY/BTMT	*ObjC*	0.000	0.000	0.261(S) 0.331(M)
	Android	0.000	0.007	
	Swift	0.000	0.000	

I: insignificant difference.
S: small difference.
M: medium difference.
L: large difference.

Table 3.6 Ratios comparison between Android and iOS

cannot reject HR^{SAK} since Cliff's δ shows no significant difference and its median and IQR are almost the same in the two platforms. That is, *SAK* is uncommon in both Android and iOS apps.

Concerning mobile code smells, we can reject the hypothesis HR^{NLMR}. Regardless of the programming language, Android apps have more instances of the code smell *NLMR*. Indeed, *NLMR* appears in almost every Android app (*cf.* Table 3.5), while there are many iOS apps without *ILMW* (*cf.* Table 3.2). We can also reject the hypothesis HR^{HEAVY} since Android apps have bigger proportions of the *HEAVY* code smells compared to iOS apps written in Objective-C and Swift (small and medium differences).

In summary, Android apps tend to host more code smells than iOS apps. The only exception to this finding is the *SAK* code smell, which is equally present in both platforms.

Discussion: The comparison results indicate that Android apps generally have more code smells than iOS apps. However, most of the compared code smells rely on numerical metrics. *NLMR/ILMW* are the only code smells that rely on boolean metrics. Hence, our results can be interpreted as Android apps having more (numerical) outliers than iOS apps. Hence, the observation of more code smells in Android apps does not necessarily mean that Android apps are globally worse in terms of quality.

To further investigate this point, we examined the quality metrics in the two platforms. In particular, we compared the distributions of common metrics used to detect code smells in Android and iOS apps. We only excluded the metrics related to the number of lines because Sumac computes the number of lines, while Paprika accounts the number of instructions. As the number of instructions does not always correspond to the number of lines, we cannot compare the two metrics.

The results of this comparison are presented in Table 3.7. The *p*-values reported in this table show a disparity between the metrics values in the different datasets. Except for the complexity, the metrics in Android are different from both Objective-C and Swift. Objective-C has the highest values in the median and the third quartile. Android has also high metrics values, but less than Objective-C. This is quite expected since Android apps are written with Java, a classic programming language known for being less verbose than C++ and Objective-C. Last, for Swift it is clear that the modern features of the language are efficient in making apps lighter, which implies lower metric values.

Discussion: These results show that Android apps do not have the highest metric values, but they show the highest standard deviation. This means that in terms of OO metrics, Android apps are not bigger than iOS apps. The high number of code smells in Android is rather due to the high number of outliers. These outliers may be attributed to the platform specificities, like the architecture and framework constraints, or to the lack of documentation and code quality assessment tools.

Metric	*Lang*	*Q1*	*MED*	*Q3*	*SD*	*p-value*
CLC	*ObjC*	2	6	16	22.34	0.71 0
	Android	3	5	13	35.92	
	Swift	0	2	5	11.72	
NOA	*ObjC*	0	3	6	6.84	1e-48 0
	Android	1	2	4	22.25	
	Swift	0	0	1	2.58	
NOM	*ObjC*	1	4	8	8.57	1.57-11 0
	Android	2	4	7	12.61	
	Swift	0	1	3	8.41	
NOM_I	*ObjC*	1	1	3	2.87	0.26 2.9e-140
	Android	1	1	3	6.97	
	Swift	0	1	2	2.61	

Table 3.7 Metrics comparison between iOS and Android.

3.5.5 Threats to Validity

In this subsection, we analyze the factors that may threaten the validity of this study using the same classification applied in Section 3.4.6:

Internal Validity: Again, in this case, the internal validity might be affected by the detection strategy of Paprika. For the Android dataset, we tried to rely on a robust set of metrics to evaluate the presence of code smells. Moreover, for the threshold-based code smells, we calculated the thresholds based on the analysis of the whole dataset, so as not to influence the detection results.

External Validity: In this study, we analyzed a large, heterogeneous dataset of Android open-source apps available on F-Droid. In a similar way, we tried to use as many open-source iOS apps as possible, as mentioned in the previous study. Despite our efforts, the iOS datasets are significantly smaller and thus probably less representative than the Android dataset. However, our analysis remains meaningful as we were careful to only conclude on statistical tests, which are suitable for samples of different sizes. We believe that our three datasets provide a neutral and diverse set of apps, but we are focusing only on open-source apps and thus we cannot generalize our results to proprietary apps. Further investigations are needed to confirm that open-source and proprietary apps are similar in terms of code smells.

Construct Validity: The construct validity can be threatened by the way we are computing thresholds. Indeed, we could use the same thresholds for all detections (*e.g.*, by using an average threshold between Android and iOS). However, we believe that in this case the thresholds will not be relevant regarding the contexts. Moreover, boxplots are commonly used for assessing values in empirical studies [136].
Conclusion Validity: As before, the main threat to the conclusion validity in this study is the validity of the statistical tests applied and we alleviated this threat by applying the same tests.

3.6 Summary

In this chapter, we presented a study that explores code smells in the iOS mobile platform. We proposed a novel catalog of six code smells that we identified from the documentation and the developers' feedback. These code smells are specific to iOS and have not been reported in the literature before. We also proposed Sumac, an open-source tool [97] that detects code smells in iOS apps written with Objective-C and Swift.

Based on the code smells catalog and our tool Sumac, we conducted an empirical study to investigate the scope of code smells in mobile apps. We first analyzed 279 iOS apps and compared the proportions of code smells in apps written with Objective-C and Swift. We observed that Objective-C and Swift apps are very different in terms of quality metrics. In particular, Swift apps tend to be lighter in terms of size and complexity. However, these differences do not seem to impact the presence of code smells, since the two types of apps exhibit OO and iOS code smells with no significant disparity. Then, we compared iOS apps with Android apps in order to investigate the potential differences between the two platforms. The comparison showed that Android apps tend to have more code smells than iOS ones. Again, we demonstrated that this difference is not due to the programming language, but rather potentially to the platform.

This work provided interesting insights for both developers and scientific community. It highlighted some of the common code smells in iOS apps and provided relevant information about the code quality in a mobile platform, which has not been addressed by the community before. Moreover, we provided an open-source toolkit, Sumac, which is—to the best of our knowledge—the first to support the detection of code smells in iOS apps. This facilitates and encourages further studies on iOS apps.

Regarding our thesis plan, this work allowed us to fill in many gaps in the literature about mobile-specific code smells and provided a first contribution towards understanding them. Notably, our catalog showed that there are common code smells between iOS and Android. This commonality can help future work to generalize their studies of code smells to many mobile platforms. In our upcoming studies, we address the second gap identified

in our literature review. In particular, we investigate the rationales behind the accrual of mobile-specific code smells.

Chapter 4

Code Smells in the Change History

In our review of the state of the art, we showed that the existing studies did not explain the rationales behind the accrual of mobile-specific code smells. Hence, our second objective for this thesis was to understand the motives of these code smells—*i.e.*, the reasons behind their prevalence in mobile apps. To investigate this point, we refer to the existing literature about technical debt, which reports many potential motives. This reference is justified as mobile-specific code smells can be considered as a form of technical debt. Indeed, they represent *"a gap between the current state of a software system and some hypothesized ideal state in which the system is optimally successful in a particular environment"*, which is a definition of technical debt [37].

Technical debt is a well founded concept in software engineering. Since its definition by Cunningham *et al.* [58], researchers and practitioners have been investigating its rationales. In particular, the study of Tom *et al.* [191] built a theoretical framework that explains, among other aspects, the precedents of technical debt. The framework shows that the reasons behind the presence of technical debt fall under six categories:

- **Pragmatism:** This refers to the internal choices made by stakeholders to focus on some product aspects to the detriment of source code quality;
- **Prioritization:** This represents all external factors that push development teams to prioritize some tasks above source code quality. According to developer interviews [191], these factors are generally *time*, *budget*, and *resource constraints*;
- **Processes:** This precedent covers all the settings adopted by teams in their development process—*e.g.*, code review, communication, and collaboration settings;
- **Attitudes:** This represents the individual behaviors of developers that may lead to unmanaged technical debt. These behaviors are usually the results of inexperience, general apathy toward quality issues, and risk appetite;
- **Ignorance:** This refers to situations where developers are unaware of how to avoid technical debt. Cast *et al.* [42] described this as *"the inability of the developers to develop high quality applications"*;

— **Oversight:** This motive explains situations where developers do not apprehend the issue of technical debt, which makes them unaware of its presence.

Our objective is to assess the relevance and impact of these motives in the case of mobile-specific code smells. This assessment is important for all future approaches and tools that intend to cope with these code smells. To realize this assessment, we opt in this chapter for an investigation of the change history of mobile apps. The latter entails details about how these code smells appear, evolve, and disappear from the source code. Hence, by analyzing it we can measure how different motives impact the lifetime of code smells, and thus assess their relevance.

As the motives of interest cover different development aspects, we adopted different approaches to study them. Specifically, we perform three large-scale empirical studies:

1. **Evolution study:** This study analyzes the introductions and removals of mobile-specific code smells aiming to understand how prioritization factors impact them. The study analyzes also the actions leading to code smell removals to inspect motives like pragmatism, ignorance, and oversight;
2. **Developers study:** This study intends to understand the role played by developers in the accrual of mobile-specific code smells. This study covers the three motives that are developer-oriented, namely attitude, ignorance, and oversight;
3. **Survival study:** This study investigates the survival of mobile-specific code smells in the change history. In particular, it analyzes the impact of processes and prioritization factors on the persistence of code smells in the codebase.

To perform these studies, we developed SNIFFER [104], a novel toolkit that accurately mines the change history of Android apps to track mobile-specific code smells. SNIFFER tackles many issues raised by the Git mining community like branch and renaming tracking. We used SNIFFER in our three studies to analyze eight Android code smells, 324 Android apps, $255k$ commits, and contributions from $4,525$ developers. We provide this collected data in a database that includes 180.013 code smell histories [102]. This database can be used in future studies that approach mobile-specific code smells.

The remainder of this chapter is organized as follows. Section 4.1 explains the design, which is common to our three empirical studies. Section 4.2 presents the design and implementation of our toolkit SNIFFER. Section 4.3 reports the evolution study, while Section 4.4 reports the developers study, and Section 4.5 presents the survival study. Section 4.6 highlights the common threats to validity for our three studies, and finally, Section 4.7 concludes with our main learned lessons.

This chapter includes a revised version of a paper under evaluation for JSS'19, and materials published in the proceedings of MSR'19 [101] and MOBILESoft'19 [103].

4.1 Study Design

We present in this section the relevant context for our research investigation and the dataset leveraged in our three empirical studies.

4.1.1 Context Selection

The core of our studies is analyzing the change history of mobile apps to follow the evolution of mobile-specific code smells. In this respect, the context selection entails the choice of (i) the mobile platform to study, and (ii) the mobile-specific code smells with their detection tool.

The Mobile Platform

We decided to focus our studies on the Android platform. With 85.9% of the market share, Android is the most popular mobile operating system as of 2018. [1] Moreover, more Android apps are available in open-source repositories compared to other mobile platforms, like iOS [100]. On top of that, both developers and research communities proposed tools to analyze the source code of Android apps and detect code smells [7, 109, 156].

Code Smells & Detection Tool

In the academic literature, the main reference to Android code smells is the catalog of Reimann *et al.* [169]. It includes 30 code smells, which are mainly performance-oriented, and covers various aspects, like user interface and data usage. Ideally, we would consider all the 30 code smells in our studies. However, for feasibility purpose, we could only consider code smells that are already detectable by state-of-the-art tools. Hence, the choice of studied code smells is determined by the adopted detection tool. In this regard, our detection relied on Paprika, an open-source tooled approach that detects Android-specific code smells from Android packages (APK). Paprika is able to detect 13 Android-specific code smells. However, after examination, we found that the code smells *Invalidate Without Rect* and *Internal Getter Setter* are now deprecated. Indeed, the method `invalidate()` was deprecated in API level 28, which deprecates the code smell *Invalidate Without Rect* [8]. As for *Internal Getter Setter*, since API level 10, internal getters are automatically optimized by Dalvik, thus the code smell is not valid anymore [7]. To address these deprecations, we excluded these two code smells from our analysis. Moreover, we wanted to focus our studies on objective code smells—*i.e.*, smells that either exist in the code or not, and cannot be introduced or removed gradually. Hence, we excluded *Heavy AsyncTask*, *Heavy Service Start* and *Heavy BroadcastReceiver*,

1. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems

which are subjective code smells [109]. We present in Table 4.1 the studied code smells with a highlight on the affected source code entities and the main performance impact.

Leaking Inner Class (*LIC*): in Android, anonymous and non-static inner classes hold a reference of the containing class. This can prevent the garbage collector from freeing the memory space of the outer class even when it is not used anymore, and thus causing memory leaks [7, 169].

Entity: Inner class.

Impact: Memory.

Member Ignoring Method (*MIM*): this smell occurs when a method that is not a constructor and does not access non-static attributes is not static. As the invocation of static methods is 15%–20% faster than dynamic invocations, the framework recommends making these methods static [108].

Entity: Method.

Impact: CPU.

No Low Memory Resolver (*NLMR*): this code smell occurs when an `Activity` does not implement the method `onLowMemory()`. This method is called by the operating system when running low on memory in order to free allocated and unused memory spaces. If it is not implemented, the operating system may kill the process [169].

Entity: Activity.

Impact: Memory.

Hashmap Usage (*HMU*): the usage of `HashMap` is inadvisable when managing small sets in Android. Using Hashmaps entails the auto-boxing process where primitive types are converted into generic objects. The issue is that generic objects are much larger than primitive types, 16 and 4 bytes, respectively. Therefore, the framework recommends using the `SparseArray` data structure that is more memory-efficient [7, 169].

Entity: Method.

Impact: Memory.

UI Overdraw (*UIO*): a UI Overdraw is a situation where a pixel of the screen is drawn many times in the same frame. This happens when the UI design consists of unneeded overlapping layers, *e.g.*, hidden backgrounds. To avoid such situations, the `canvas.quickreject()` API should be used to define the view boundaries that are drawable [7, 169].

Entity: View.

Impact: GPU.
Unsupported Hardware Acceleration (*UHA*): in Android, most of the drawing operations are executed in the GPU. Rare drawing operations that are executed in the CPU, *e.g.*, `drawPath` method in `android.graphics.Canvas`, should be avoided to reduce CPU load [106, 143]. **Entity**: Method. **Impact**: CPU.
Init OnDraw (*IOD*): *a.k.a.* DrawAllocation, this occurs when allocations are made inside `onDraw()` routines. The `onDraw()` methods are responsible for drawing `Views` and they are invoked 60 times per second. Therefore, allocations (*init*) should be avoided inside them in order to avoid memory churn [7]. **Entity**: View. **Impact**: Memory.
Unsuited LRU Cache Size (*UCS*): in Android, a cache can be used to store frequently used objects with the *Least Recently Used* (LRU) API. The code smell occurs when the LRU is initialized without checking the available memory via the method `getMemoryClass()`. The available memory may vary considerably according to the device so it is necessary to adapt the cache size to the available memory [106, 137]. **Entity**: Method. **Impact**: Memory.

Table 4.1 Studied code smells.

4.1.2 Dataset and Selection Criteria

To retrieve the apps and their change history, we rely on GitHub—the largest social coding platform. In order to have a representative dataset, we only include real Android apps—*i.e.*, we exclude demo apps, libraries, etc. To this end, we need a criterion to distinguish Android apps from projects of other frameworks. One option could be to use GitHub search on repositories' titles, descriptions, and topics. The issue with this option is that GitHub repositories that satisfy this criterion could also be tests, templates, or even APIs. Thus, it is not possible to rely only on the search option of GitHub. We therefore opted for the apps published by the FDroid online repository [2] to create our dataset. We used a web

2. `https://f-droid.org`

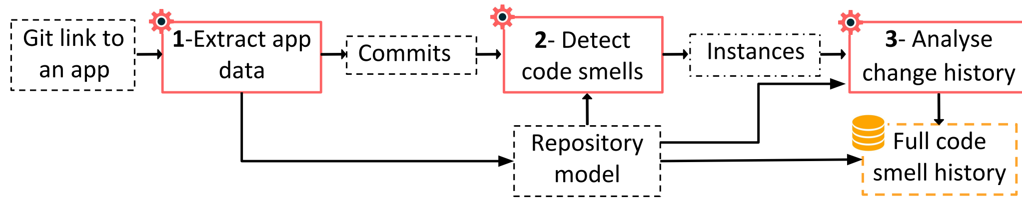

Figure 4.1 Overview of the SNIFFER toolkit.

crawler to collect the GitHub links of apps available on FDROID. Then using these links, we fetched the repositories from GitHub. [3] For computational constraints and in order to focus on repositories with communities, we only kept projects with at least two developers. This filter resulted in 324 projects with 255,798 commits. The full list of projects can be found in our artifacts [95].

4.2 Data Extraction: Sniffer

We propose in this section SNIFFER, a novel tool for tracking code smells in Android apps. SNIFFER is an open-source [104], heavily tested, and documented toolkit that tracks the full history of Android-specific code smells. It tackles many issues raised by the Git mining community like branch and renaming tracking [120]. Figure 4.1 illustrates its main steps of SNIFFER. First, from the repository of the app under study, it extracts the commits and other necessary metadata like branches, releases, commit authors, etc. In the second step, it analyzes the source code of each commit separately to detect code smells instances. Finally, based on the code smell instances and the repository metadata, it tracks the history of each smell and records it in the output database. In the following subsections, we explain the role of each step with a brief description of its implementation.

Step 1: Extract App Data

Input: Git link to an Android app.
Output: Commits and repository model.
Description: This step is performed by the sub-module `CommitLooper` [104]. First, SNIFFER clones the repository from Git and parses its log to obtain the list of commits. Afterwards, it analyzes the repository to extract its model. This model consists of properties of different repository elements. Table 4.2 describes the main properties of commits, developers, and releases.

While most of the properties are directly extracted from Git, some others need more processing to be retrieved. In particular, the SDK version is retrieved from the `manifest.xml`

3. `http://ghtorrent.org`

Element	*Property*	*Description*
Commit	`size`	`(int,int)`: the number of additions and deletions.
	`author`	The commit author.
	`date`	The commit authoring date.
	`message`	The commit message.
	`is_merge`	`(boolean)`: `true` if it is a merge commit.
	`original_branch`	The branch in which the commit was first introduced.
	`parents`	The previous commits.
	`sdk_version`	The Android SDK version used in the commit.
Developer	`number_of_commits`	The number of commits authored by the developer.
Release	`date`	The date of the release.
	`commit`	A reference to the commit pointed by the release.
	`message`	The release message.

Table 4.2 Repository metadata.

file, which describes the configuration of the Android app. Also, properties related to branches are not directly available. Branches are a local concept in Git, thus information about them are not recorded to be easily retrieved. Indeed, given a specific commit, Git cannot tell what is its original branch. For this, Sniffer makes an additional analysis to attribute each commit to its original branch.

Branch extraction: The distribution of commits among branches can only be retrieved by navigating the repository commit tree. Figure 4.2 shows an example of a commit tree. In the tree, every commit points towards one or multiple parents—*i.e.*, precedent commits. If we navigate the first parents, we will only get the commits of the master branch, `G->D->C->B->A` in this example. Retrieving the full history of the repository, with all the branches, requires navigating all the parents. Therefore, Sniffer navigates the commit tree by crossing all the parents and extracts the repository branches. Then, it updates the branch-related properties. For instance, commit `D` will have `original_branch = master` and `parents = {C,F}`. These properties will allow us to track the smell history accurately in *Step 3*.

Step 2: Detect Code Smells

Input: Commits and repository model.
Output: Code smell instances per commit.
Description: This step is performed by the sub-module `SmellDetector` [104]. As explained beforehand, Sniffer relies on Paprika [109] to detect Android code smells. Originally,

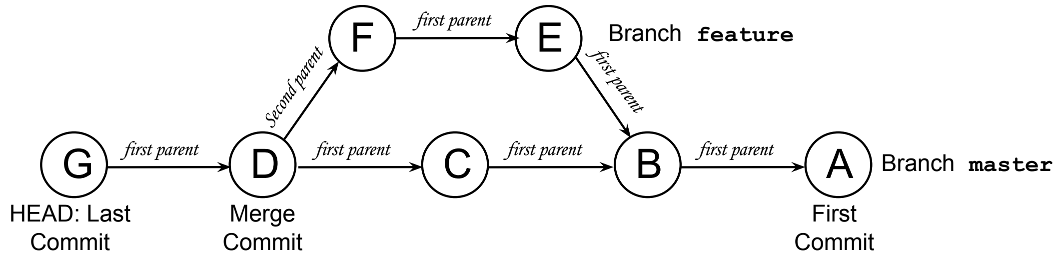

Figure 4.2 An example of a commit tree.

PAPRIKA detects code smells from the APK, it does not analyze the source code. However, we wanted to detect code smells directly from the source code of commits. Therefore, we needed to integrate a static analyzer into SNIFFER that feeds PAPRIKA with the necessary data. In such a way, before going through smell detection with PAPRIKA, each commit goes through the static analysis.

Static analysis: SNIFFER performs static analysis using SPOON [160], a framework for Java-based programs analysis and transformation. SNIFFER launches SPOON [4] on the commit source code to build an abstract syntax tree. Afterwards, it explores this tree to extract code entities (*e.g.*, classes and methods), properties (*e.g.*, names, types), and metrics (*e.g.*, number of lines, complexity). Together, these elements constitute a usable model by PAPRIKA.

Detection of code smell instances: Fed with the model built by the static analyzer, PAPRIKA detects the code smell instances. To this point, commits are still processed separately. Thus, this step produces a separate list of code smell instances for each commit.

Step 3: Analyze Change History

Input: Code smell instances per commit and the repository model.
Output: Full code smell history.
Description: This step is performed by the sub-module `SmellTracker` [104]. In this step, SNIFFER tracks the full history of every code smell. As our studies focus on objective Android code smells, we only have to look at the current and previous commits to detect smell introductions and removals. If a code smell instance appears in a commit while absent from the previous, a smell introduction is detected. In the same way, if a commit does not exhibit

4. `http://spoon.gforge.inria.fr/`

an instance of a smell that appeared previously, a smell removal is detected. In order for this process to work, Sniffer needs to (a) retrieve the previous commits accurately and (b) track renamings.

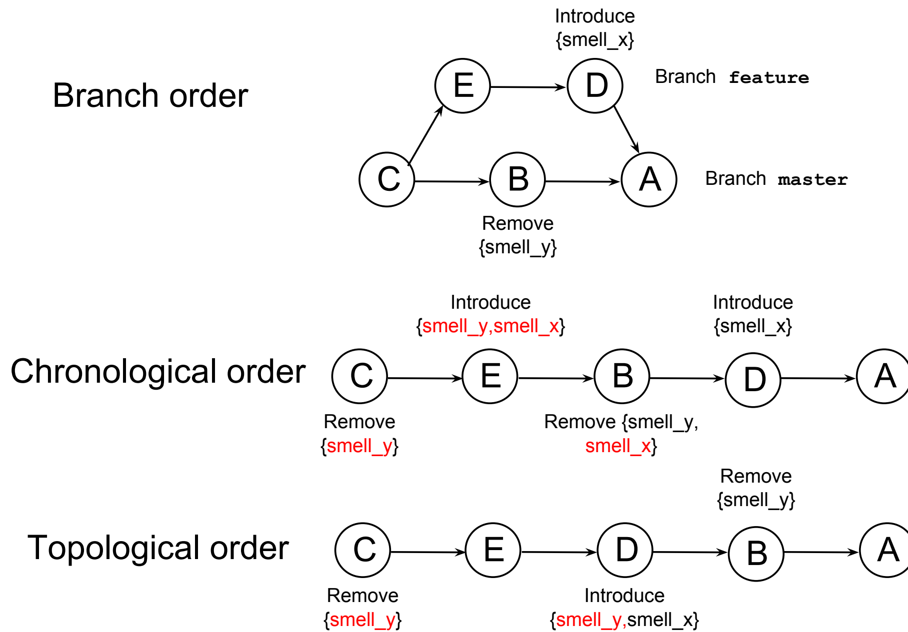

Figure 4.3 Commits ordered in different settings.

Retrieve the previous commits accurately: Retrieving the previous commits requires an order in the commit history. In this regard, Git log provides two main commit orders:

— *Chronological* order, by date or authoring date,

— *Topological* order.

The *chronological* order aligns commits relying only on the date, but it does not consider their branches. As for the *topological* order, it aligns the commits of all the branches to make sure that children commits are always shown before their parents. These two orders flatten the commit tree into one branch and this implies a loss of accuracy. This loss affects the detection of code smell introductions and removals. To illustrate this effect, we depict in Figure 4.3 an example of a commit tree with two branches. The branch order presents the real commit history where the commit D introduced smell_x and commit B removed smell_y. With the chronological order, commit B is placed between commits D and E. With the topological order, commits D and E are placed between B and C. Moreover, in both orders the commit C has only one previous commit instead of two. As shown in Figure 4.3, these placements caused many false code smell introductions and removals. This shows that these orders are suitable for repositories with a single branch. However, for multiple branches, we need to follow a branch-aware order. For this purpose, Sniffer considers the branches extracted in *Step 1* to retrieve previous commits. Specifically, it uses the commit properties

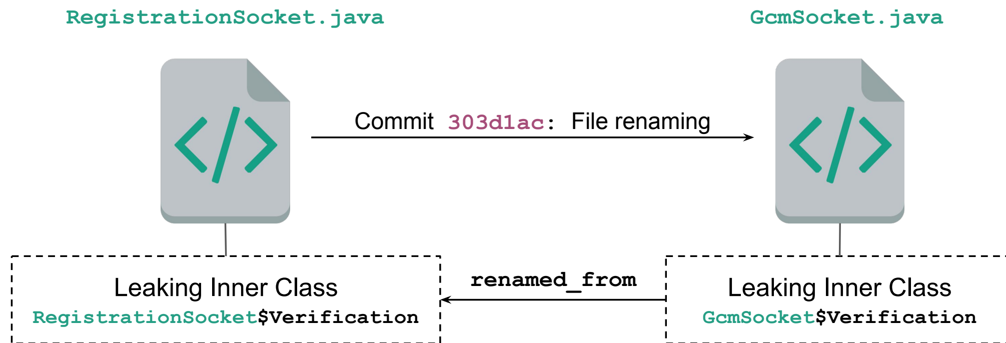

Figure 4.4 Tracking code smell renamings.

original_branch and parents to get one or multiple previous commits. This allows to stay faithful to the real order of commits and accurately detect code smell introductions and removals.

Track renamings: In the process of development, files and folders can be renamed. These renamings hinder the tracking of code smells. For instance, Figure 4.4 depicts an example of a file renaming that happens in the application Silence [176].

If we do not detect this renaming, we would consider `GcmSocket$Verification` as a new instance of the code smells *Leaking Inner Class* (LIC), and we will lose track of the instance `RegistrationSocket$Verification`. On top of that, the commit `303d1ac` will be attributed with wrong smell introduction and removal. To prevent these mistakes, Sniffer relies on Git to detect all the renamings happening in the repository. Git tracks files with their contents instead of their names or paths. Hence, if a file or a folder is renamed and the content is not drastically changed, Git is able to keep track of the file. Git uses a similarity algorithm [86] to compute a similarity index between two files. By default, if the similarity index is above 50 %, the two files are considered the same. Sniffer uses this feature via the command `git log -find-renames -summary`, which shows all the renamings that happened in the repository. Sniffer uses these detected renamings to link the files and code smells in the output database. In this way, Sniffer detects in the example of Figure 4.4 that the code smell `RegistrationSocket$Verification` is a new name for `RegistrationSocket$Verification`. It links the two code smells with the relationship `renamed_from` and stores this information in the output database. Consequently, no new code smell introductions or removals are detected, and the history of code smells is accurately tracked.

The generated history is saved in a PostgreSQL database that can be queried to analyze different aspects of mobile apps and code smells [95].

4.2.1 Validation

In order to ensure the relevance of our studies, we validated the performance of our data extraction by assessing the accuracy of Sniffer and its underlying tool Paprika on our dataset. The validation samples are available in our companion artifacts [95].

Paprika

Hecht *et al.* [106] have already validated Paprika with a F1-score of 0.9 in previous studies. They validated the accuracy of the used code smell definitions and the performance of the detection. The objective of our validation of Paprika is to check that its detection is also accurate on our dataset. Therefore, we randomly selected a sample of 599 code smell instances. We used a stratified sample to make sure to consider a statistically significant sample for each code smell. This represents a 95 % statistically significant stratified sample with a 10 % confidence interval of the 180,013 code smell instances detected in our dataset. The stratum of the sample is represented by the eight studied code smells. After the selection, one author manually analyzed the instances to check their correctness. We found that all the sample instances are conform to the adopted definitions of code smells. Hence, we can affirm that the Paprika code smell detection is effective in our dataset.

Sniffer

We aimed to validate the accuracy of the code smell history generated by Sniffer. For this, we randomly selected a sample of 384 commits from our dataset. This represents a 95 % statistically significant stratified sample with a 5 % confidence interval of the 255,798 commits in our dataset. After the selection, one author analyzed every commit to check that the detected code smell introductions and removals are correct, and Sniffer did not miss any code smell introduction or removal. Based on these results, we computed the number of *true positives* (TP), *false positives* (FP), and *false negatives* (FN). These numbers are reported in Table 4.3.

	TP	***FP***	***FN***	***Precision***	***Recall***	***F1-score***
Introductions	151	7	0	0.95	1	0.97
Removals	85	7	0	0.92	1	0.96

Table 4.3 Validation of Sniffer.

We did not find any case of missed code smell introductions or removals, $FN = 0$. However, we found cases where false code smell introductions and removals are detected: $FP = 7$ for both of them. These false positives are all due to a commit from the Shopping List app [82], which renamed 12 Java files. Three of these renamings were accompanied with

major modifications in the source code. Thus, the similarity between the files was above 50 % and Git could not detect the renaming. Consequently, SNIFFER could not track the code smells of these files and detected seven false code smell introductions and removals.

Using the results of the manual analysis, we computed the following performance measures.

$$\text{Precision} = \frac{TP}{TP+FP} \quad \text{Recall} = \frac{TP}{TP+FN}$$

$$\text{F1-score} = 2 \times \frac{\text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$$

The values of these measures are reported in Table 4.3. According to these values, SNIFFER is effective for detecting both code smell introductions and removals.

4.3 Evolution Study

In this study, we analyze the introductions and removals of Android code smells to identify the motives that favor the presence of code smells. In particular, we aim to answer the following research questions:

— **RQ 1:** How frequent are code smell introductions?
  This question allows us to quantify the extent of code smells in the change history.

— **RQ 2:** How do releases impact code smell introductions and removals?
  We focus on releases, as they are usually considered as the main prioritization factor that leads to technical debt [191]. Moreover, mobile apps are known for having more frequent releases and updates [138], which may be another favorable factor for mobile-specific code smells.

— **RQ 3:** How are code smells removed?
  This question allows us to:
  — Check whether code smells are removed intentionally or not. This check is important to inspect developers' awareness about code smells and thus assess motives like pragmatism, ignorance, and oversight;
  — Learn removal fashions from developer actions and aliment future studies about code smell refactoring.

In the following subsections, we will explain how we leveraged the data extracted by SNIFFER to answer these research questions. Afterwards, we report and discuss the study results, then we conclude with a study summary that highlights the main findings.

4.3.1 Data Analysis

In this subsection, we describe our approach for analyzing the collected data in order to answer our research questions. Table 4.4 reports on the list of metrics that we defined for the data analysis.

	Metric	Description
Code smell type	#introductions	The number of instances introduced in the dataset.
	%affected-apps	The percentage of apps affected by the code smell.
	%diffuseness	The diffuseness of the code smell instances in the source code of an app.
	#removals	The number of instances removed in the dataset.
	%removals	The percentage of instances removed—*i.e.*, $\frac{\#\text{removals}}{\#\text{introductions}}$.
	#code-removed	The number of instances removed with source code removal.
	%code-removed	The percentage of instances removed with source code removal—*i.e.*, $\frac{\#\text{code-removed}}{\#\text{removals}}$.
Commit	#commit-introductions	The number of code smell instances introduced by the commit.
	#commit-removals	The number of code smell instances removed by the commit.
	distance-to-release	The distance between the commit and the next release in terms of number of commits.
	time-to-release	The distance between the commit and the next release in terms of number of days.

Table 4.4 Study metrics.

As shown in Table 4.1, every code smell type affects a specific entity of the source code. Therefore, to compute the metric %diffuseness, we only focus on these entities. For instance, the code smell *Init OnDraw* affects only the entity `View`, thus we compute the percentage of views affected. This allows us to focus on the relevant parts of the source code and have a precise vision about the code smell diffuseness. For each app a, the diffuseness of a type of code smells t that affects an entity e is defined by:

$$\text{\%diffuseness}(a,t) = \frac{\#\text{affected-entities}(a,t)}{\#\text{available-entities}(a,e)}$$

For the metrics #code-removed and %code-removed, we analyzed the code smell history built by SNIFFER to detect the source code modifications that accompanied code smell removals. In particular, we counted every code smell removal where the host entity of the code smell was also removed. For instance, when an instance of the code smell *No Low Memory Resolver* is removed, the removal can be counted in #code-removed only if the host `Activity` has been removed in the same commit.

RQ1: How Frequent Are Code Smell Introductions?

To inspect the prevalence of code smells, we first computed the following metrics for each code smell type: #introductions and %affected-apps. These metrics allow us to compare the

prevalence of different code smell types. Then, to have a precise assessment of this prevalence, we also used the metric: %diffuseness. We computed the diffuseness of each code smell type in every app of our dataset. Finally, we plotted the distribution to show how diffuse are code smells compared to their host entities.

RQ2: How Do Releases Impact Code Smell Introductions and Removals?

This research question investigates the impact of releases on code smell evolution. To ensure the relevance of this investigation, we paid careful attention to the suitability of the studied apps for a release inspection. In particular, we manually checked the timeline of each app to verify that it publishes releases through all the change history. We excluded apps that did not use releases at all, and apps that used them only at some stage. For instance, the Chanu app [145] only started using releases in the last 100 commits, while the first 1,337 commits do not have any release. Hence, this app is, to a large extent, release-free and thus irrelevant for this research question. Out of the 324 studied apps, we found 156 that used releases during all the change history. The list of these apps can be found in our study artifacts [96]. It is also worth noting that as Android apps are known for continuous delivery and releasing [10, 138], we considered in this analysis both minor and major releases. This allows us to perform a fine-grained study with more releases to analyze.

After selecting relevant apps, we evaluated the impact of releases on code smell introductions and removals. First, we visualized for each project the evolution of source code and code smells along with releases. The objective of this visualization is not to evaluate the impact of releases on code smells, but it rather gives us insights about it. To actually measure this impact, we needed to analyze the impact of approaching releases on the numbers of introductions and removals in the commit. Therefore, we used the metrics distance-to-release and time-to-release.

Distance to release: We aimed to assess the relationship between the distance to release and the numbers of code smells introduced and removed per commit. For this, we computed the correlation between the distance-to-release and both #commit-introductions and #commit-removals using Spearman's rank coefficient. Spearman is a non-parametric measure that assesses how well the relationship between two variables can be described using a monotonic function. This measure is adequate for our analysis as it does not require the variables normality and does not assess the linearity.

Time to release: Using the metric time-to-release, we extracted three commit sets:
— Commits authored one day before a release,
— Commits authored one week before a release,
— Commits authored one month before a release.

Then, we compared the #commit-introductions and #commit-removals in the three sets using Mann-Whitney U and Cliff's δ. We used the two-tailed Mann-Whitney U test [174] with a 99% confidence level, to check if the distributions of introductions and removals are identical in the three sets. To quantify the effect size of the presumed difference between the sets, we used Cliff's δ [170].

RQ3: How Are Android Code Smells Removed?

Quantitative analysis: First, to assess how frequently are code smells removed, we computed for each type of code smells the following metrics: #removals and %removals. Then, to have insights about the actions that lead to code smell removals, we computed: #code-removed and %code-removed. The metric %code-removed reports on the percentage of code smell instances that were removed with a source code deletion. Code smells being removed because the source code is deleted may be a sign that the removal is accidental and does not originate from an intended refactoring. However, this sign is not enough to conclude about the removal nature, hence the need for a thorough qualitative analysis of code smell removals.

Qualitative analysis: The objective of the qualitative analysis is to inspect code smell removal fashions and check whether these removals manifest any intentional refactoring behavior. For this, we manually analyzed a sample of smell-removing commits. We used a stratified sample to make sure to consider a statistically significant sample for each code smell. In particular, we randomly selected a set of 561 code smell removals from our dataset. This represents a 95% statistically significant stratified sample with a 10% confidence interval of the $143,995$ removals detected in our dataset. The stratum of the sample is represented by the eight studied code smells. After sampling, we analyzed every smell-removing commit to inspect two aspects:

— **Commit actions:** The source code modifications that led to the removal of the code smell instance. In this aspect, every code smell type has different theoretical ways to remove it. We inspect the commits to investigate the ways used in practice for concretely removing code smells from the codebase.

— **The commit message:** we looked for any mention of the code smell removal. In this regard, we were aware that developers could refer to the smell without explicitly mentioning it in the commit message. Therefore, we thoroughly read the commit messages to look for implicit mentions of the code smell removal. For this aspect of the analysis, there are no analytical categories. We rather tagged the commit with a boolean to show whether its message mentions the code smell removal or not.

For both aspects, one author analyzed the sample to build the analytical categories and tag every commit. Thereafter, the other authors discussed and double-checked the coding

and assignments. The double-check allows us to verify the relevance of the categories, and reorganize (refine or merge) them, if needed.

4.3.2 Study Results

This section reports on the results of our evolution study.

RQ1: How Frequent Are Code Smell Introductions?

Table 4.6 reports on the number of code smells introduced and the percentage of apps affected.

Code smell	***LIC***	***MIM***	***NLMR***	***HMU***	***UIO***	***UHA***	***IOD***	***UCS***	***All***
#introductions	98,751	72,228	4,198	3,944	514	267	93	18	180,013
%affected-apps	96	85	99	60	36	20	15	2	99

Table 4.6 Numbers of code smell introductions.

The table shows that, in the 324 apps, $180,013$ code smell instances were introduced. This number reflects the widespread of code smells in Android apps. However, it is worth noting that not all code smells are frequently introduced. Indeed, the table shows a significant disparity between the different code smell types. The code smells *LIC* and *MIM* were introduced tens of thousands of times, while *UCS* and *IOD* were only introduced less than 100 times. These results highlight two interesting observations:

— The most frequently introduced code smells (*LIC* and *MIM*) are both about source code entities that should be static for performance optimization.

— The UI-related code smells (*UIO*, *UHA*, and *IOD*) are among the least frequently introduced code smells.

Regarding affected apps, Table 4.6 shows that 99% of apps had at least one code smell introduction in their change history, which again highlights the widespread of the phenomenon. The table also shows that the disparity in introduction frequency is reflected in the percentage of affected apps as frequent code smells tend to affect more apps. However, we observe that having more instances does not always imply affecting more apps. In particular, the code smell *NLMR* is much less present than the code smells *LIC* and *MIM*, $4,198$ *vs.* $98,751$ and $72,228$, respectively. However, it affected more apps, 99% *vs.* 96% and 85%.

To have a clearer vision about these disparities, we reported in Figure 4.5 the diffuseness of code smells within their source code host entities in the studied apps. The figure shows that *NLMR* is the most diffuse code smell. At least 50% of the dataset apps had this code smell in all their activities, $median = 100\%$. *LIC* is also very diffuse. In most of the apps, it affected more than 80% of the inner classes. Code smells that are hosted by views are less diffuse. On average, 15% of the views are infected with *UIO*. As for *IOD*, generally it only

infected less than 10% of the views. Finally, code smells hosted by methods are the least diffuse. *MIM*, *HMU*, *UHA*, and *UCS* are present in less than 3% of the methods. This low diffuseness is not surprising as the number of methods is very high.

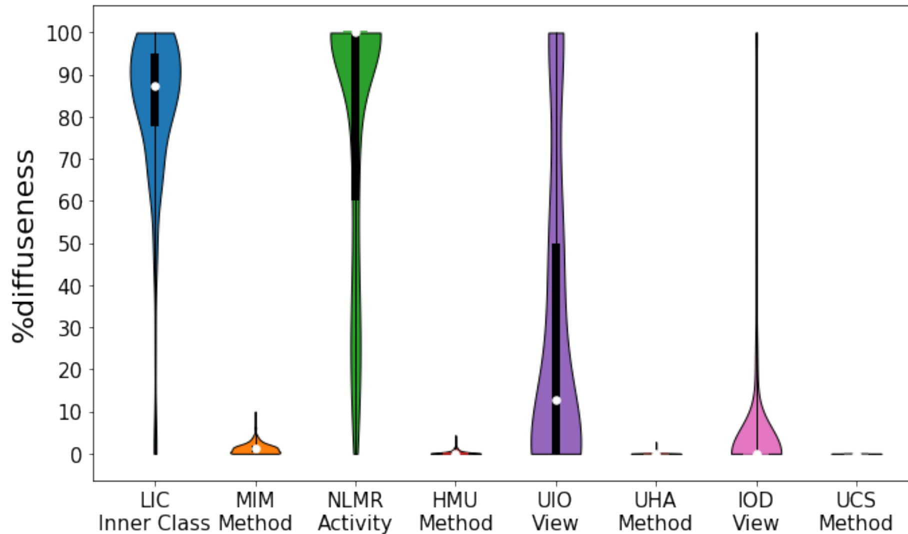

Figure 4.5 Distribution of code smell %diffuseness in studied apps.

These results show that some frequent code smells, like *MIM*, are actually not very diffuse, they only impact a low percentage of their potential host entities. Yet, code smells that seem less frequent like *UIO* and *IOD* are actually more diffuse and affect a higher percentage of entities.

Android code smells are not introduced and diffused equally. *No Low Memory Resolver* and *Leaking Inner Class* are the most diffuse, on average they impact more than 80% of the activities and inner classes, respectively.

RQ2: How Do Releases Impact Code Smell Introductions and Removals?

We generated code smell evolution curves for all the studied apps. These curves can be found in our artifacts [95]. Figure 4.6 shows an example of the generated evolution curves that represents the Seadroid app. The figure highlights the releases in order to show the changes in code smell numbers when approaching releases. From our manual examination of all the evolution curves, we did not observe any tendency of code smell increase or decline immediately before or after releases.

In the remainder of this section, we will further investigate this observation using the metrics *distance to release* and *time before release*.

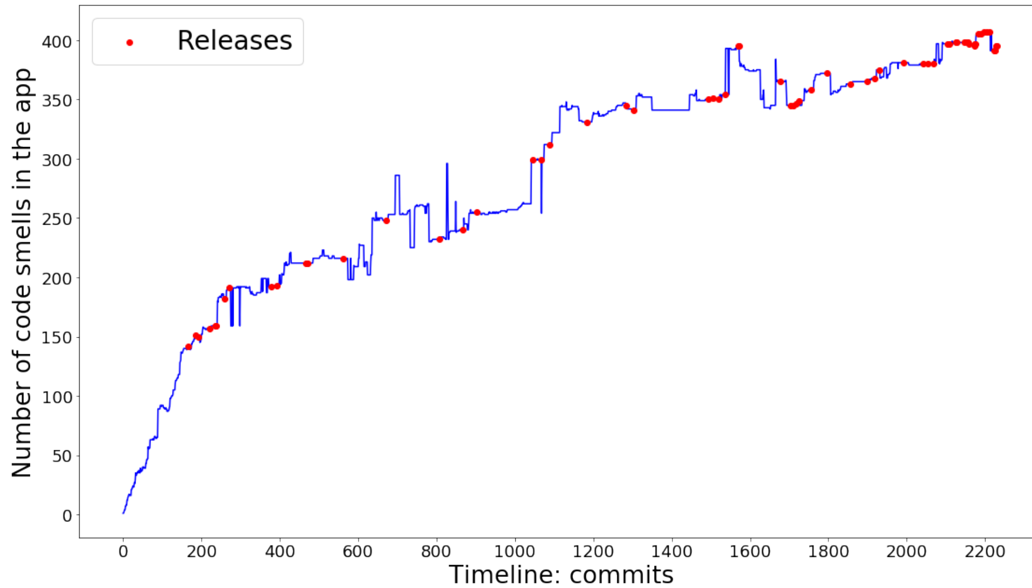

Figure 4.6 The evolution of code smells in the Seafile Android client app.

Distance to release: Figure 4.7 presents two scatter plots that reports on the relationship between the distance from releasing and the number of code smell introductions and removals per commit. The first thing that leaps to the eye is the similarity between the two plots. Code smell introductions and removals are similarly distributed regarding the distance from releasing. We do not notice any time window where the code smell introductions and removals are negatively correlated. We also do not visually observe any correlation between the distance from release and code smell introductions and removals. Indeed, the Spearman's rank correlation coefficients confirm the absence of such correlations.

$$Spearman(\textsf{distance-to-release}, \#\textsf{commit-introductions}) \begin{cases} \rho = 0.04 \\ p\text{-}value < 0.05 \end{cases}$$

$$Spearman(\textsf{distance-to-release}, \#\textsf{commit-removals}) \begin{cases} \rho = 0.01 \\ p\text{-}value < 0.05 \end{cases}$$

The results show that, for both correlations, the *p-value* is below the threshold. Hence, we can consider the computed coefficients as statistically significant. As these correlation coefficients are negligible, we can conclude that there is no monotonic relationship between the distance from releasing and the numbers of introductions and removals per commit.

Time to release: After analyzing the impact of the distance to release, we investigate the impact of the time to release on code smell introductions and removals.

Figure 4.8 depicts the density function of code smell introductions and removals in different timings. First, we observe that code smell introductions and removals are distributed similarly. For

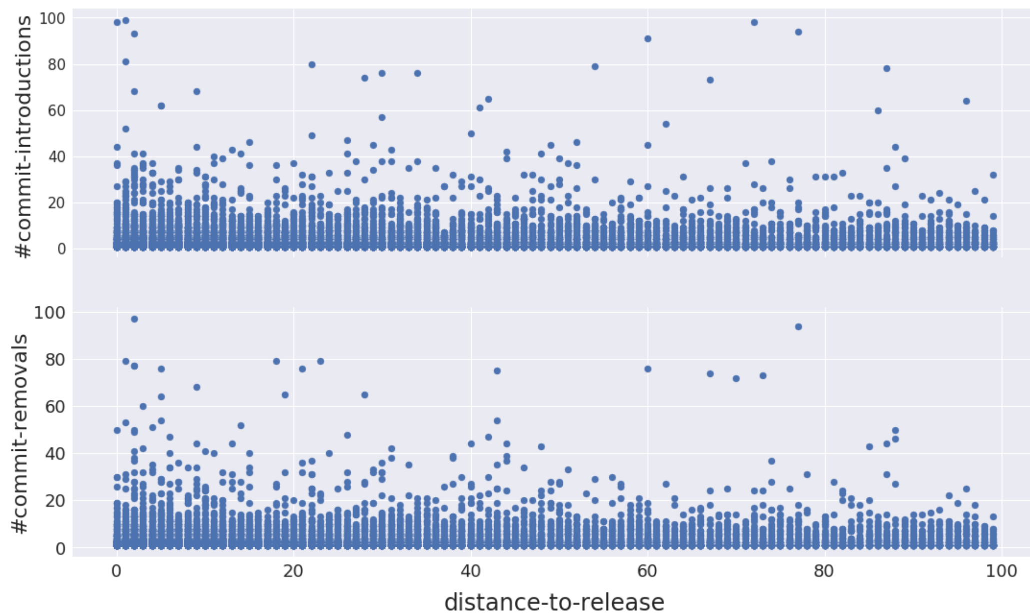

Figure 4.7 The number of code smell introductions and removals per commit in the last 100 commits before release.

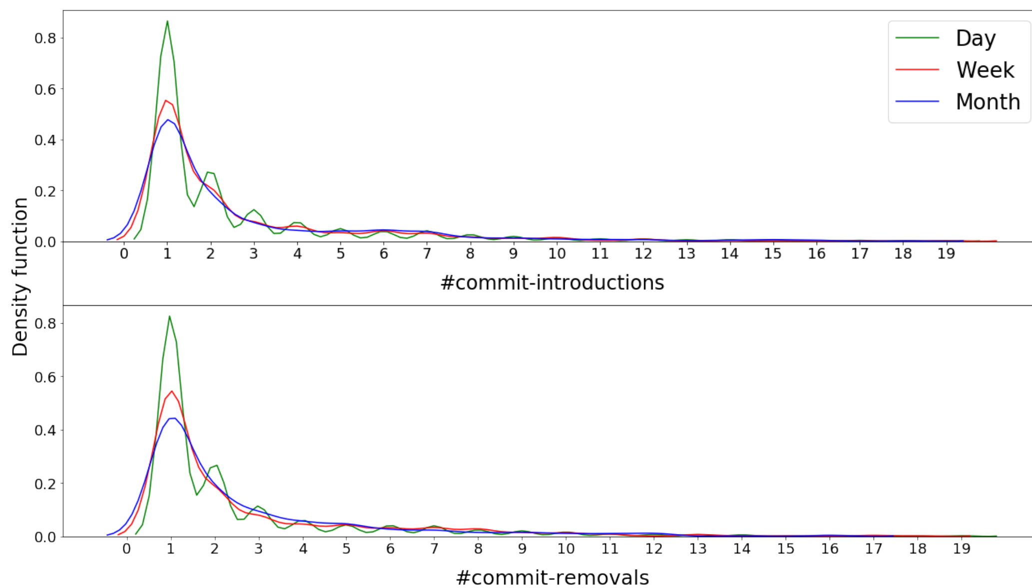

Figure 4.8 The density function of code smell introductions and removals one day, one week, and one month before releasing.

each timing, the density function of code smell introductions and removals are analogous. As for the comparison between code smell introductions performed at different times, we observe that commits performed one day before releasing have a higher probability to only have one code smell introduction. Commits performed one week or one month before release tend also to have around one code smell introduction, but they also have chances to introduce more code smells. This means that code smells authored one day before the release do not necessarily have more code smell introductions. Code smell removals follow exactly the same distribution for every timing. Thus, we can infer that time to release has no visible impact on code smell introductions and removals. To confirm this observation, we compare in Table 4.7 the code smell introductions performed one day, one week, and one month before the release.

	LIC	***MIM***	***NLMR***	***HMU***	***UIO***	***IOD***	***UHA***	***UCS***	***All***
Day	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	$p > 0.01$
Week	$0.01(N)$	$0.05(N)$	$0.08(N)$	$0.20(S)$	$0.18(S)$	$0.10(S)$	–	–	$0.00(N)$
Day	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	–	–	–	$p > 0.01$
Month	$0.02(N)$	$0.04(N)$	$0.08(N)$	$0.02(N)$	–	–	–	–	$0.01(N)$
Week	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	–	–	–	$p > 0.01$
Month	$0.04(N)$	$0.00(N)$	$0.00(N)$	$0.04(N)$	–	–	–	–	$0.01(N)$

Table 4.7 Compare #commit-introductions in commits authored one day, one week, and one month before releasing.

The table results show that for all code smells, there is no significant difference between code smell introductions occurring in different dates before the release ($p - value > 0.01$). The effect size values confirm the results, all the quantified differences are small or negligible.

Similarly, Table 4.7 compares code smell removals in commits authored one day, one week, and one month before the release. The results are similar to the ones observed for code smell introductions. The differences between different commits sets are insignificant ($p\text{-}value > 0.01$) and effect sizes are small or negligible regardless of the code smell type.

These observations suggest that there is no difference between the introduction and removal tendencies in commits authored just before release and those written days or weeks before. It is worth

	LIC	***MIM***	***NLMR***	***HMU***	***UIO***	***IOD***	***UHA***	***UCS***	***All***
Day	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	$p > 0.01$
Week	$0.05(N)$	$0.03(N)$	$0.02(N)$	$0.05(N)$	$0.29(S)$	–	–	–	$0.01(N)$
Day	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	–	–	–	$p > 0.01$
Month	$0.02(N)$	$0.07(N)$	$0.30(S)$	$0.01(N)$	–	–	–	–	$0.03(N)$
Week	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	–	–	–	$p > 0.01$
Month	$0.02(N)$	$0.04(N)$	$0.30(S)$	$0.04(N)$	–	–	–	–	$0.01(N)$

Table 4.8 Compare #commit-removals in commits authored one day, one week, and one month before releasing.

noting that for code smells *UIO*, *IOD*, *UHA*, and *UCS*, the number of instances was in some cases insufficient for performing the statistical tests. Hence, these results are only valid for the most present code smells: *LIC*, *MIM*, *NLMR*, and *HMU*.

Releases do not have an impact on the introductions and removals of Android code smells.

RQ3: How Are Code Smells Removed?

Quantitative analysis: Table 4.9 reports on the number and percentage of removals. The table shows that, on average, 79% of the code smell instances are removed. Looking at every code smell type separately, the removal rate varies between 35% and 93%. *MIM* is the most removed code smell with 93% of its instances were removed along the change history. On the other hand, *UHA* is the least removed code smell with only 35% of its instances removed. The other code smells have a coherent removal percentage: they all had from 60% to 70% of their instances removed.

Code smell	***LIC***	***MIM***	***NLMR***	***HMU***	***UIO***	***IOD***	***UHA***	***UCS***	***All***
#removals	70,654	67,777	2,526	2,509	305	147	66	11	143,995
%removals	71	93	60	63	59	35	70	61	79
#code-removed	67,169	13,809	1,625	1,824	273	123	33	8	84,864
%code-removed	95	20	64	73	90	84	50	73	59

Table 4.9 Number and percentage of code smell removals.

The table also reports on the number and percentage of code smell instances removed within source code removal—*i.e.*, code-removed. The table shows that, overall, 59% of the code smell removals are source code removals. Indeed, for all code smell types except *MIM*, more than 50% of code smell removals are accompanied with the removal of their host entities. *MIM* is the only code smell that is rarely removed with source code removals— %code-removed=20%.

Overall, 79% of code smell instances are removed through the change history. Except for *Member Ignoring Method*, code smell removal generally occurs because of source code removal.

Qualitative analysis: We report on the results of our manual analysis of 561 code smell removals. It is worth noting that, to leverage the replication of this study, all the results of this qualitative analysis are included in our companion artifacts [95]. We found that, for some code smells, the removal fashions were similar, thus we reported them together. Also, for the sake of clarification, we remind for each code smell all the possible ways to remove it from the source code before reporting the ways found in the analyzed sample.

Code smell: Leaking Inner Class (LIC)

Sample size: 96 *LIC* removing commits.
Possible removals: *LIC* can be removed by:
— Making the affected inner class static,

— Removing the affected inner class.

Commit actions: We found that in 98% of the cases, the *LIC* instances are removed because of source code deletion. Specifically, we found that the code smell disappears as the inner classes are removed as a side effect of source code evolution. For instance, a commit from the Seadroid app that fixes bugs also removes unused code that contained a non-static inner class [80]. Hence, the commit has removed a *LIC* instance as a side effect of the bug fixing. This finding explains the high percentage of code-removed found for *LIC* in the quantitative analysis (95%).

We only found one case of a *LIC* removal that was not caused by source code removal. It was a commit that refactored a feature and made an inner class private and static, thus removing a code smell instance [75]. As this commit made diverse other modifications, we could not affirm that the action was an intended code smell refactoring.

Commit message: We did not find any explicit or implicit mention of *LIC* in the messages of the smell-removing commits. Moreover, the messages did not refer specifically to the inner classes removal. Even the unique commit that removed the *LIC* instance with a modification did not mention anything about the matter in its message [75].

Code smell: Member Ignoring Method (MIM)

Sample size: 96 *MIM* removing commits.

Possible removals: *MIM* can be removed by:

— Making the affected method static,

— Introducing code that accesses non-static attributes to the affected method,

— Removing the affected method.

Commit actions: We found that only 15% of *MIM* removals are due to the deletion of the host methods. In most of the cases, *MIM* was rather removed with the introduction of source code. Specifically, when empty and primitive methods are developed—with instructions added inside—they do not correspond to the *MIM* definition anymore, and thus the code smell instances are removed. Also other instances are removed from full methods with the introduction of new instructions that access non-static attributes and methods. Finally, we did not find any case of *MIM* removal that was performed by only making the method static.

Commit message: We did not find any commit message that referred to the removal of *MIM* instances.

Code smell: No Low Memory Resolver (NLMR)

Sample size:93 *NLMR* removing commits.

Possible removals: *NLMR* can be removed by:

— Introducing the method `onLowMemory()` to the activity,

— Removing the affected activity.

Commit actions: We found that 65% of *NLMR* instances are removed with source code deletion. This deletion is caused by major modifications in the code base, like major migrations and important features. For instance, a commit from the Silence app refactored the whole app to start using `Fragment` components [76]. One consequence of these modifications is the deletion of the `SecureSMS` activity, which used to be an instance of *NLMR*.

As for the remainder 35% instances, the removal was due to the conversion of host activities into other components. For example, a commit from the K-9 app converts an activity that was an instance

of *NLMR* into a fragment [77]. As the code smell *NLMR* is about activities, the class as fragment does not correspond to the definition anymore and thus the code smell instance is removed. Other than these two actions, we did not find any other ways of removing *NLMR* instances. In particular, we did not find any case where the method `onLowMemory()` is introduced to refactor the code smell.
Commit message: We did not find any commit message that mentioned the removal of *NLMR* instances. We found one message that mentions that the commit performs memory improvement in two classes [74]. Nonetheless, these improvements were not related to the *NLMR* code smell.

Code smells: Hashmap Usage (HMU), Unsupported Hardware Acceleration (UHA) & Init OnDraw (IOD)

Sample size: 93 instances of *HMU*, 59 instances of *UHA*, and 40 instances of *IOD*.
Possible removals: These code smells can be removed by:

— Removing the instruction that introduced them,
— Removing their host entities.

Commit actions: In the manual analysis of these code smells, we inspected whether the removal was due to the deletion of large code chunks or only the removal of the specific instructions that caused the instance. We found that the instances of *HMU* and *UHA* are in 70% of the cases removed with the deletion of big chunks of code. As for *IOD*, there are equally instances removed with big code deletions as instances removed with only instructions removals. Looking for potential intended refactorings, we carefully examined the cases where the code smells instances are removed at a low granularity level—*i.e.*, instructions. In all the *HMU* and *UHA* instances, we did not find a code smell removal that could represent an intended refactoring. All the instances are removed as a side effect of modifications inside methods that do not specifically target the code smell instruction. Surprisingly, we found that that 22% of *IOD* instances are removed with precise modifications that sound like proper refactoring. Indeed, there are nine *IOD* instances that specifically removed the allocation instruction or extracted it out of the `onDraw()` method. Another element that incites us to describe these modifications as intended is that they removed the Android Lint of *DrawAllocation*. This shows that the developers were aware of removing a code smell instance.
Commit message: Out of the nine potential proper refactorings of *IOD*, we found six commit messages that mentioned the code smell removal. This confirms that the operations are intended refactorings. As for *HMU* and *UHA*, none of the analyzed commit messages mentioned their removal.

Code smells: UI Overdraw (UIO) & Unsuited LRU Cache Size (UCS)

Sample size: 74 instances of *UIO* and 10 instances of *UCS*.
Possible removals: These code smells can be removed by:

— Adding method calls: `clipRect()` or `quickReject()` for *UIO*, and `getMemoryClass()` for *UCS*,
— Removing the instruction that introduced them,
— Removing their host methods.

Commit actions: We found that in more than 70% of the cases, both code smells are mainly removed with big code deletions. The remainder instances were mainly modifications inside methods that implied the deletion of the instructions causing *UIO* and *UCS*. The only exception was one *UIO* instance, which was removed with the introduction of a call to the method `clipRect()`. This modification was the only case that sounded like a proper refactoring.

Commit message: None of the analyzed commit messages mentioned the code smells *UIO* and *UCS*.

> **Android code smells are mainly removed as a side effect of source code evolution activities. *Init OnDraw* and *UI Overdraw* are the only code smells that were subject to apparent refactoring.**

4.3.3 Discussion & Threats

We discuss in this section our results in light of the previous works on code smells and mobile apps. Then, based on this discussion, we highlight the implications of our study. Finally, we present the threats to validity that are specific to our evolution study. Other threats that are common to our three studies will be discussed later in a dedicated section.

Discussion

RQ1: How frequent are code smell introductions?
The results of RQ1 showed that *LIC* and *MIM* are the most frequently introduced code smells. Interestingly, these two code smells can be simply avoided by making inner classes and methods static. That is, their refactoring does not require costly operations and there is no effort needed to avoid them. Hence, their prevalence in the codebase can hardly be explained by time or effort trade-offs. The only explanation to the frequency of such easily avoided code smells would be the developers' ignorance about the issue. This ignorance can be due to the lack of knowledge about the framework and performance aspects or to the developers' disbelief in code smells.

Another interesting observation of RQ1 is that *LIC* and *NLMR* are the most diffuse code smells with a median diffuseness of 90%. This means that, every time a developer builds an activity or an inner class, there is a 90% chance a code smell is introduced. This shows that these code smells must be studied and questioned in depth to understand what are the rationales behind their prevalence and how do developers perceive them.

RQ2: How do releases impact code smell introductions and removals?
The results of RQ2 showed that the pressure of releases do not have an impact on code smell introductions and removals. This shows that code smells are not introduced as a result of rapid coding and time trade-offs. Hence, we can exclude the time pressure from the potential factors causing code smell prevalence in mobile apps. Excluding the time factor strengthens the chances of other factors like developers' ignorance, which aligns with our aforementioned hypothesis.

RQ3: How are code smells removed?
The quantitative and qualitative findings of RQ3 showed that code smells are mainly removed with source code deletion as a side effect of other maintenance activities. Even for *MIM* instances, which are removed with source code introduction, we found that they are removed because the empty and primitive methods are developed with new instructions. While we cannot judge the intentions of a source code modification, most of the observed commits did not reveal signs of intended refactoring. On top of that, the analyzed commit messages did not mention anything about code smell removals.

This again strengthens the hypothesis that developers lack awareness and consideration for mobile code smells. The only exception to this observation was the proper refactoring of *IOD* and *UIO*, which are detected by Android Lint [7]. Interestingly, some of these refactorings explicitly deleted Android Lint suppressions. This shows that developers considered the linter warnings and responded with an intended refactoring. This demonstrates the importance of integrating code smells in built-in tools, like Android Lint.

Implications

Based on the results and this discussion, the implications of our study are the following:

1. Our findings suggest that *developers ignore mobile-specific code smells.* This leads us to hypothesize that *ignorance* and *oversight* are important motives behind the accrual of Android code smells. Hence, we encourage researchers and tool makers should put more efforts in the communication about these code smells and their impact;
2. Our results show that releases do not impact the introductions and removals of Android code smells. This finding shows that releasing pressure, which is the main prioritization aspect, is not a motive of Android code smells;
3. Our findings show that *code smell removals that occur in mobile apps are mostly accidental.* The modifications that lead to code smell removals are only source code deletions and do not present proper refactorings. Unfortunately, this means that we cannot learn refactoring approaches from the removals of the studied code smells. Hence, future studies that intend to learn from the change history to build automated refactoring tools cannot rely on the removals of these code smells as learning examples;
4. The absence of intended code smell removals shows that the motives of code smells are probably ignorance and oversight rather than pragmatism. Indeed, in the case of pragmatism we would expect some documented code smells or mentions of the refactoring in the removal commits;
5. The refactorings observed for *IOD* instances highlight the importance of static analyzers like Android Lint in the improvement of source code quality. These observations encourage researchers and tool makers to invest in the creation and integration of static analyzers with more code smell coverage;
6. Our findings show that some mobile code smells, like *LIC* and *NLMR*, are diffuse to the point where they are introduced nearly every time their host entities are created. This surprisingly high diffuseness should encourage future studies to study developers' perception of these code smells and question their definitions.

Threats to Validity

One possible threat to the assessment of our results could be the accuracy of qualitative analysis and particularly the coding step in RQ3. We used a consensual coding to alleviate this threat and strengthen the reliability of our conclusions. Each code smell instance has been encoded by at least two authors. Initially, every author coded the instance independently to avoid influencing the analysis of other authors. Afterwards, the authors discussed and compared their classifications. To support the credibility of our study, we inductively built our theory from large representative stratified samples of code smell instances—561 for RQ2.

4.3.4 Study Summary

We presented in this section a large-scale empirical study that leverages quantitative and qualitative analyzes to understand the evolution of Android-specific code smells. The main findings of our evolution study are:

Finding 1: Android-specific code smells are not introduced and diffused equally. *No Low Memory Resolver* and *Leaking Inner Class* are the most diffuse, they impact 90% of the activities and inner classes. This surprisingly high diffuseness should encourage future studies to study developers' perception of these code smells and question their definitions.

Finding 2: Releases do not have any impact on the intensity of Android code smell introductions or removals. This finding challenges the common belief about technical debt and shows that code smells are not introduced as a result of rapid coding and time trade-offs.

Finding 3: Android code smells are mainly removed as a side effect of source code evolution activities. *Init OnDraw* and *UI Overdraw* are the only code smells that were subject to apparent refactorings. This finding suggests that future studies that intend to learn from the change history to build automated refactoring tools cannot rely on the removals of the studied Android code smells as learning examples.

Finding 4: Together our results suggest that ignorance and oversight are important motives of Android code smells. On the other hand, other motives such as prioritization and pragmatism do not seem to have a relevant impact on the studied code smells.

4.4 Developers Study

The objective of this study is to analyze the role played by developers in the accrual of Android code smells. This analysis will allow us to assess the importance of developer-oriented motives—*i.e.*, attitudes, ignorance and oversight, in the context of mobile code smells. To conduct this study, we built on the existing studies about mobile-specific code smells. However, none of these studies has analyzed the developers angle. They only approached this topic in their explanatory hypotheses when discussing the possible factors of mobile code smells accumulation. In particular, they hypothesized that some mobile developers who lack experience and knowledge about the framework, are responsible for the accrual of mobile code smells [106]. Our study builds on these hypotheses and addresses the developers role with a large scale analysis. Specifically, we aim at answering the following research questions:

RQ1: How do developers contribute in code smell introductions and removals?

This question is important for quantifying developers interest and awareness about Android code smells. If developers who introduce and remove code smells are isolated and distinct, this would show that the unawareness about code smells is not prevalent and only restrained to some developers and *vice versa.* In this topic, the literature and the common understanding of code smells provide three main hypotheses.

Hypothesis 1 (H1): *Code smell introductions and removals are isolated practices.*

This hypothesis is supported by studies that suggested that the presence of code smells is the result of the actions of only some developers [106].

Hypothesis 2 (H2): *There is an inverse correlation between code smell introductions and removals among developers.*

This hypothesis is motivated by the reasoning that some *bad developers*, who are unaware about code smells, will tend to introduce and never remove them and *vice versa* [106, 109].

Hypothesis 3 (H3): *For each code smell type, there is an inverse correlation between code smell introductions and removals among developers.*

This hypothesis aligns with the previous one, but goes deeper by considering the knowledge about every code smell separately.

RQ2: How does seniority impact developers contribution?
This question allows us to investigate whether code smell unawareness is caused by inexperience. This information is necessary to identify the developers who should be targeted in priority when designing tools and documentations about code smells. The suggested hypotheses for this question are the following.

Hypothesis 4 (H4): *Newcomers tend to introduce more and remove less code smells.*

This hypothesis relies on the conventional wisdom that inexperienced developers who are new to the project are bad contributors.

Hypothesis 5 (H5): *Reputable developers tend to introduce less and remove more code smells.*

Reputable developers have a status in the community and they are considered as good contributors [60]. This hypothesis suggests that, as good developers, they should introduce less and remove more code smells.

In the following subsections, we will explain how we leveraged the data extracted by Sniffer to answer these research questions. Afterwards, we report and discuss the study results, then we conclude with a study summary that highlights the main findings.

4.4.1 Data Analysis

Table 4.10 reports on a list of metrics that we defined for the data analysis. We defined the rate metrics, namely %introductions and %removals, to compare the developers' role in terms of code smells regardless of the size of their contributions. Indeed, comparing the number of code smells introduced by a developer who has 100 commits and another who has only 10 commits can be biased. The developer with more commits has more chances to introduce and remove code smells. Similarly, the metric %contribution allows to relate the number of commits to the project size. For instance, a contribution of 10 commits in a project of 20 commits is totally different from a contribution of 10 commits in a project of 100 commits. For the definition of the metrics newcomer and regular, we followed the approaches proposed in previous studies. As a matter of fact, Tufano *et al.* [197] used the threshold of three commits to distinguish newcomers from regular developers. We present in the following how we used these metrics in order to answer our research questions.

RQ1: How Do Developers Contribute in Code Smell Introductions and Removals?

The analysis of this research question consisted in testing its three hypotheses.

Metric	***Description***
#commits	(int): the number of authored commits.
#introductions	(int): the number of code smells introduced.
#removals	(int): the number of code smells removed.
%introductions	(float): introduction rate, *i.e.*, the average number of smells introduced per commit: *#introductions/#commits*.
%removals	(float): removal rate, *i.e.*, the average number of smells removed per commit: *#removals/#commits*.
%contribution	(float): the size of the developer's contribution compared to the project size: *#commits/#project size*.
#followers	(int): the number of followers.
introducer	(boolean): true if the developer introduced at least one code smell.
remover	(boolean): true if the developer removed at least one code smell.
neutral	(boolean): true if the developer did not introduce or remove any code smell.
newcomer	(boolean): true if the developer has less than three commits in the project.
regular	(boolean): true if the developer has more than three commits in the project.
infamous	(boolean): true if the developer has less followers than 75% of the population.
famous	(boolean): true if the developer has more followers than 75% of the population.

Table 4.10 Developer metrics.

***H1**: Code smell introductions and removals are isolated practices.*
This hypothesis suggests that only an isolated part of Android developers are responsible for code smell introductions and removals. To test this hypothesis, we investigated the distribution of developers regarding code smell introductions and removals. Specifically, we computed the numbers and percentages of developers that have the attributes introducer, remover, and neutral. This allowed us to split the developers into different groups and have an insight about the general tendency of code smell introductions and removals among Android developers.

***H2**: There is an inverse correlation between code smell introductions and removals among developers.*
We took a two-step approach to test this hypothesis. First, we measured the relationship between the metrics #introductions and #removals using Spearman's rank correlation coefficient. Spearman is a non-parametric measure that assesses how well the relationship between two variables can be described using a monotonic function. This measure is adequate for our analysis as it does not require the normality of the variables and does not assess the linearity.
As the metrics #introductions and #removals can be biased by the number of commits, we proceeded to the second step with different metrics. Specifically, we used Spearman to measure the correlation between %introductions and %removals.

***H3**: For each code smell type, there is an inverse correlation between code smell introductions and removals among developers.*
This hypothesis states that at the code smell level, the introduction and removal are inversely correlated. This means that developers who are unaware about a specific type of code smells, will tend to introduce it and never remove it. Similarly, developers who are aware about the code smell type will tend to remove it and never introduce it. To test this hypothesis, we computed the metrics %introductions and %removals for each type of code smell, *e.g.*, %introductions_LIC, %removals_LIC for the *LIC* code smell. As we study eight code smells, this process gave us 16 metrics. Afterwards, we computed the correlation between the two metrics of each code smell type, *e.g.*, $correlation(\%introductions_MIM, \%removals_MIM)$. The correlation coefficient used is again Spearman's rank.

RQ2: How Does Seniority Impact Developers Contribution?

The objective of this research question is to assess the impact of the developer's seniority on code smell introductions and removals. We measured the seniority with (i) the experience in the development project and (ii) the reputation in the community.

***H4**: Newcomers tend to introduce more and remove less code smells.*
To assess this hypothesis, we followed two approaches. First, we split the developers into two groups using the metrics newcomer and regular. Then, we compared the distribution of the metrics %introductions and %removals in the two groups. In particular, we used a two-tailed Mann-Whitney U test [174], with a 99% confidence level, to check if the distributions of introductions and removals are identical in the two sets. To quantify the effect size of the presumed difference, we used Cliff's δ [170], which is suitable for ordinal data and it makes no assumptions of a particular distribution [170].

In the second approach, we relied on the metric %contribution, which reflects the involvement of a developer in a project. We assessed the impact of this involvement on code smell introductions

and removals. Concretely, we computed the Spearman's rank of the metric %contribution with %introductions and %removals respectively.

***H5**: Reputable developers tend to introduce less and remove more code smells.*
In this hypothesis, we opted for the number of followers as an estimation of the developer's reputation. The number of followers is a signal of status in the community, developers with many followers are treated as celebrities [60].

To assess the hypothesis, we first used the metrics famous and infamous to split the developers into two groups. Then, we compared the tendency of code smell introductions and removals in the two groups using the metrics %introductions and %removals. To perform this comparison we relied on the quartiles, Mann-Whitney U, and Cliff's δ.

As a second approach, we measured directly the correlation between the reputation and code smell introductions and removals. For this, we computed the Spearman's rank of the metric #followers with %introductions and %removals, respectively.

4.4.2 Study Results

In this section, we report and analyze the results of our experiments with the aim of answering our research questions.

RQ1: How Do Developers Contribute in Code Smell Introductions and Removals?

H1: Table 4.12 reports on the distribution of developers in terms of code smell introductions and removals. We observe that only 35% of the developers are implicated in code smell introductions. The remaining 65% did not introduce any code smell instances. The distribution in terms of code smell removals is similar. Only 31% of the developers participated in code smell removals. We also observe that 28% of the developers are implicated in both code smell introductions and removals. This intersection between introducers and removers is highlighted in Figure 4.9. The figure shows that there is an important intersection between developers that introduce and remove code smells. This means that developers who introduce and remove code smells are mainly the same. Another interesting observation from Table 4.12 is that 61% of developers are neutral, they did not introduce or remove any code smell instances.

	introducer	***remover***	***introducer and remover***	***neutral***	***All***
# developers	1590	1414	1269	2790	4525
% developers	35	31	28	61	100

Table 4.12 Distribution of developers according to code smell introductions and removals.

To further investigate the nature of these groups, we compare in Figure 4.10 the distribution of their number of commits. The distribution is illustrated with a density function, which allows to highlight how many commits are authored by the developers in the three groups.

The first thing that leaps to the eye is that neutral developers are concentrated around a low number of commits. Most of them have only authored less than 5 commits. This means that developers

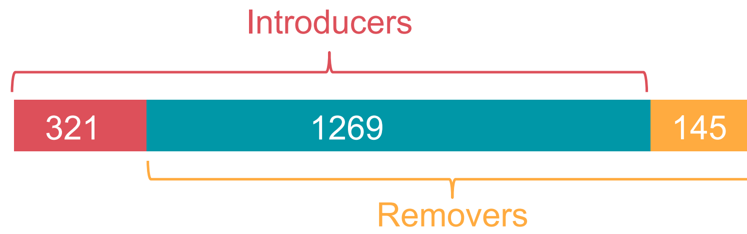

Figure 4.9 Intersection between introducers and removers.

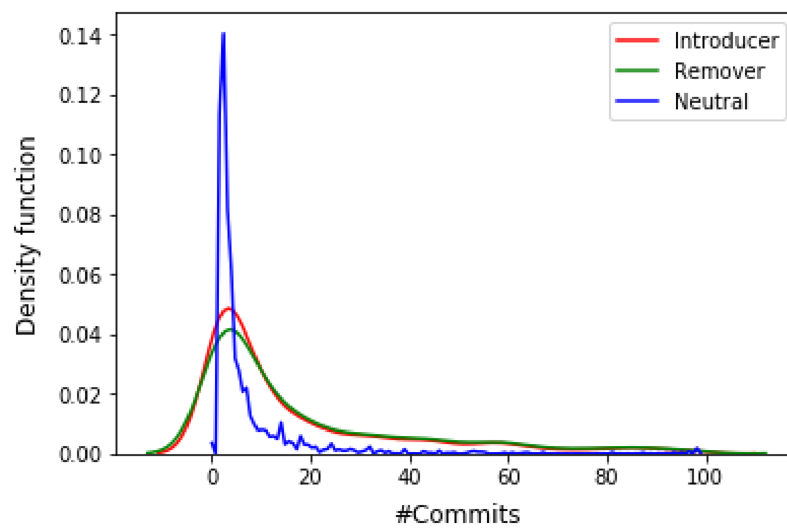

Figure 4.10 The distribution of the number of commits among the studied developers.

who did not introduce or remove any code smells are developers who did not commit regularly. On the other side, introducers and removers are less concentrated around the low values. Developers in these two groups have authored more commits. It is also worth noting that introducer and remover groups are similarly distributed which aligns with Figure 4.9 where they manifested a considerable intersection.

To sum up, even if Table 4.12 shows that introducers and removers are only one third of the total contributors. They actually represent most of the regular developers. Developers who did not participate in code smell introductions and removals have very few commits. Based on these findings, we can assert that code smell introduction and removal are not isolated practices. Thus, we can reject Hypothesis 1.

Developers who introduce and remove code smells are mainly the same. Developers who did not introduce or remove code smells are not regular contributors.

H2: The results of the correlation between the numbers of code smell introductions and removals among developers are the following.

$$Spearman(\#introductions, \#removals) \begin{cases} r = 0.94 \\ p\text{-}value < 0.01 \end{cases}$$

The $p-value$ shows that the computed r coefficient is statistically significant. The coefficient value, 0.94, shows that there is a strong positive relationship between the numbers of introductions and removals performed by a developer. That is, the more a developer tends to introduce code smells, the more she will remove. To further investigate this finding, we evaluated the relationship with other measures. The results of the correlation between the introduction and removal rates are the following.

$$Spearman(\%introductions, \%removals) \begin{cases} r = 0.77 \\ p\text{-}value < 0.01 \end{cases}$$

As the $p-value$ is lower than the threshold 0.01, the computed coefficient is statistically significant. The coefficient value, 0.77, shows that the introduction and removal rates are positively correlated. This means that the more a developer introduces code smells per commit, the more she removes as well. Hence, even independently of the number of commits, code smell introduction and removal are positively correlated among developers. Based on these results, we can reject Hypothesis 2.

The more a developer introduces code smells, the more she tends to remove them.

H3: Table 4.13 reports the correlation coefficients between the %introductions and %removals per code smell type. For the eight code smells, the $p-values$ are under the threshold, thus the results are statistically significant. The correlation coefficients depict that the introduction

Smell	***LIC***	***MIM***	***NLMR***	***HMU***	***UIO***	***UHA***	***IOD***	***UCS***
Correlation	0.9	1.0	0.8	0.6	0.6	0.8	0.5	1.0
p-value				< 0.01				

Table 4.13 Correlation between introductions and removals per code smell type.

and removal rates are positively correlated. Indeed, for *LIC*, *MIM*, *NLMR*, *UHA* and *UCS*, the introduction and removal rates are strongly correlated with a coefficient over 0.8. This means that developers who introduced these types of code smells also removed them and *vice versa*. For *HMU*, *UIO*, and *IOD*, the correlation is between 0.5 and 0.6. This means that there is an average positive correlation between the introduction and removal rates of these code smells among developers. These findings show that developers remove the same type of code smells that they introduce. There is no code smell where developers who removed instances did not tend to introduce new ones. Hence, we can affirm that there is no inverse correlation between introductions and removals at the code smell level. Thus, we can reject Hypothesis 3.

The more a developer introduces a type of code smells, the more she removes it.

RQ2: How Does Seniority Impact Developers Contribution?

H4: Table 4.14 compares the introduction and removal rates among newcomers and regular contributors. First, we observe that, using the criterion defined in the study design, our dataset contains 3375 newcomers and 1793 regular contributors. It is worth noting that the sum of these two groups is above the total number of developers: 4,525. This due to the involvement of many of developers in several studied projects. Thus, these developers did not have one global contribution in this context, but multiple contributions.

		size	***Q1***	***Med***	***Q3***	***p-value***	***Cliff***
%introductions	***newcomer***	3375	0	0	0	< 0.01	$0.46(M) \downarrow$
	regular	1793	0	0.21	0.76		
%removals	***newcomer***	3375	0	0	0	< 0.01	$0.44(M) \downarrow$
	regular	1793	0	0.11	0.50		

Table 4.14 Comparison between newcomers and regular developers in terms of introduction and removal rates.

Regarding introductions, we observe that newcomers tend to introduce very few code smells per commit, *Q1, Median*, and *Q3* are null. This means that at least 75% of the newcomers

did not introduce any code smell per commit. On the other side, regular contributors tend to introduce more code smells. Indeed, 50% of them introduce more than 0.21 code smells per commit and 25% of them are above 0.76 introductions. This superiority is confirmed by the significant *p-value* computed by Mann-Whitney U and the Cliff's δ value 0.46. The latter shows that there is a medium difference in the introduction rates in favor of regular contributors.

As for removals, the table shows that newcomers do not tend to remove code smells. *Q1, Median*, and *Q3* values are null. Regular contributors tend to remove more code smells. 50% of them remove more than 0.11 code smells per commit and 75% of them have a removal rate above 0.50. Mann-Whitney U and Cliff's δ values confirm this observation. Removal rates among regular contributors are superior to the rates of newcomers.

To push forward this analysis, we computed the correlation between the experience in the project and code smell introductions and removals. The results are the following:

$$Spearman(\%contribution, \%introductions) \begin{cases} r = 0.27 \\ p\text{-}value < 0.01 \end{cases}$$

$$Spearman(\%contribution, \%removals) \begin{cases} r = 0.24 \\ p\text{-}value < 0.01 \end{cases}$$

In both correlations, the $p-value$ indicates that the results are statistically significant. For the introduction, the r value, 0.27 shows that there is a weak positive correlation between the experience in the project and the introduction rate. Likewise for code smell removals, the r value, 0.24, indicates that the correlation between experience and removals is positive and weak. This means that experience does not have an impact on the developer's role in terms of code smells. Developers with few contributions, like newcomers, do not forcibly introduce more or remove less code smells. Hence, based on the comparison and correlation results, we can reject Hypothesis 4.

Newcomers are not particularly bad contributors. They do not introduce or remove more code smells than regular developers.

H5: Figure 4.11 and Table 4.15 summarize the distribution of the number of followers among the studied developers.

	Q1	***Median***	***Q3***	***IQR***
#followers	1	8	25	24

Table 4.15 The numbers of followers among the studied developers.

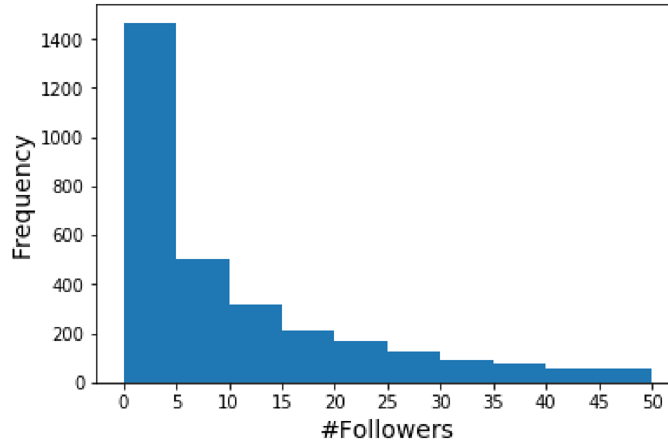

Figure 4.11 The distribution of the number of followers among studied developers.

We observe that the majority of the studied developers have less than 25 followers. Based on the values of Table 4.15, we defined our thresholds for splitting developers into two groups:

— infamous: developers with ≤ 1 followers,

— famous: developers with ≥ 25 followers.

Table 4.16 compares the code smell introduction and removal rates among infamous and famous developers.

		size	***Q1***	***Med***	***Q3***	***p-value***	***Cliff***
%introductions	***infamous***	945	0	0	0	< 0.01	$0.18(S)\downarrow$
	famous	906	0	0	0.47		
%removals	***infamous***	945	0	0	0	< 0.01	$0.19(S)\downarrow$
	famous	906	0	0	0.29		

Table 4.16 Comparison between infamous and famous developers in term of code smell introduction and removal rates.

For the introduction rate, we observe that in both groups, the introduction per commit is around zero for most of the developers. Indeed, *Q1* and *Median* values are null for both infamous and famous developers. The difference arises in the third quartile values, *Q3*. The latter shows that, while 25% of famous developers introduce more than 0.47 code smells per commit, infamous developers are mostly around 0 introductions per commit. This difference is confirmed by the Mann-Whitney U results, as the *p-value* is below the confidence threshold. The Cliff's δ value, 0.18, shows that there is a small difference between the two groups in favor of famous developers. This means that famous developers have a slightly higher introduction rate than the infamous ones.

We observe similar results for removal rates. For both groups, 50% of the developers remove 0 code smells per commit—*Median=0*. However, the *Q3* value shows that 25% of famous developers introduce more than 0.29 code smells per commit while the infamous ones are mostly around 0 introductions. Mann-Whitney U and Cliff's δ results confirm this observation. The *p-value* is below the threshold, which means that there is a statistically significant difference between the removal rates in both groups. The Cliff's δ shows that, while small 0.19, this difference is in favor of famous developers. This means that famous developers tend to remove slightly more code smells than infamous ones.

To consolidate this analysis, we computed the correlation between the developer's reputation and her code smell introductions and removals. The results are the following:

$$Spearman(\#followers, \%introductions) \begin{cases} r = 0.14 \\ p\text{-}value < 0.01 \end{cases}$$

$$Spearman(\#followers, \%removals) \begin{cases} r = 0.15 \\ p\text{-}value < 0.01 \end{cases}$$

As the $p - values$ are below the threshold, we can consider the computed coefficients as statistically significant. The correlation coefficient between the number of followers and the introduction rate, 0.14, suggests that there is a very weak positive correlation between them. Likewise, the correlation between the number of followers and the removal rates is weak, $r = 0.15$. This means that the reputation does not have an impact on the code smell activity. That is, famous developers are not significantly better contributors in terms of code smells. Based on these findings, we can reject Hypothesis 5.

Reputable developers are not particularly good contributors in terms of code smells.

4.4.3 Discussion

We conducted this study with the hypothesis that some developers are responsible for the accrual of mobile code smells. In light of our results, we were able to reject this hypothesis [106]. Even if the numbers show that 61% of the developers are neutral, this percentage is biased by the huge number of developers who had very few commits. Indeed, developers who committed regularly are more present and had more chances to introduce or remove code smells. This means that the phenomenon of code smells is not limited to a small group of developers. The results of *RQ2* pushed this idea forward. Besides not being a small isolated group, developers who contributed to code smells are not necessarily newcomers or infamous in their communities. This implies that there is no isolated group of junior developers to blame for code smells, the responsibility is rather shared among all regular

contributors. Our analysis also showed that developers who introduce and remove code smells are not distinct, they are mainly the same group. On top of that, among these developers, the introduction and removal tendencies are correlated, even at the smell level. This implies that code smell removals are not intended. Indeed, a developer who consciously removes a code smell, would not tend to introduce it elsewhere. This behavior appears more like accidental introductions and removals of code smells. Therefore, we challenge the suggestion that Android developers *"perform refactorings to improve the code structure and enforce solid programming principles"* [109]. In fact, similar observations were previously made about the refactoring of OO code smells. Studies showed that, sometimes, code smells refactoring is due to maintenance activities and cannot be considered as a proper refactoring [161, 197].

Based on this analysis, we suggest that the studied mobile code smells are mainly driven by ignorance and oversight instead of individual attitudes. The results also suggest that the introductions of Android code smells are not strategic or tactic choices [191]. That is, developers do not deliberately accept this technical debt in favor of other objectives, like productivity. Rather, they introduce the debt with no leverage—*i.e.*, they do not gain anything in return. To refine and further explore this hypothesis, we encourage future research works to study the self-admittance [59] of Android code smells.

The observed oversight also incites us to question the importance of mobile code smells and the efficiency of the communication channels used by framework builders. Effectively, the unawareness of most developers may be due to the unimportance of these code smells in practice. This questioning is particularly relevant as the code smells are performance-oriented. If these code smells effectively impacted apps in practice, the performance degradation would have attracted developers awareness. This hypothesis has the following implications on researchers:

— Studies should reevaluate the importance and impact of mobile code smells in practice. We highly encourage future works to perform empirical studies on real life scenarios and real mobile apps to assess the effective impact of these code smells on performance;
— The definition of code smells should not only be a top-down process. That is, mobile code smells, which are for now only based on documentation, should also consider the viewpoint of developers. We invite researchers to involve developers in the evaluation of these code smells and the definition of future ones. The developers' opinion would allow us to better evaluate the importance and severity of these code smells in practice.

The other possible explanation of the prevalent unawareness towards mobile code smells is the vague communication about them. It is unlikely that all developers carelessly consider the framework documentation, especially when it comes to performance. Hence, we suggest for framework builders to:

— Highlight and stress the importance and impact of code smells;

- Use more pervasive channels to raise developers awareness about code smells. In particular, framework built-in tools, like linters, should emphasize the importance and impact of code smells;
- Researchers and tool makers should work on just-in-time detection and refactoring tools that run as soon as the change is committed to prevent the accumulation of code smells.

4.4.4 Study Summary

We presented in this section an empirical investigation of the role played by developers in the accrual of mobile-specific code smells. This study resulted in several findings that challenge the conventional wisdom:
Finding 1: The accrual of Android code smells is not the responsibility of an isolated group of developers. Most regular developers participated in the introduction of code smells.
Finding 2: There are no distinct groups of code smell *introducers* and *removers*. Developers who introduce and remove code smells are mostly the same.
Finding 3: While newcomers are not to blame for the accrual of mobile code smells, reputable developers are not necessarily good contributors.
Finding 4: Together our results suggest that Android code smells are driven by developers ignorance and oversight rather than individual attitudes.

4.5 Survival Study

This study investigates the impact of processes and prioritization on the survival of Android code smells. Besides identifying code smell motives, this investigation is important for understanding the extent and importance of Android code smells. Indeed, a code smell that persists in the software history for long periods can be more critical than code smells that disappear in only a few commits. To address this study, we formulated the following research questions:

- **RQ1:** For how long do Android-specific code smells survive in the codebase?
- **RQ2:** What are the factors that impact the survival of Android-specific code smells? We investigate in this question the impact of project size, releasing practices, and code smell properties on the survival chances of code smell instances. The effect of the project size interests us as the common sense suggests that code smells persist longer in big and complex projects. Regarding releases, Android apps are known for having more frequent releases and updates [138]. Our aim is to investigate the impact of this particularity on code smell survival. As for code smell properties, we are particularly interested in the impact of Android Lint [9]. Android Lint is the mainstream linter—*i.e.*, static analyzer, for Android. It is integrated and activated by default in the official

IDE Android Studio. Our objective is to inspect whether the code smells detected and prioritized by Android Lint are removed faster from the codebase.

In the following subsections, we will explain how we leveraged the data extracted by SNIFFER to answer these research questions. Afterwards, we report and discuss the study results, then we conclude with a study summary that highlights the main findings.

4.5.1 Data Analysis

Element	*Metric*	*Description*
Project size	#Commits	(int): the number of commits in the project.
	#Developers	(int): the number of developers contributing to the project.
	#Classes	(int): the number of classes in the project.
Release	#Releases	(int): the number of releases in the project.
	Cycle	(int): the average number of days between the project releases.
Code smell	Linted	(boolean): true if the code smell is detectable by Android Lint.
	Priority	[1-10]: this metric is only valid for *Linted* code smells. It presents the priority given to the code smell in Android Lint.
	Granularity	(categories): the level of granularity of the code smell's host entity, namely: inner class, method, or class level.

Table 4.17 Metrics for *RQ2*.

We explain in this sub-section our data analysis approach for answering our two research questions.

RQ1: For How Long Do Android-Specific Code Smells Survive in the Codebase?

To answer this research question, we relied on the statistical technique of survival analysis, *a.k.a.* time-to-event analysis [118]. The technique analyzes the expected duration of time until one or more events happen (*e.g.*, a death in biological organisms or a failure in mechanical systems). This technique suits well our study since we are interested in the duration of time until a code smell is removed from the codebase. Hence, in the context of our study, the subjects are code smell instances and the event of interest is their removal from the codebase. As for the time-to-event, it refers to the lifetime between the instance introduction and its removal.

The survival function $S(t)$ is defined as:

$$S(t) = Probability(T > t)$$

Where T is a random lifetime from the population under study. That is, the survival function defines the probability for a subject (a code smell in our case) for surviving past time t. The survival function has the following properties [61]:

1. $0 \leq S(t) \leq 1$;
2. $S(t)$ is a non-increasing function of t;
3. $S(0) = 1$ and $\lim_{t \to \infty} S(t) \to 0$.

Interestingly, survival analysis models take into account two data types:

— **Complete data**: this represents subjects where the event of interest was already observed. In our study, this refers to code smells that were removed from the codebase;

— **Censored data**: this represents subjects that left during the observation period and the event of interest was not observed for them. In our case, this refers to code smells that were not removed during the analyzed commits.

To measure the lifetime of code smell instances, we relied on two metrics:

— #Days: The number of days between the commit that introduces the code smell and the one that removes it;

— #Effective commits: The number of commits between the code smell introduction and removal. For this metric, we only counted commits that performed modifications in the file of the code smell. This fine-grained-analysis allows us to exclude irrelevant commits that did not effectively impact the code smell host file.

Using these metrics as a lifetime measure, we built the survival function for each code smell type. Specifically, we used the non-parametric estimator Kaplan Meier [36] to generate survival models as a function of #Days and #Effective commits, respectively.

To push forward the analysis, we report on the survival curves per code smell type. This allows us to compare the survival of the eight studied Android code smells and investigate their differences. Moreover, we checked the statistical significance of these differences using the Log Rank test [105]. This non-parametric test is appropriate for comparing survival distributions when the population includes censored data. We used a 95% confidence interval—*i.e.*, $\alpha = 0.95$, with a null hypothesis assuming that the survival curves are the same.

For implementing this analysis, we respected the outline defined by Syer *et al.* [186]. We used the Python package Lifelines [61]. We used `KaplanMeierFitter` with the option `ci_show=False` to omit confidence interval in the curves. Also, we used `pairwise_logrank_test` with `use_bonferroni=True` to apply the Bonferroni [201] correction and counteract the problem of multiple comparisons.

RQ2: What Are the Factors That Impact the Survival of Android-Specific Code Smells?

In this research question, we investigated the impact of project size, releasing practices, and code smell properties on the survival chances of code smell instances. For this purpose, we defined the metrics presented in Table 4.17.

Project size: We specifically analyzed the size in terms of #Commits, #Developers, and #Classes.

Releasing practices: We analyzed the impact of the metrics #Releases and Cycle on the survival rates. To assure the relevance of the studied apps for a release inspection, we used the same set of apps used for release analysis in Section 4.3. We recall that this set contains 156 that use releases during all their change history.

Code smell: Unlike project and release metrics, the values of code smell metrics are determined by the code smell type. Specifically, the values of Linted and Priority are defined by Android Lint [9]. Android Lint is able to detect four of our code smells: namely *LIC*, *HMU*, *UIO*, and *IOD*. It attributes a priority level from 1 to 10 that describes the importance of these code smells, where 1 is the least important and 10 is the most important. As for Granularity, it is determined by the code smell host entities that are presented in Table 4.1. In this respect, the entities `Activity` and `View` both represent the granularity level of class. For more clarification, we report in Table 4.19 the values of these three metrics per code smell type.

Code smell	***Linted***	***Priority***	***Granularity***
LIC	True	6	Inner Class
MIM	False	–	Method
NLMR	False	–	Class
HMU	True	4	Method
UIO	True	3	Class
UHA	False	–	Class
IOD	True	9	Class
UCS	False	–	Method

Table 4.19 The values of *Linted*, *Popularity*, and *Granularity* per code smell type.

To investigate the impact of the defined metrics on code smells survival, we used survival regression and stratified survival analysis.

Survival regression: We used this approach to analyze the impact of numerical metrics—*i.e.*, #Commits, #Developers, #Classes, #Releases, and Cycle. The survival regression allows us to regress different covariates against the lifetime variable. The most popular regression technique is Cox's proportional hazard model [55]. Cox's model has three statistical assumptions:

1. *proportional hazards*: the effects of covariates upon survival are constant over time;
2. *linear relationship*: the covariates make a linear contribution to the model;
3. *independence*: the covariates are independent variables.

We assessed these assumptions on our dataset and found that our metrics do not respect the proportional hazard assumption. That is, the impact of our numerical metrics on survival are not constant and evolve over time. Thus, we opted for the alternative technique of Aalen's additive model, which allows time-varying covariate effects [1]. The Aalen's model defines the hazard function by:

$$\lambda(t|x) = b_0(t) + b_1(t) * x_1 + ... + b_n(t) * x_n$$

Where x_i refers to the studied covariates and $b_i(t)$ is a function that defines the regression coefficient over time. In our study, the covariates are the numerical metrics and the regression coefficients describe their impact on the survival of code smell instances. For the interpretation of these coefficients, we should note that the hazard function can also be defined as $\lambda(t) = 1 - S(t)$. This means that an increase in the hazard function implies a decrease in the survival one. Consequently, the metrics that have a positive hazard regression coefficient, systematically decrease the survival chances of the studied code smells and *vice versa*.

For implementation, we used `AalenAdditiveFitter` from Lifelines package [61]. This implementation estimates $\int b(t)\,\mathrm{d}t$ instead of $b(t)$. We used the function `smoothed_hazards_()` with the parameter $bandwidth = 100$ to get the actual hazard coefficients ($b(t)$).

Stratified survival analysis: We followed this approach to analyze the impact of categorical metrics, namely Linted, Priority, and Granularity. To assess the impact of these metrics, we relied on the same technique used for *RQ1*, Kaplan Meier. Specifically, we computed the survival function for each metric and category. Then, we compared the survivals among the different categories of each metric using the Log Rank test. For instance, we computed two survival curves for the metric Linted, one for Linted=True and another for Linted=False. Then, we compared the two curves using the Log Rank test.

4.5.2 Study Results

RQ1: For How Long Do Android-Specific Code Smells Survive in the Codebase?

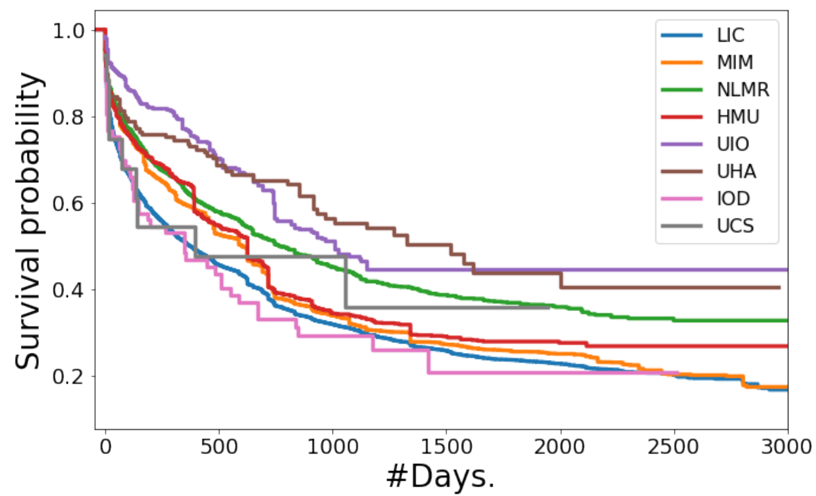

Figure 4.12 Survival curves in days.

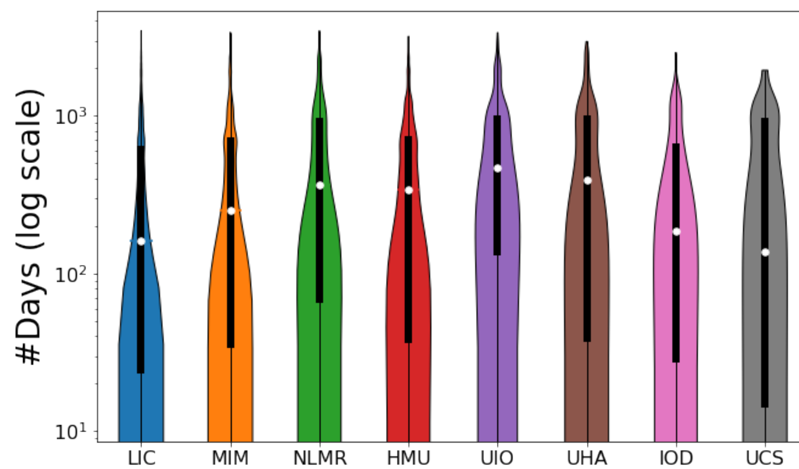

Figure 4.13 Code smell lifetime in terms of days.

Survival in days: Figures 4.12 and 4.13 show the results of Kaplan Meier analysis. To explicit our results, we also present the survival distribution in Table 4.20. Figure 4.12 illustrates the survival curves of the eight studied code smells in terms of days. The figure indicates that the survival probabilities differ considerably depending on the code smell type. Indeed, 3,000 days after introduction, code smells like *LIC* and *MIM* have almost no chances to be alive, whereas code smells—like *UHA* and *UIO*—still have 40% chances to be alive. This disparity is confirmed by Figure 4.13 and Table 4.20. We can observe that, on average,

	LIC	***MIM***	***NLMR***	***HMU***	***UIO***	***IOD***	***UHA***	***UCS***	***All***
Q1	41	117	136	111	400	300	60	16	52
Med	370	602	765	625	1,007	1,516	347	395	441
Q3	1,536	1,978	∞	∞	∞	∞	1,422	∞	1,691

Table 4.20 Kaplan Meier survival in days.

after 441 days, 50% of all the code smells are still present in the codebase. However, this median value increases significantly among the code smells *MIM*, *NLMR*, *HMU*, *UIO*, and *UHA*. 50% of these code smells are still alive after more than 600 days. We assessed the statistical significance of these differences with the Log Rank test. While we only report the significant results, the full summary of the pairwise Log Rank test between the 8 code smells is available with our artifacts [96]. Overall, the results allowed to reject the null hypothesis assuming that the survival curves of the 8 code smells are the same—*i.e.*, $p - value < 0.05$. We confirmed that the survival order shown in Table 4.20 is significant. More specifically, we found the following:

— *UIO* and *UHA* are the code smells that survive the longest. It takes around three years to remove 50% of their instances ($median > 1000\ days$);

— *NLMR*, *HMU*, and *MIM* also have a high survival tendency with an average of two years ($median > 600\ days$);

— The least surviving code smells are *IOD*, *LIC*, and *UCS*. 50% of their instances are removed after only one year ($median > 300\ days$).

> 50% **of the instances of Android-specific code smells stay alive in the codebase for more than** 441 **days.**

Survival in effective commits: Figures 4.14 and 4.15 reports on the results of Kaplan Meier analysis with effective commits. The figures show that in the 100 effective commits that follow the code smell introduction, most of the instances are removed. We also observe that there are slight differences between the survival tendencies of the eight code smells. Indeed, Table 4.21 and the Log Rank test results show the following:

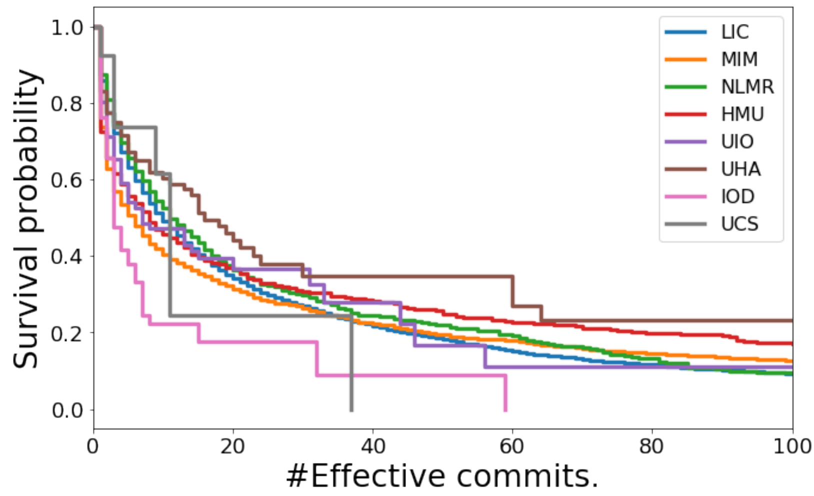

Figure 4.14 Survival curves in effective commits.

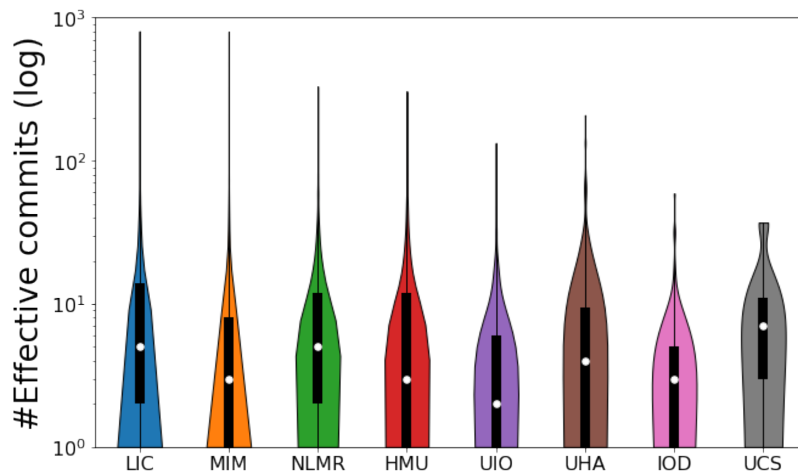

Figure 4.15 Code smell lifetime in terms of effective commits.

	LIC	***MIM***	***NLMR***	***HMU***	***UIO***	***IOD***	***UHA***	***UCS***	***All***
Q1	3	1	3	1	2	3	2	3	3
Med	10	6	11	8	7	16	3	11	9
Q3	34	33	38	50	44	64	7	11	34

Table 4.21 Kaplan Meier survival in effective commits.

— *UHA* is the longest surviving code smell. It takes 16 effective commits to remove 50% of its instances and even after 64 effective commits 25% of its instances are still alive ($median = 16$ and $Q3 = 64$);
— *IOD* is the least surviving code smell. 75% of its instances are removed after only seven commits on the host file ($Q3 = 7$);
— The other code smell types only have slight survival differences. On average, their instances survive for nine effective commits and 75% of them disappear after 30 commits.

> 75% **of the instances of Android-specific code smells are removed after** 34 **commits on the host file.**

RQ2: What Are the Factors That Impact the Survival of Android-Specific Code Smells?

Survival regression: Figure 4.16 shows the results of Aalen's additive analysis for the project metrics #Commits, #Developers, and #Classes. It is worth noting that the three metrics are not statistically independent in our dataset. Thus, we measured their regression coefficients separately to avoid the correlation bias. From the figure, the first thing that leaps to the eye is that the three metrics have a positive impact on the hazard of code smells. Indeed, the curve values are positive during all the code smell lifetime. Thus, the project size negatively impacts the survival possibility, which means the bigger the project is, the less the code smells survive in the codebase. Overall, the positive impact on the hazard is more present in the first days after the code smell introduction. Indeed, the curves of the three metrics have their highest values in the first 500 days. In the days after, the hazard rates drop significantly to stabilize after 1,500 days. This aligns with the survival curves observed in *RQ1* where the average survival was less than 500 days.

Figure 4.17 shows the results of Aalen's additive analysis for the release metrics #Releases and Cycle. These two metrics are not highly correlated in our dataset $|r| < 0.3$. Hence, we can analyze their impact simultaneously with a multivariate regression.

We observe from Figure 4.17 that the hazard coefficients for the two variables are positive along the code smell lifetime. This shows that the two metrics have a positive impact on the hazard function. That is, increasing the number of releases and the releasing cycle tend to shorten the code smell lifetimes. Interestingly, Figure 4.17 also indicates that the cycle has more impact on the survival rates than the number of releases. Accordingly with the units, this means that adding one day to the average releasing cycle has more impact on survival rates than adding one release to the whole project.

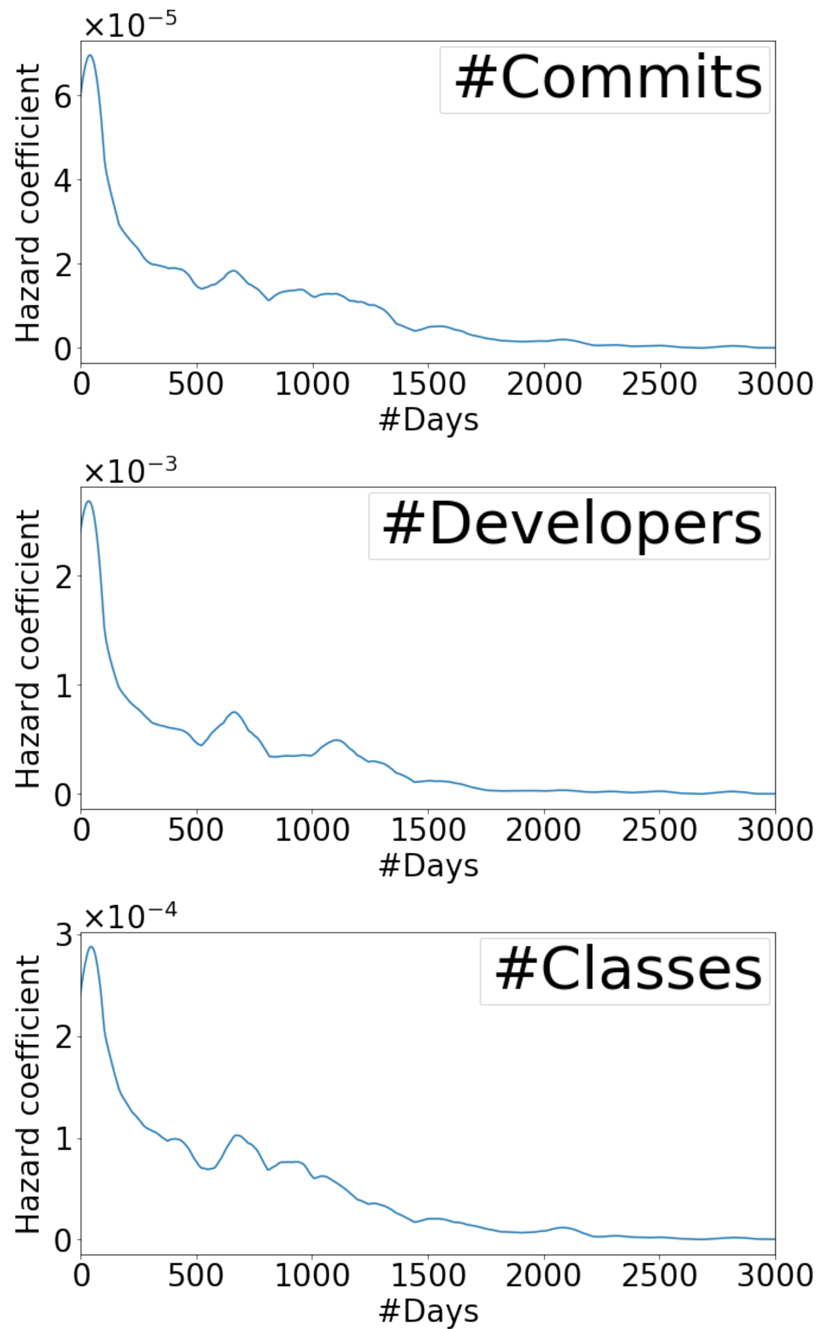

Figure 4.16 The estimated hazard coefficients for #Commits, #Developers, and #Classes.

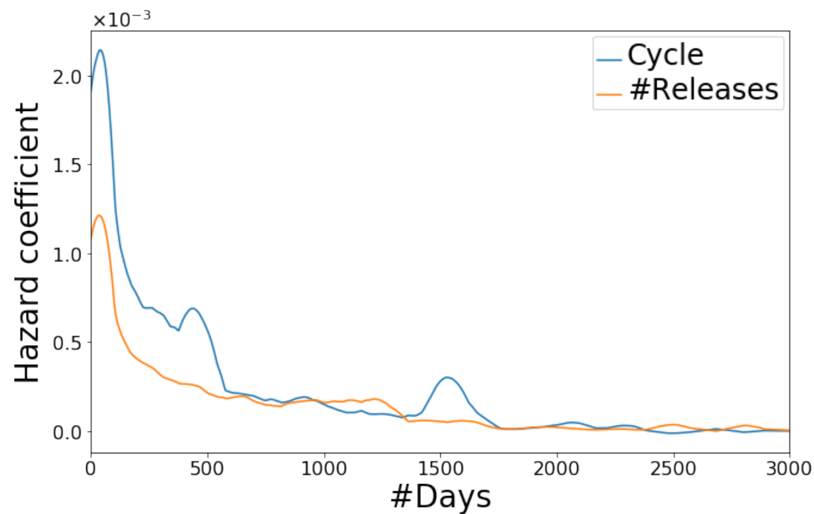

Figure 4.17 The hazard coefficient of #Releases and Cycle.

Code smells disappear faster in projects that are bigger in terms of commits, developers, and classes. Projects with longer releasing cycles and more releases also manifest shorter code smell lifetimes.

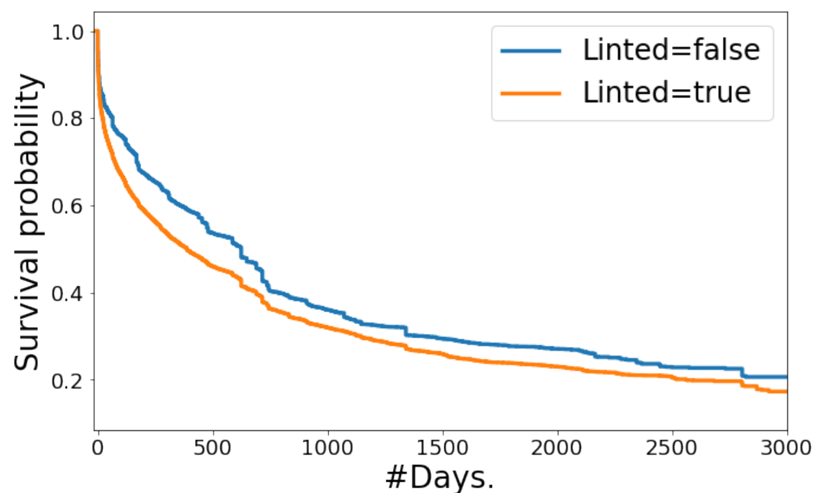

Figure 4.18 The impact of the presence in the linter on survival.

Stratified survival analysis: Figure 4.18 compares the survival curves for code smells based on the metric *Linted*. The figure shows that the survival curve of *Linted* code smells is always under the curve of other code smells. This means that code smells that are present in Android Lint have less survival chances than other code smells. The Log Rank test confirmed

the statistical significance of this observation with $p-value < 0.05$. This allows us to reject the null hypothesis assuming that the two curves are the same.

Code smells detected by Android Lint are removed from the codebase faster than other types of code smells.

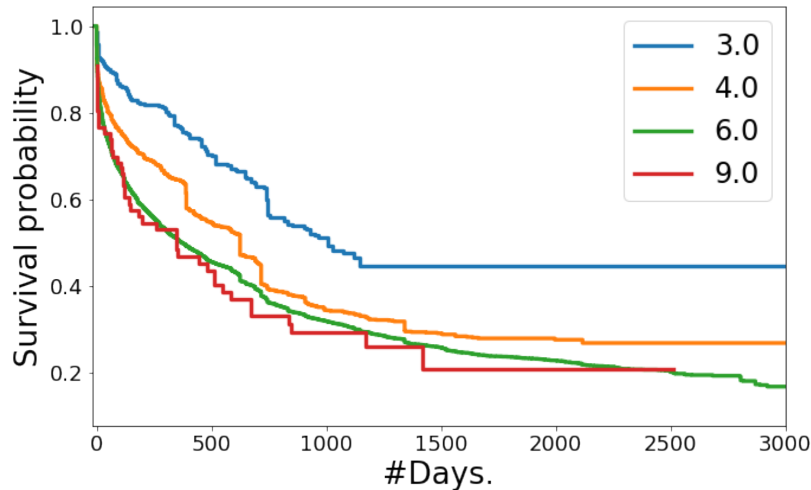

Figure 4.19 The impact of linter priority on code smell survival.

Figure 4.19 shows the results of Kaplan Meier analysis stratified with the metric Priority. It compares the survival curves for code smells depending on their priority on Android Lint. Overall, we observe that code smells with higher priorities have less survival chances. Indeed, the code smells with the highest priority—nine—have the lowest survival chances. Their survival curve is always below other curves. On the other hand, code smells with the lowest priority—three—survive longer than other code smells. The Log Rank test results showed that all these differences are statistically significant with a 95% confidence interval. Hence, we can confirm that the more a code smell is prioritized, the less its survival chances are.

Code smells that are prioritized by Android Lint have less survival chances. They are removed faster than other code smells.

Figure 4.20 compares the survival curves of code smells depending on their granularity. The figure shows that code smells hosted by inner classes have the least survival chances, they disappear before other code smells. We also observe that code smells hosted by classes survive way more than code smells hosted by methods or inner classes. The Log Rank resulted in $p-values < 0.005$ for all the pairwise comparisons between the three curves. As a result, we can reject the hypotheses assuming that the survival curves of different granularities are the same.

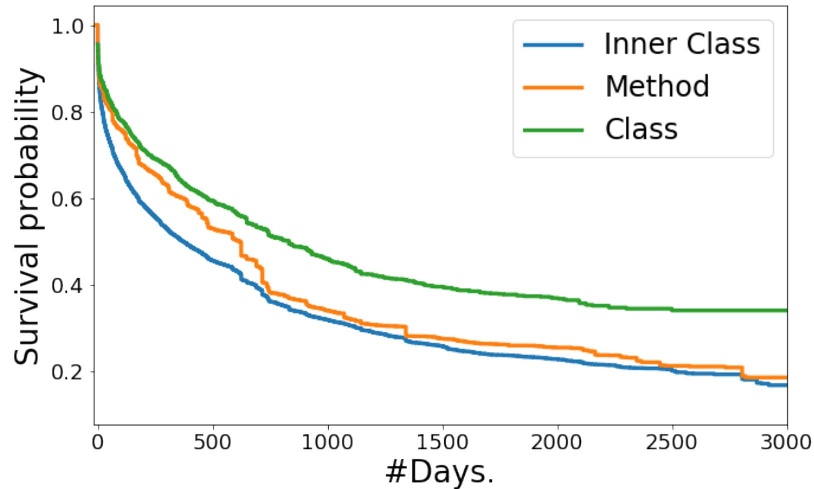

Figure 4.20 The impact of granularity on code smell survival.

The code smell granularity has an important impact on its survival chances. Code smells hosted by classes survive significantly more than smells hosted by methods and inner classes.

4.5.3 Discussion

Compared to the code smell longevity reported in previous studies [27, 117, 161, 197], Android code smells have a significantly shorter lifespan. Effectively, the study of Tufano *et al.* [197] reported that the median survival of OO code smells in Android apps is over 1,000 days and 1,000 effective commits. In our study, the only Android code smells that reach this average longevity are *UIO* and *UHA*. All other Android code smells have a median survival of 400 days. This shows that Android code smells are by far less persistent than OO code smells. This can be due to the nature of Android code smells, which lead them to be introduced and removed faster. Indeed, Android code smells are rather low granularity instances and they can be removed accidentally in general. For instance, a code smell like HashMap Usage is caused by the instantiation of a `HashMap` collection. Therefore, any instance of this code smell can be accidentally introduced or removed by only modifying one instruction. On the other side, OO code smells tend to be more sophisticated and may require extensive modifications to refactor them. Interestingly, our results of the granularity analysis align with this hypothesis. Indeed, the comparison demonstrated that code smells that are hosted by entities of higher granularity (class) remain longer in the source code. This confirms that, when the code smell is at a low granularity level, it becomes easier to remove and thus disappear faster from the source code.

Another surprising finding is that Android code smells have a very short lifespan in terms of effective commits. This finding highlights a huge gap between the number of days and

effective commits needed to remove the code smell instances, 400 days *vs.* nine effective commits. This contrast can be due to the size and complexity of Android apps. Indeed, in large software systems, the modifications are diffused across different software components. Hence, a code smell can survive in such systems for years if its host entity is not frequently modified. In this way, the survival in terms of days becomes much higher than the survival in terms of effective commits.

Our survival regression analysis showed that Android code smells disappear faster in projects that are bigger in terms of commits, developers, and classes. This observation can be due to the activity and dynamic of these projects. Indeed, bigger projects that have more contributors are potentially more active and thus are subject to more modifications. This dynamism makes these projects more prone to code smell removals. As for releases, our results showed that code smells survive less in projects that adopt more releases and longer releasing cycles. The two observations may seem contradictory as we would expect a high number of releases to imply shorter cycles. However, in practice our results showed that the two variables do not have a strong negative correlation. This means that to fasten code smell removals, developers should release frequently without going as far as to shrink releasing cycles.

Finally, our results showed that code smells detected and prioritized by Android Lint are removed faster than other code smells. This can be a direct or an indirect effect of Android Lint. Indeed, when the linter is adopted in the development process, it directly detects code smells from the source code and encourages developers to remove them. Moreover, the linter can also make developers aware of the code smells by means of communication. In this way, even developers who disabled the linter in their *Integrated Development Environment* (IDE) may still be aware about these code smells.

To sum up, our work takes a step forward in the study of mobile code smells and opens up perspectives for new research directions. Concretely, it has the following implications:

- Our results showed that Android code smells survive less in projects with more releases. This means that the high releasing frequency in the mobile ecosystem [138] does not necessarily promote bad development practices. Hence, we can exclude the releasing prioritization from the potential motives of Android code smells;
- The short lifespans of Android code smells incite us to hypothesize that their removal is not intended and rather accidental. We invite future works to investigate this hypothesis and inspect the actions leading to the removal of Android code smells;
- We showed that linters can help in improving the quality of mobile apps. We encourage the community to work on approaches and tools that allow the detection and refactoring of larger sets of mobile code smells;
- Our results align with previous findings about the efficiency of integrating software quality tools in the development workflow [47, 114, 171]. We invite future works to invest in the integration of tools like aDoctor and Paprika in the IDE;

— We confirm the importance of priority and severity indicators in static analysis tools [35, 47, 114]. We encourage tool makers to adopt such measures in their software quality tools;
— The observed importance of the linter and its configuration suggest that the processes have an impact on the presence of Android code smells, and thus they can be considered as a relevant motive.

4.5.4 Study Summary

We presented in this section an empirical study that investigated the lifespan of Android-specific code smells in the change history. By analyzing $180k$ instances of code smells, we found that:

Finding 1: While in terms of time Android code smells can remain in the codebase for years before being removed, it only takes 34 effective commits to remove 75% of them.

Finding 2: Android code smells disappear faster in bigger projects with more commits, developers, and classes.

Finding 3: The high releasing frequency in the mobile ecosystem does not necessarily promote long code smell lifespans.

Finding 4: Android code smells that are detected and prioritized by Android Lint tend to disappear before other code smell types.

Finding 5: Together our findings suggest that releasing prioritization is not a relevant technical debt motive in the case of Android code smells. On the other hand, processes such as adopted tools and their configuration show an important impact on the studied code smells.

4.6 Common Threats to Validity

Internal Validity: The main threat to our internal validity could be an imprecise detection of code smell introductions and removals. This imprecision is relevant in situations where code smells are introduced and removed gradually, or when the change history is not accurately tracked. However, this study only considered objective code smells that can be introduced or removed in a single commit. As for the history tracking, we relied on SNIFFER, which considers both branches and renamings. On top of that, the validation showed that SNIFFER accurately tracks code smell introductions and removals (F1-score $= \{0.97, 0.96\}$).

External Validity: The main threat to external validity is the representativeness of the results. We used a sample of 324 open-source Android apps with $255k$ commits and $180k$ code smell instances. It would have been preferable to consider also closed-source apps to build a more diverse dataset. However, we did not have access to any proprietary software that can serve this study. We also encourage future studies to consider other datasets of open-source

apps to extend this study [71, 121]. Another possible threat is that our study only concerns eight Android-specific code smells. Without a further investigation, these results should not be generalized to other code smells or development frameworks. We therefore encourage future studies to replicate our work on other datasets and with different code smells and mobile platforms.

Construct Validity: In our case, the construct validity might be affected by the gap between the concept of code smells in theory and the detection performed by PAPRIKA. However, the definitions adopted by PAPRIKA have been validated by Android developers with a precision of (0.88) and a recall of (0.93) [106, 108]. On top of that, our validation showed that PAPRIKA is also effective in the dataset under study.

Conclusion Validity: The main threat to the conclusion validity in this study is the validity of the statistical tests applied. We alleviated this threat by applying a set of commonly-accepted tests employed in the empirical software engineering community [136]. We paid attention not to violate the assumptions of the performed statistical tests. We are also using non-parametric tests that do not require making assumptions about the distribution of the data. Finally, we did not make conclusions that cannot be validated with the presented results.

4.7 Summary

We presented in this chapter four main contributions of our thesis. First, we proposed SNIFFER, a novel toolkit that tracks the full history of Android-specific code smells. Our toolkit tackles many issues raised by the Git mining community, like branch and renaming tracking [120]. Hence, we encourage future mining studies to use SNIFFER, which is open-source and heavily tested and documented [104]. We also provided a comprehensible replication packages that can be reused in future studies [93, 95, 96]. These packages include the artifacts of our study, the results of our qualitative analysis, and a database storing the history of $180k$ mobile-specific code smell instances.

We presented three empirical studies about Android-specific code smells in the change history. These studies covered eight Android code smells, 324 Android apps, $255k$ commits, and contributions from $4,525$ developers. The results of these studies shed the light on many potential rationales behind the accrual of Android code smells. In particular, we showed that:

- Pragmatism and prioritization choices are not relevant motives of technical debt in the case of Android code smells;
- Individual attitudes of Android developers are not the reason behind the prevalence of code smells. The problem is rather caused by ignorance and oversight, which seem to be common among developers;

— The processes adopted by development teams—*e.g.*, the use of tools and their configuration can help in limiting the presence of code smells.

These findings suggest that the main reason behind Android code smells are related to the developers knowledge and the processes that they adopt. For these reasons, we dedicate our next study to:

— an investigation of developers' understanding of Android code smells,
— and an analysis of the role of linting tools in coping with Android code smells.

Chapter 5

Developers' Perception of Mobile Code Smells

Following our study of the change history, we suggested that the main reason behind the accrual of code smells is developers' ignorance and oversight. To further investigate this hypothesis, we decided to follow a qualitative method and ask developers about their perception of mobile code smells. The most intuitive way to implement this method would be to design surveys and interviews to get the desired information from developers. However, asking developers directly about mobile-specific code smells may have the following flaws:

— If developers are completely unaware about code smells, we will need to introduce them to the topic before surveying them. However, our introduction may lead to biases in the qualitative study;

— If developers have some knowledge about code smells, the latter may be an abstract concept for them. Indeed, developers may read about code smells and have theoretical insights about it without binding it to the practices that they follow when coding. Hence, it could be difficult for them to give faithful and practical judgment of code smells.

To avoid these threats, we needed a study design where we could catch developers' perception of code smells without asking them direct questions about code smells. In this regard, one possible proxy between developers and code smells could be a linter— *a.k.a.* static analyzer. The linter is the tool that detects bad development practices and presents them to the developer. It is generally integrated in the IDE so developers can interact with it and fix the flagged issues. Hence, linters can be considered as a proxy between bad practices (code smells in our case) and developers. Besides, as developers actively use linters during the development process, they can have genuine and clear thoughts about them. Thus, if we ask developers about a mobile-specific linter, we can get insights about mobile code smells while avoiding the aforementioned threats.

Surveying developers about a mobile-specific linter is also important since one of our conclusions in the previous chapter was that tools like the linter can have a significant impact on code smells. Moreover, as shown in our literature review, linters have always been proposed as a solution for detecting code smells and limiting their presence. This applies to both research and development communities. For instance, Android Studio, the official IDE for Android development, integrates a linter—called Android Lint—that detects performance bad practices, which are the equivalent of our studied code smells. However, despite the availability of these linters, Android developers do not rely heavily on linters to deal with performance concerns. A survey conducted by Linarez *et al.* [125] with 485 Android developers showed that, when using tools to manage performance, most developers rely on profilers and framework tools, and only five participants reported using a linter. In order to confirm this phenomenon and lay the foundation for our study, we published an online survey and asked Android developers about their linter usage. All the details of this survey and its results are available in Appendix B. The results of this survey reported that only 51 % of Android Lint users rely on it for performance purposes. Given that performance checks are enabled by default in Android Lint, it is important to question how Android developers perceive the usefulness of linters when it comes to performance. It is also important to highlight the benefits of adopting linters for performance issues in order to encourage developers to use them.

To address the two aforementioned issues, (i) developers perception of code smells and (ii) the usage of linters for performance, we conduct a qualitative study about the usage of Android Lint by developers. We focus on Android Lint since it is the most used linter for Android, and it detects a large set of performance bad practices. Our study aims to specifically answer the following research questions:

— **RQ 1:** Why do Android developers use linters for performance purposes?

— **RQ 2:** What are the constraints of using linters for performance purposes?

On top of that, our study design gives us the opportunity to investigate other aspects about the fashions that allow mobile developers to achieve the eventual benefits of linters for performance. Thus, we also investigate the question:

— **RQ 3:** How do Android developers use linters for performance purposes?

The remainder of this chapter is organized as follows. We start with a description of our methodology in Section 5.1. Section 5.2 reports and discusses the results of our qualitative study. We identify in Section 5.3 the implications of our results, and in Section 5.4 its limitations, before concluding in Section 5.5.

This chapter is a revised version of a paper published in the proceedings of ASE'18 [98].

5.1 Methodology

Our objective is to investigate with an open mind the benefits and limitations of using a linter for performance purposes in Android. Therefore, we follow a qualitative research approach [56] based on classic Grounded Theory concepts [3]. With this approach, we aim to discover new ideas from data instead of testing pre-designed research questions. Specifically, we conducted interviews with 14 experienced Android developers. The interview design and the selected participants are presented in Sections 5.1.1 and 5.1.2, respectively. Afterwards, we transcribed and analyzed the interviews, as explained in Section 5.1.3.

5.1.1 Interviews

As commonly done in empirical software engineering research, we designed semi-structured interviews. This kind of interview consists of a list of starter questions with potential follow up questions. We followed the advice given by Hove and Anda *et al.* [111] to design and conduct the interviews. In particular, we paid careful attention to explaining the objectives of the interviews. We explained that interviews are not judgmental and we incited the participants to talk freely. We asked open-questions, such as: *"Why do you think that Lint is useful for performance purposes?"*, and we asked for details whenever possible. We designed the interview with basically 12 questions, and depending on participants' replies, we asked additional questions to explore interesting topics. The main questions are the following:

1. Why do you use the linter to deal with performance concerns?
2. What are the benefits that you perceived with this usage?
3. How do you use the linter to deal with performance concerns?
4. How do you configure the linter?
5. Do you use it individually or in a team?
6. In a collaborative project, how do you configure the linter to optimize the app performance?
7. Do you integrate the linter in the build or CI? Why?
8. Do you change the priority or severity of performance checks? Why?
9. Do you ignore or suppress performance checks sometimes? Why?
10. Are there any performance checks that you consider irrelevant? Why?
11. Do you write your own performance checks? Why?
12. In your opinion, what are the constraints of using the linter for performance purposes?

With the permission of interviewees, the interviews were recorded and they lasted from 18 to 47 minutes, with an average duration of 30 minutes. We performed two interviews face-to-face, and the 12 others were performed with an online call. One participant was not able to participate in the call and instead received a list of questions via email and gave written answers.

5.1.2 Participants

Our objective was to select experienced Android developers who use the linter for performance. We did not want to exclude *open-source software* (OSS) developers, nor developers working on commercial projects. For that purpose, we relied on many channels to contact potential participants.

GitHub: First, we selected the most popular Android apps on GitHub relying on the number of stars. Afterwards, we selected the projects that use the linter by looking for configuration files—*e.g.*, `lint.xml`, or configurations in Gradle files. Then, we manually analyzed the Top-100 projects to identify the developers who actively work with the linter. We only found 41 developers who contributed to the linter configuration. As it is complex to guess if developers are motivated to use the linter for performance only from the configuration files, we contacted these developers to ask them if they use the linter for performance or not. Out of 41 mails sent, we received 18 answers—*i.e.*, a response rate of 43 %. 13 developers answered that they use the linter for performance, and five others answered negatively. We replied to the 13 developers to explain the objectives of our interview and invite them to participate. We received six acceptances and two rejects, the other developers did not answer.

Forums and meetups: To select commercial Android developers, we sent forms in developer forums [168]. In the forms, we explicitly explained that we are looking for Android developers who use the linter for performance. We received six answers from forums, one of them was irrelevant because the developer did not have a real experience with Android Lint. We also communicated the same message in Android development meetups. From meetups, we selected three persons who satisfied our criteria.

Overall, this selection process resulted in 14 participants. After conducting 14 interviews, we considered that the collected information was enough to provide us with theoretical saturation [88]—*i.e.*, all concepts in the theory are well-developed. Thus, we did not perform a second batch of selection and interviews.

To keep the anonymity of the 14 selected participants, we refer to them with code names. We also omit all the personal information like company or project names. Table 5.1 shows the participants' codes, their experience in Android and Android Lint in terms of years of professional development, and the types of projects they work on. It is worth mentioning that, with the term "commercial", we refer to projects that are developed in an industrial environment and are not based on an open-source community. Also, we found that all the developers selected from GitHub were also involved in commercial projects, and two developers spotted from forums were involved in both commercial and OSS projects.

Table 5.1 shows that, out of 14 participants, 11 have more than 5 years of experience in Android. Compared to the age of the Android framework (8 years), this experience is quite

Participant	***Android experience***	***Lint experience***	***Project type***
P1	8	5	OSS & Commercial
P2	8	4	OSS & Commercial
P3	8	4	Commercial
P4	8	5	OSS & Commercial
P5	8	4	OSS & Commercial
P6	8	5	OSS & Commercial
P7	6	4	OSS & Commercial
P8	5	3	OSS & Commercial
P9	4	4	Commercial
P10	5	3	Commercial
P11	2	2	Commercial
P12	5	4	Commercial
P13	4	1	Commercial
P14	8	4	OSS & Commercial

Table 5.1 Partcipants' code name, years of experience in Android and Android Lint, and the type of projects they work on

strong. As for Android Lint, which has been introduced with Android Studio in 2013, 10 of our participants have more than four years of experience in using it.

5.1.3 Analysis

We carefully followed the analytical strategy presented by *Schmidt et al.* [173], which is well adapted for semi-structured interviews. This strategy has proved itself in the context of research approaches that postulate an open kind of theoretical prior understanding, but do not reject explicit pre-assumptions [173]. Before proceeding to the analysis, we transcribed the interviews recordings into texts using a denaturalism approach. The denaturalism approach allows us to focus on informational content, while still working for a *"full and faithful transcription"* [148]. In what follows, we show how we adopted the analytical strategy steps.

Form Material-Oriented Analytical Categories

In this step, we define the semantic categories that interest us. In our case, we investigate the motivation and arguments of Android developers who use the linter for performance purposes, and the constraints of such usage. Therefore, our categories are initially the following two topics:

— Why do Android developers use linters for performance purposes?

— What are the constraints of using linters for performance purposes?

After our discussions with participants, we noticed that an additional category is highlighted by developers, namely:

— How do Android developers use linters for performance purposes?

We found this additional topic enlightening, thus we included it in our analytical categories. Once the categories are set, we read and analyzed each interview to determine which categories it includes. In the analysis, we do not only consider answers to our questions, but also how developers use the terms and which aspects they supplement or omit. After this analysis, we can supplement or correct our analytical categories again.

Assemble the Analytical Categories into a Guide for Coding

We assemble the categories into an analytical guide, and for each category different versions are formulated. The versions stand for different subcategories identified in the interviews in reference to one of the categories. In the following steps, the analytical guide can be tested and validated. Indeed, the categories and their versions may be refined, made more distinctive or completely omitted from the coding guide.

Code the Material

At this stage, we read the interviews and try to relate each passage to a category and a variant formulation. As we focus on labeling the passages, we may omit special features and individual details of each interview. Yet, these details will be analyzed and highlighted in the last step. To strengthen the reliability of our conclusions, we use a consensual coding. Therefore, each interview is coded by at least two authors. Initially every author codes the interview independently, afterwards, the authors discuss and compare their classification. In case of discrepancies, the authors attempt to negotiate a consensual solution.

Categories	Versions (subcategories)	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
Benefits	Learn about the framework	x	x	x				x		x	x	x			
	Anticipate performance bottlenecks	x	x			x	x				x			x	
	The linter is easy to use		x		x		x		x			x			x
	Develop the performance culture	x				x		x			x				
	Save time		x	x						x					
	Contribute to credibility and reputation	x					x	x							
	Raise awareness of app performance			x		x									
	Ease performance debugging									x					
Fashions	Integrate from scratch		x			x				x	x	x		x	
	Target milestones	x				x	x	x							
	Adopt performance sprints					x							x		
	Improve performance incrementally									x					
	Support individual development	x		x		x			x	x		x		x	
	Check performance rules in a team		x		x		x	x		x	x		x		x
	Prioritize performance aspects		x	x		x	x			x				x	
Constraints	Static analysis is not suitable for performance	x	x	x	x			x					x		
	Nobody complained	x		x	x	x			x	x			x		
	We do not have time	x					x	x							
	Performance is not important in our case	x							x				x		
	Some rules are irrelevant		x										x		
	Results are not well presented	x			x		x					x		x	x
	The linter lacks precision									x	x			x	
	The linter does not have enough checks					x	x			x					
	It is difficult to write new rules		x												

Table 5.2 Results of the coding process.

5.2 Results

We present in Table 5.2 the results of the coding process in order to contribute to the transparency and verifiability of our study.

5.2.1 Why Do Android Developers Use Linters for Performance Purposes?

Linter Helps Developers to Learn about the Framework

As the Android framework is relatively young, developers are eager to learn about its underlying performance constraints. Half of the participants stated that the linter is very instructive in that respect (P1, P2, P3, P7, P9, P10, P11). *"Lint will actually help you become a better Android programmer"* (P3). Indeed, the performance checks of the linter mentor the developers to use the framework efficiently: *"I see the performance checks as a guide of the Android framework"* (P7). Other participants mentioned that their understanding of performance, and their programming habits evolved thanks to the linter: *"Everytime I learn a new thing about performance, and then it becomes a habit"* (P11). As an example of these cases, participant P9 mentioned DrawAllocation, a bad practice that consists of allocating new memory spaces in the `onDraw()` method: *"I was creating a new object in a paint method so Lint gave me a warning. That was the first and last time I see it, because now I pay attention to this"* (P9).

Some participants emphasized that junior Android developers should particularly use the linter for performance: *"Lint checks are extremely useful for helping out beginner Android developers or junior members of my team to enforce better code performance"* (P2). Indeed, junior Android developers, even if they have a prior experience in desktop development, may lack understanding of mobile framework specificities. Thus, they are prone to performance-related bad practices and they need the linter to learn how to keep their mobile apps efficient.

Discussion: Participants from previous studies have also reported that learning is one of the main benefits of using linters [47, 192]. Specifically, the developers learned with the linter about the syntax of the programming language, idioms, or libraries. In the case of our study, the participants showed that even important concepts about the Android framework can be learned through the linter. This can be a great incentive for Android developers to use linters with performance and other framework related checks.

Linter Can Anticipate Some Performance Bottlenecks

Many participants reported that the linter supports them in detecting bad practices that can cause performance bottlenecks (P1, P2, P5, P6, P10, P13). *"Lint is very useful to*

tell in advance what is going wrong" (P1). The participants stated that the linter is very efficient in detecting code-level performance issues, *"Lint is very good at finding patterned issues that follow a specific rule and can be solved very simply"* (P2). When asked why they want to anticipate performance problems, the participants reported that performance issues should not be noticed by users, "*it is always better to detect eventual problems before they are reported by end-users or managers.*"(P10). In fact, when end-users notice performance issues, they can uninstall the app or give bad reviews, and from there it can be hard to gain back users confidence. Moreover, the participants explained that once bottlenecks are reported by users, it can be complex to identify their root cause and fix them. *"by using Lint, we try to detect performance issues as soon as possible, because when they occur, they are generally very diffuse so it is hard to detect them with the profiler"* (P5). For instance, *"when I have 8 threads with cross-thread method calls, locating a problem with the profiler becomes very difficult"* (P5). In that respect, anticipating performance bottlenecks saves also time for developers.

Discussion: Previous studies have shown that developers prefer to manage performance reactively, and hence to wait the problems to occur before handling them, with the objective to gain time [125]. Here, our participants express a different point of view. They explain that when performance bottlenecks occur, they require so much time and effort to be fixed. They are hence in favor of a more proactive approach that aims to detect and fix bottlenecks before they occur. It is important to transmit this information to developers community, and especially novice Android developers. The latter may be unaware of the complexity of locating and fixing performance bottlenecks, thus they can make wrong strategic choices. As there should be a trade-off between reactive and proactive approaches, we also encourage future works to make real world comparisons between them.

Linter Is Easy to Use

Many developers said that they use the performance checks of Android Lint because they found it simple and easy to use (P2, P4, P6, P8, P11, P14). Indeed, Android Lint is already integrated in the IDE and all the checks, including performance, are enabled by default: *"it is built into Android studio, the checks performed by the linter appear automatically so we get those benefits just kind of immediately through the syntax highlighting"* (P2). Hence, the usage of the linter is seamless and effortless: *"it is just built into the tooling so well and so seamlessly, and it does not really cost me anything to have Lint running"* (P2). The participants also appreciated the integration in other development tools: *"we can easily integrate it in the Gradle and continuous integration, so we can ensure that performance and other Lint checks are respected"* (P6).

Discussion: This benefit is more related to the use of the linter itself rather than to performance concerns. Previous studies showed that static analyzers should be easy and seamless to encourage developers to use them [114]. Our findings confirm this, as participants stated clearly that they use the linter because it does not cost them anything. Now, Android Lint has an additional privilege by being integrated in the official IDE, Android Studio, and activated by default and this motivated developers to adopt it. This fact aligns with previous works [47, 114, 172] where researchers suggested that the integration in the development workflow helps static analyzers to accumulate the trust of developers.

Linter Can Develop the Performance Culture

In development teams, developers can have different perceptions and levels of interest regarding performance. Thus, it sounds important to rely on a linter for enforcing performance good practices (P1, P5, P7, P10). The use of a linter ensures that all team members follow the same performance guidelines: *"we have to make sure that everyone respects the rules and understands why we are using the performance checks of Lint"* (P10). On top of that, the performance checks that will occur will certainly be the source of discussions among team members: *" the objective is to share and level our knowledge on performance. When Lint reports on performance problems, we can disagree with each other on whether to consider it or not, so we will discuss and understand the problem, then make a wise decision about it"* (P5). That being said, the usage of the linter at a team level is fruitful in many ways. On one side, it allows to keep all the team members at the same page about performance choices. Besides, it arises discussions about performance, and thus enriches the performance culture in the team.

Discussion: Previous study showed that developers use the linter to have an objective tool that avoids endless discussions about code style [192]. The statements of our participants show that the linter itself can trigger discussions in the context of performance. Unlike code style, performance is an important aspect that requires a deep thinking from developers especially in the context of mobile apps. Hence, it is normal that developers appreciate the discussions triggered by the linter about it.

Linter Can Save the Developer's Time

Some participants explained that they use the linter to automate time-consuming tasks (P2, P3, P9). In particular, the linter can fill in for repetitive tasks: *"Lint helps you to save time in a lot of ways. One that comes to my mind is the identification of unused resources. The linter saves me a lot of time as I don't have to cross-check all resources manually"* (P2). Indeed, keeping unused resources is a performance bad practice that increases the APK size. Developers used to manually check and remove all the useless resources, which can be tedious

and error prone when there are many resources and when the app is built with many flavors. Using Android Lint to detect unused resources then helps developers and saves their time. Another task where the usage of linter helps saving time is code review: *"it is a quick and easy code review. It gives you a bunch of information. Now, whether I want to implement those messages or not, that is another decision"* (P3). Code review is an important repetitive task in software development that can be very time consuming, especially at team level. As stated by the participants, the linter can partially automate this task, so that developers do not have to review trivial performance issues and can therefore focus on important aspects.

Discussion: Linters are also known for saving time in contexts where eventual issues can be automatically detected, *e.g.,* bug detection [114]. The particularity of the cases reported by our participants is that on top of saving time by detecting eventual issues, the linter automates repetitive tasks. As these tasks are concrete and concern all types of Android apps, this can be more appealing for developers to adopt the linter.

Linter Contributes to Increase Credibility and Reputation

Developers always want to maintain a good reputation among their peers, and the linter can help them to do that (P1, P6, P7). First, developers need to keep their credibility among colleagues: *"I have to use Lint before making a release to make sure that my work meets the expectations of my superiors and colleagues"* (P6). Another context where reputation is important is open-source projects: *"before releasing code to open-source, I am required to check that all the warnings are away and that my code has a high quality"* (P1). (P7) added: *"I work on a company where we do some open-source projects. Before publishing code, I always have to run the linter to make sure of not making obvious performance mistakes. This impacts not only my credibility, but also the image of the company"* (P7).

Discussion: Since performance is critical in Android apps, making obvious mistakes can seriously affect the credibility of the developer among her peers. Hence, the linter performance rules can give a valuable support for Android developers to maintain their reputation.

Linter Raises the Awareness of the App Performance

Android apps are nowadays very complex software systems, and it is hard for developers to be aware of the implications of their development choices in terms of performance. Some participants stated that they use the linter to always be aware of their app performance (P3, P5). *"Sometimes Lint will catch performance aspects that I did not really think about, and so it will give me a time or a moment to think about and decide"* (P3). This shows that the linter messages incite developers to carefully think and give more attention to performance. This awareness can be particularly important when developers are working alone on the

project, *"if I am the only developer, for me this thinking is critical and Lint is a must-have"* (P3). This applies also to the cases where the issues reported by the linter are not applicable: "*in some cases I am not able to apply the changes requested by Lint. But still I need to have a good and valid reason for this. So I am aware of the trade-off I made*" (P5).

Discussion: It is important to distinguish the awareness of app performance with the learning of performance good practices. Here, the linter incites the developers to think and understand their app performance. When developers understand their apps, they can make decisions or solve eventual problems more easily.

Linter Eases Performance Debugging

Some participants did not only use the linter to directly improve performance, but they also obeyed non-performance rules to ensure high code quality, and consequently ease eventual performance profiling and debugging (P9). *"By using Lint and other static analyzers like Sonar, I ensure that my code is well designed and readable. So when a performance issue is reported, debugging and profiling becomes easier. I also can easily and quickly apply the fixes"* (P9). The linter then also helps to build clean and maintainable apps that further developers can easily debug and refactor for improving performance.

Discussion: This represents a benefit of using the linter in general, and not related to the usage of performance checks. However, it is still interesting to observe that some developers have a deep understanding of the software development process. Linters cannot prevent all performance bottlenecks, bugs or other issues. Therefore, developers should always keep their code clean and maintainable, because it makes further maintenance operations easier.

5.2.2 How Do Android Developers Use Linters for Performance Purposes?

Linter Integrates along the Project Life Cycle

The participants reported different strategies to use the linter through the project life cycle in order to keep their apps effective. In the remainder, we report on the strategies they identified.

Integrating from scratch: Many participants reported that they prefer using the linter from the project startup (P2, P5, P9, P10, P11, P13). Participant P5 explained that, when starting a project from scratch, she tries to keep the code very clean by considering all the Lint checks. When asked about the configuration adopted in this case, the participants said that they keep the default configuration provided by the linter, *"in this situation I do not need any additional setting, Lint is configured by default"* (P9). We also asked these participants about the motivation behind this strategy. They explained that, when the project advances

without the linter, it is more difficult to control performance *a posteriori*. For instance: *"we had a case where we retrieved a project that was built by another team without Lint. We have got thousands of errors and warnings. So we were less motivated to put back Lint and recover the app performance"* (P7). Indeed, it is easy with this strategy to motivate developers to respect performance checks because the code base is clean and there is no existing debt to tackle.

Targeting milestones: Five participants mentioned that they extensively use the linter at the end of features or for releases (P1, P5, P6, P7). *"I never use Lint in the beginning of the project or while prototyping. I use it for releases to make sure that my code meets the expectations"* (P6). As for features: *"towards the end of adding a new feature, I will run through Lint then I will go through all of them and I determine whether or not I want to spend the time to do it"* (P3). When asked about the configuration used for this strategy, participant P5 stated: *"we have different Lint profiles, in the release profile we activate additional rules"*. This strategy allows developers to go fast while producing new features or prototyping without hindering the final app performance.

Adopting performance sprints: Two participants reported that they dedicate sprints for improving the app performance (P5, P12). Participant P12 stated: *"while working, we do not have concrete performance objectives. But when we notice that the app starts lagging, we decide to dedicate time for this and improve the app performance"*. As for participant P5: *"generally while coding, we try to respect the performance checks just as other Lint checks. Then, we regularly program performance sprints and there we will be specifically interested in performance rules"*. While the strategy reported by participant P12 is purely reactive, the strategy of participant P5 is still proactive.

Improving performance incrementally: One participant explained how she deals with legacy code where the linter was not used before (P9). *"I configure Lint to incrementally increase the app performance. I define a baseline, then I work to decrease this baseline progressively. I also try to ensure that the new code is always more effective than the old one"* (P9). Android Lint allows to define a baseline file—*i.e.*, a snapshot of the current warnings of the project. The baseline allows to use Lint without addressing old issues. When asked about how the incremental improvement happens, the participant P9 replied *"I change the severity of some checks, for example I configure some rules to block the compilation"*.

Discussion: Integrating the linter from project start-up is commonly advised [192]. Moreover, previous studies show that developers are less likely to fix the linter warnings on legacy code [31]. The statements of some participants are aligned with this common wisdom. However, the strategies (b) and (c) show that developers can adopt the linter differently

according to their work style. In particular, developers who are prototyping or are in rush to release can adopt strategy (b) and apply the linter after finishing their core development. Interestingly, developers who prefer to manage performance reactively can also leverage the linter by following strategy (c). Finally, strategy (d) shows that the configurability of the linter maximizes its chances to be adopted.

Linter Can Be Used under Different Settings

Some participants reported using the linter individually, while others explained how they use it with their team.

Supporting individual development: Half of the participants reported that the usage of the linter was a personal choice (P1, P3, P5, P8, P9, P11, P13). *"I only run Lint as myself as part of a review that I want to do"* (P3). These participants usually use it from the IDE interactively: *"It is through Android studio interactively"* (P13).

Checking performance rules in a team: Other participants reported that the usage of the linter for performance purposes was required on a team level (P2, P4, P6, P7, P9, P10, P12, P14). *"In the team, Lint avoids accumulating problems. We have a defined set of rules and every team member must respect them"* (P10). In this cases, the linter is generally a step in the continuous integration chain: *"it is set up with continuous integration, so Lint runs after every commit and you will get a report. Then, you can choose to look at it or not"* (P2).

Discussion: The reported settings of using the linter do not apply exclusively for performance. The participants explanations underline the importance of the linter interactivity and integration in the development workflow.

Linter Prioritizes Performance Aspects

Many participants said that while they use the linter, they prioritize performance related issues (P2, P3, P5, P6, P9, P13). For instance: *"there are so many different checks but I would say performance usually catches my eye"* (P2). Some participants gave also distinct priorities to different performance aspects: *"if it is anything about threading, I will take a look at it and review it before deciding if I want to fix it or not"* (P3), *"I give so much importance to UI performance and all memory-related issues"* (P13). Some participants expressed these priorities with a configuration: *"I changed the severity of rules that interest me, so they block the compilation"* (P9).

Discussion: Each app can have different specificities and needs in terms of performance. Thanks to configurability, the linter can help developers to focus on performance aspects that sound relevant and critical for them.

5.2.3 What Are the Constraints of Using Linters for Performance Purposes?

The constraints reported by participants were structured around two main topics: (i) social challenges and (ii) tool limitations.

Linter Faces Social Challenges

The participants reported cultural elements that make the use of linter for performance challenging. The participants encountered these issues in their work environment, with colleagues or superiors.

Static analysis is not suitable for performance: Many participants described that developers generally think that static analysis is not suitable for improving performance (P1, P2, P3, P4, P7, P12). Participant P1 stated: *"I think that there is a gap in understanding why static analysis is useful for performance".* Participants explained that this mindset is due to the nature of static analysis: *"because Lint is only looking at the code. Some developers feel that there should be a better tool that analyzes the app while running"* (P3). Other participants thought this gap is due to the complexity of performance issues: *"for the actual real-world bottlenecks that most apps face, it is not the Lint that will help you. Performance issues are very complicated or have multiple causes that cannot be pinpointed to a one line of Java code"* (P2). Participant P4 stated that this gap may be due to the confusion of the term "performance issue": *"For each performance issue, there is a root cause and an observation. The term performance issue is often used to refer to both. But, it is necessary to distinguish them. The default Lint rules contain some basic and trivial root causes, which could statically be identified. But, in some cases, you have an observation and you cannot guess the root cause. So here Lint cannot help you. To sump up, Lint requires you to define in advance what you are looking for, it is hard to use it to match the observation and the cause".*

Nobody complained: Many participants reported that they regularly deal with colleagues and superiors who believe that performance should be managed reactively (P1, P3, P4, P5, P8, P9, P12). For example, participant P5 stated that the common rule in her environment is *"only when the superiors or the end-users complain that the app is lagging or not smooth, we say ok we have to care about performance".* Participant P5 highlighted a case where this pressure came from a superior: *"performance refactoring is a back-office task that the product owner cannot see. It is hard to negotiate these tasks".* Some participants underlined that this mindset is particularly tied to performance more than any other software quality aspect: *"with performance you do not want to do a lot, you want to make sure that you are really looking at the issue and not trying to over optimize the code"* (P3).

We do not have time: This mindset is very related to the previous one. However, we observed cases where the developers explain that the performance checks of the linter are not considered only for time constraints without any explicit agreement on the management of performance reactively (P1, P6, P7). Participant P1 reports observing this mindset in many companies: *"why waste time on performance rules? Let us move ahead and we will figure about this later. Unfortunately that later never happens, or comes only when the problem is big enough"*.

Performance is not important in our case: Some participants reported working in contexts where performance was considered irrelevant (P1, P8, P12). Participant P1 reports experiencing this situation in young projects: *"when you build a small app and do not know whether it will scale or not, it does not seem useful to spend time in making sure that static analysis of performance is done right"*. Participant P8 described a case where performance was considered irrelevant for a type of apps: *"we did very basic app development and not particularly hardware-based development. We developed Uber clones, and all those apps did not require any specific performance from the device"*.

The linter rules are irrelevant: Two participants described that some performance checks are considered irrelevant (P2, P12). Participant P12 gave examples of Lint performance rules that do not really have an impact: *"OverDraw is a rule that used to be relevant a long time ago, but now with powerful smartphones it is not anymore. Moreover, the Android system detects all these trivial issues and fixes them automatically. Developers are obsessed with Overdraw, it became a cult actually, but this is not what really hinders the performance of your app"*. The problem OverDraw occurs when a background drawable is set on a root view. This is problematic because the theme background will be painted in vain as the drawable will completely cover it.

Discussion: Calcagno *et al.* [39] have already referred to the social challenge while describing their experience in integrating a static analyzer into the software development cycle at Facebook. Some of the reported challenges, *e.g.,* we do not have time, apply to linters in general. However, the other mindsets are particularly resistant to the use of linters for performance. The belief that static analysis is not suitable for performance seems to be prevalent, six participants mentioned it explicitly. Developers are using linters as tools that report minor issues like styling violations and deprecations. They are not aware enough of the capabilities of static analysis in detecting performance issues. Tool makers should put more efforts on communicating about linters as tools that can accompany developers in different development aspects. The mindset that performance should be managed reactively confirms previous observations about Android apps development [125]. This finding shows that Android developers still lack understanding about the implications of performance bottlenecks.

As for linter rules that are considered irrelevant, this incites the research community to dig deeper into the impact of these practices. For many bad practices, like OverDraw, we still lack precise measurements of their penalties on performance in real world contexts.

Linter Suffers from Limitations

The participants reported several linter limitations that make it complicated to use it for performance purposes.

Not well presented: Several participants mentioned that the linter rules are not well organized or explained (P1, P4, P6, P11, P13, P14). Interestingly, three participants said explicitly that, for a long time, they did not even know that Android Lint had performance related checks (P1, P11, P14), *"I did not know there are different categories in Lint. For me it is just Lint, I do not distinguish between them"* (P1). Furthermore, participant P14 explained that in the beginning she did not know that some rules are related to performance, and thus she treated them as other checks like typography. Other participants complained about the unclarity of the messages, *"it is not always clear, for example performance checks about graphics, I cannot really understand them at all"*. On top of that, some participants found that rules are not well organized: *"there is a hierarchy with levels of importance, but I find it useless, it does not help me. So if I want to focus only on performance aspects, I have to search everywhere"* (P6). The same participant underlined the unclarity of the priorities given by the linter to the checks: *"I try to obey, but I do not really understand the logic behind the priority and severity"*. Indeed, Android Lint does not give explanations about the priority and severity attributed to each check, so we cannot understand the rationales behind them.

Imprecision: Some participants complained about the imprecision of the detection performed by the linter for performance checks (P9, P10, P13). Participant P9 described situations where the code contained a performance bad practice, but the corresponding linter check was unable to detect it: *"in some drawing methods, I was calling a method that called another one that made an intensive computing with memory allocations. Lint did not detect this as DrawAllocation, it actually does not look so deep into the code"*. Other participants reported false positives in performance checks. Participant P13 said: *"I regularly have false warnings about unused resources when the resource is used via a library"*, and participant P10 stated: *"Lint indicates an unused resource but the image is actually used with a variable"*.

Poverty: Some participants mentioned that the linter is not very rich with performance checks (P5, P6, P9). Participant P5 stated: *"I do not see so many performance-related Lint checks. And the existing ones are so generic"*. In the same vein, participant P9 said: *"I rarely*

see suggestions or important warnings about performance aspects. Very few!". Furthermore, the participants complained about the absence of linter checks for Kotlin (P6, P9).

The difficulty of writing new rules: Participant P2 described her trial to write a linter check and the difficulties she faced: *"I wanted to write specific Lint rules and after a few trials I ended up discovering that it is difficult to define something general enough to warrant a Lint check. Also, the effort put in to build a custom Lint check is pretty high. That is why it is not a common tactic for development teams especially on a per project basis"*. The participant pointed also the complexity of using the created rule in a team: *"to build a Lint check, then distribute it to the team, then have the whole team use it, is difficult"*.

Discussion: The fact that at least three participants reported that for a long time they used the linter without noticing these performance checks was striking. Tool makers have to work more on showcasing different checks categories. Also, the linter messages should highlight more the impact of performance practices to motivate developers to consider them seriously. The imprecision is a common limitation of linters [114], and performance checks are no different in that respect. Similarly, the participants statements about the difficulty to write linter rules align with previous works [47]. As many other tools, Android Lint provides the possibility to write new rules but this task is complex and time-consuming. Thus, developers are not motivated to write their own rules.

5.3 Implications

We summarize in this section the several implications of our results for developers, researchers, and tools creators. Our findings are based on the usage of linters for performance purposes in Android. Nevertheless, they can also apply to other development platforms.

5.3.1 For Developers

Our results provide motivations for developers to use linters for performance and show them how to maximize their benefits.

Benefits

Developers can find several benefits in using the linter for performance. In particular, developers can use the linter with performance checks to:

— Learn about the mobile framework and its performance constraints,

— Anticipate performance bottlenecks that can be arduous to identify and fix,

— Develop the performance culture in the team,

— Save time by automating concrete repetitive tasks,

— Save their reputation among peers,
— Increase their awareness and understanding of their apps performance.

Developers should also be aware that the usage of the linter is seamless and can be integrated in along the development workflow.

Usage Fashions

Our participants recommend to use the linter for performance in the following ways:
— From project startup to motivate developers to keep the code clean,
— Only before releases to dedicate the early development stages only for prototyping and making the important features,
— In performance sprints: developers can configure the linter to focus only on performance in some sprints. This approach works also for developers who prefer to manage the performance reactively,
— Improve performance incrementally: developers should configure the linter carefully on legacy code to avoid chaos and developers discouragement,
— Individually in an interactive way in the IDE or in a team with the continuous integration,
— Prioritizing performance aspects: developers can configure the linter to focus on performance aspect that interest them and fit with their app needs.

5.3.2 For Researchers

Our findings confirm hypotheses from several previous works and open up perspectives for new research directions:
— We confirm that the mindset of managing performance reactively is prevalent [125]. As there should be a trade-off between reactive and proactive approaches, we encourage future works to make real world comparisons between them;
— Our study confirms that some Android developers are unaware of code smells and their importance;
— Some developers challenge the relevance and impact of performance bad practices. We therefore encourage future works to investigate and provide precise evidences about the impact of such practices;
— Some developers are eager to consider more performance-related checks. This should incite researchers to identify and characterize more practices that can hinder the performance of mobile apps.

5.3.3 For Tool Creators

Our findings confirm the importance of some linter features and highlight new needs and challenges:

— Our findings align with previous works that suggest that simplicity and integration in the development workflow help static analyzers to increase the trust of developers [47, 114, 172]. We encourage tool creators to ease the integration of their linters in development tools;
— Our findings show that linters should be more clear and explicit about the categories of checks. Clarity is also required in the explanations of the potential impacts of performance checks. We cannot expect from developers to seriously consider the rules if we do not provide enough information about their nature and impacts;
— Providing the possibility to write linter rules is not enough. Writing a linter rule should be simple and less time-consuming to motivate developers to do it;
— Tool makers should put more efforts in communicating about the capabilities of static analysis in detecting performance bad practices;
— Given the benefits reported by participants, we invite more tool makers to include performance-related checks in their linters.

5.4 Threats To Validity

We discuss in this section the main issues that may threaten the validity of our study.

Transferability: One limitation to the generalizability of our study is the sample size. The sample size is not large and thus it may not be representative of all Android developers. To alleviate this fact, we interviewed highly-experienced Android developers. Nonetheless, this selection may also introduce a new bias to the study. As a matter of fact, the benefits and constraints reported by junior Android developers can be different. We would have liked more participants. However, transcribing interviews is a manual and time-consuming task. Hence, having more interviews may involve more workload and would affect the accuracy and quality of the analysis. We found our results valuable and enough for theoretical saturation. The study conducted by Tómasdóttir *et al.* [192], which we discuss in our related works, approaches a similar topic with a similar sample size (15 participants). Another possible threat is that we only interviewed Android developers who use the linter for performance purposes. Android developers who use the linter without performance checks may also have their word to say about the limitations of using a linter for performance. Thus, more studies should be conducted to understand why some Android developers use the linter and disable performance checks. As we focused our study on Android Lint, our results cannot be generalizable to other Android linters. However, we believe that the choice of Android Lint was sound for two reasons. First, it is a built-in tool of Android Studio and is activated by default, thus a large proportion of Android developers should be using it. This fact was confirmed in our preliminary online survey, where 97 % of the participants who used the linters were actually relying on Android Lint, *cf.* Appendix B and our companion artifacts [99].

Secondly, it is the only linter that has performance checks specific to the Android platform, and this detail is the core of our study.

Credibility: One possible threat to our study results could be the credibility of participants answers. We interviewed developers who have a strong knowledge about the topic. However, we cannot be sure that their answers were based on real experience or knowledge acquired from external resources. To alleviate this issue, we tried always to ask for details and relate to developers project and working environment. Also, we emphasized before the interviews that the process is not judgmental.

Confirmability: One possible threat could be the accuracy of the interviews analysis and particularly the coding step. We use a consensual coding to alleviate this threat and strengthen the reliability of our conclusions. Each interview has been encoded by at least two authors. Initially, every author coded the interview independently to avoid influencing the analysis of other authors. Afterwards, the authors discussed and compared their classifications.

5.5 Summary

We investigated in this study the benefits and the usage of linters for performance concerns in Android apps. Specifically, we conducted a qualitative study based on interviews with experienced Android developers. The results of our study provided many indications about the usage of linters for performance:

- **Finding 1:** Linters can be useful for anticipating and debugging performance bottlenecks and they can also be used to raise performance culture among developers;
- **Finding 2:** Besides the intuitive proactive approach for using linters, developers can also use linters reactively in performance sprints;
- **Finding 3:** Linters should be more clear and explicit about the categories of checks. Clarity is also required in the explanations of the potential impacts of performance checks.

Based on these findings, we encourage future studies and tool makers to aliment their static analyzers with more performance code smells. We also, suggest that these tools should be more explicit about code smell definitions and categories.

Our results also provided important insights about developers' perception of Android code smells:

- **Finding 4:** There is a prevalent ignorance of mobile-specific code smells among developers;
- **Finding 5:** The mindset of managing performance reactively is common [125]. Future studies should adapt to this mindset and focus on reactive approaches and tools for dealing with code smells like profilers;

— **Finding 6:** Some developers challenge the relevance and impact of performance bad practices. We therefore encourage future works to question code smell definitions and provide precise evidences about the importance of such practices.

The finding about code smell irrelevance from the viewpoint of certain developers drew our attention and incited us to question the foundation of mobile-specific code smells. Hence, we decided to dedicate our next study to an inspection of code smell definitions.

Chapter 6

A Second Thought About Mobile Code Smells

Our previous chapter highlighted several complaints from developers towards the instances of mobile-specific code smells. These complaints concerned false positives, irrelevant instances, and instances that did not show a real impact on performance. To understand the rationales behind these complaints, we dedicate this study to an investigation of instances from mobile-specific code smells. To frame this investigation, we formulate the research question:

RQ: What are the instances of mobile-specific code smells?

Our objective with this open question is to inspect the instances of code smells without a prior hypothesis about their characteristics. Therefore, we adopt in this study an exploratory approach that aims, with an open-mind, to build a theory about mobile-specific code smells. This theory describes the characteristics of code smell instances and allows us to check the presence of the flaws reported by developers. Moreover, as code smell instances are the product of the adopted definitions and the detection techniques, this theory can also provide insights about the flaws of the definitions and detection techniques. These insights are important because they provide second thoughts about the established foundation about code smell definition and detection. As a matter fact, mobile-specific code smells were defined from the official framework documentation and other online resources, like blogs and forums. Contrarily to OO code smells, mobile code smells were not defined from an analysis of the state of practice. Hence, it is important to question these definitions and check their relevance in practice. As for the detection techniques, the insights are important to verify the relevance of the approach followed by code smell detection tools. While all the tools that targeted the catalog of Reimann *et al.* [169] relied on static analysis, we still lack knowledge about the adequacy of such technique for detecting mobile code smells. This questioning aligns with the concerns raised by developers about the suitability of static analysis for detecting performance-oriented practices.

The remainder of this chapter is organized as follows. Section 6.1 explains the study design, while Section 6.2 reports on the results. Section 6.3 discusses the implications and threats to validity and Section 6.4 concludes with our main findings.

6.1 Study Design

In this section, we explain the process that we followed to answer our research question. First, we describe the selection of code smell instances for our study. Then, we define our approach for analyzing these instances and building the desired theory.

6.1.1 Data Selection

The selection of code smell instances is determined by the choice of (i) the mobile platform, (ii) the definitions, (iii) the detection tool, and (iv) the dataset of mobile apps to analyze.

Mobile Platform

The motivation behind this study lays in the concerns raised by Android developers about the linter. These concerns were intended for Android-specific code smells and we do not know if they are generalizable to other mobile code smells. Thus, we chose to keep the focus of this investigation on the Android platform.

Definitions

As shown in our literature review in Chapter 2, studies of Android-specific code smells adopted definitions from the catalog of Reimann *et al.* [169]. These definitions are inspired from the Android documentation and they resembled to the definitions adopted by tools like Android Lint [9]. Hence, we adopted these definitions in our process for generating code smell instances. We recall these definitions in our qualitative analysis in the upcoming sections.

Detection Tool

There are different tools that allow the detection of mobile-specific code smells. In this study, we chose to use Paprika because it relies on explicit code smell definitions. Paprika isolates code smell definitions in a querying layer, which is independent from the rest of the detection process. This isolation allows us to tune the code smell definitions when needed. Hence, we can ensure that the resulting instances are faithful to the code smell definitions that are commonly adopted by the research community. Paprika relies on the technique of static analysis, which is also used by other tools like Android Lint [9] and aDoctor [158]. Consequently, observations about detection flaws from Paprika can be—with necessary precautions—generalized to other detection tools for mobile-specific code smells.

The choice of the study code smells is determined by the tool choice. We included in this study all the objective code smells that are detectable by PAPRIKA. Subjective code smells rely on subjective numerical thresholds. For instance, the detection of the code smell *Heavy Service Start* in PAPRIKA uses the rule `number_of_instructions>17` to determine whether the method `onStartCommand()` is heavy or not. These thresholds, 17 in our example, can be defined by researchers, developers, or from dataset statistics, and thus they vary depending on the tool and the study. Therefore, we excluded subjective code smells to ensure that our studied instances are not altered by subjective choices.

After excluding the subjective and deprecated code smells, PAPRIKA can detect eight Android-specific code smells. We present in Table 6.1 these eight code smells and their host entities—*i.e.*, the type of source code structures that can host them.

Code Smell	***Host entity***
No Low Memory Resolver (NLMR)	Activity
Init OnDraw (IOD)	View
UI Overdraw (UIO)	View
Leaking Inner Class (LIC)	Inner class
Member Ignoring Method (MIM)	Method
Hashmap Usage (HMU)	Method
Unsupported Hardware Acceleration (UHA)	Method
Unsuited LRU Cache Size (UCS)	Method

Table 6.1 Studied code smells.

Dataset

We relied on the dataset that we built in our change history studies, *cf.* Chapter 4. The dataset included 324 Android apps with contributions from 4,525 developers. Applying PAPRIKA on this dataset resulted in 180.013 code smell instances. As our study relies on manual qualitative analysis, we could not analyze all the detected instances. Therefore, we selected from our dataset a sample of code smells to analyze. We used a stratified sample of 599 instances to make sure to consider a statistically significant sample for each code smell. This represents a 95 % statistically significant stratified sample with a 10 % confidence interval of the 180,013 code smell instances detected in our dataset. The stratum of the sample is represented by the eight studied code smells.

6.1.2 Data Analysis

In this subsection, we describe our approach for analyzing the collected data in order to answer our research questions.

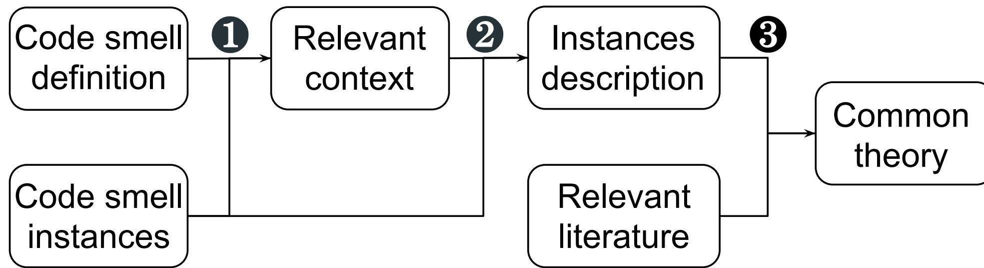

Figure 6.1 Design of the qualitative analysis.

Our objective is to explore with an open-mind the characteristics of Android-specific code smell instances. Therefore, we relied on Grounded Theory [90], a well-established methodology that allows to generate a theory from qualitative data. The literature contains different schools about the design and implementation of Grounded Theory [44, 89, 184]. In our analysis, we followed Charmaz's approach [44], which combines Glaser's inductive school with constructivist epistemology. This approach has been adopted in several software engineering studies to build theories [32]. We considered in this study the feedback of these works and we carefully followed their recommendations in our research design. Figure 6.1 depicts the design of our qualitative analysis with its three main steps.

Step 1: Extract Relevant Context

Input: Code smell definition and sample instances.
Output: Relevant context for instance analysis.
In this step, we defined for each code smell type the relevant context for its qualitative analysis. This context consists of the source code elements that should be considered in the manual analysis to be able to understand every code smell instance. Initially, we defined this context based on the code smell definition and the code smell instances available in the sample. Afterwards, if the analysis step requires the inspection of other parts of the software system, we augment the context with additional source code elements.

Step 2: Extract a Description for the Instances

Input: Relevant context and sample instances.
Output: Instances description.
In this step, we manually analyzed every instance of the sample based on the extracted relevant context. In this analysis, we explored the common characteristics of code smell instances without a predefined topic in mind. For most of the analyzed code smells, this exploratory approach led to a coding process.

Coding process: We followed this process in the cases where code smell instances showed interesting analytical categories that can be used to describe the dataset. The coding process follows two steps:

1. **Build a coding guide:** We analyzed each instance to identify the categories included in it. Then, we assembled the categories into an analytical guide and we formulated for each category different versions. The versions stand for different subcategories identified in the instances in reference to one of the categories. Once the coding guide is built, it can be tested and validated during the open-coding. Indeed, the categories and their versions may be refined, made more distinctive or completely omitted from the coding guide;
2. **Open-code the instances:** We analyzed each code smell instance in order to assign it to a category and a sub-category. We performed these assignments with a constant comparison to iteratively and continuously compare categories and aggregate them into higher levels of abstraction and therefore update our coding-guide. To strengthen the reliability of our assignments, we also used a consensual coding. That is, each instance is coded by at least two authors. Initially every author coded the instances independently, afterwards, the authors discussed and compared their categorization. In case of discrepancies, the authors attempted to negotiate a consensual solution.

The result of the coding process is a structured description of code smell instances. For some code smell types, we did not identify any analytical categories and we did not follow a coding process. Hence, their description consisted in unstructured insights about the code smell instances.

Step 3: Build a Theory

Input: Instances description and the relevant literature.
Output: A theory about mobile-specific code smells.
In this step, we analyzed for each code smell type the instances description in light of the existing literature. This literature includes all resources that can help us to interpret the instances description and generate new concepts. Indeed, depending on the code smell type, the literature can include research works or resources about the Android framework and the Java programming language. The analysis in this step follows two phases:

1. **Analyze the descriptions:** We inductively analyzed the descriptions of each code smell type separately and discussed them in light of the relevant literature. The objective of this induction is to highlight interesting topics and generate new concepts and relationships that can later serve in our common theory;
2. **Assemble a common theory:** Based on the discussions of the eight code smells, we built a common theory that describes all the studied mobile-specific code smells. In

particular, we identified the recurrent concepts and relationships from the discussions and assembled them to formulate one common theory for all studied code smells.

The analysis process from Step 1 to Step 3.1 were performed on each code smell type, separately. It is only at Step 3.2 that we consider all the code smell types together.

6.2 Study Results

This section reports the results of our qualitative analysis by following the steps depicted in our design, *cf.* Figure 6.1. For each code smell type, we remind the commonly adopted definition, then we present the relevant context that we extracted for the manual analysis. We present the results of this manual analysis as a description of the instances. Afterwards, we discuss and analyze this description in light of the related literature. Finally, we assemble our results in a theory that describes the eight studied code smells. To leverage the replication of this study, all the presented results are included in our companion artifacts [94].

6.2.1 Leaking Inner Class (LIC)

Sample

The *LIC* sample included 96 instances.

Definition

In Android, a leaking inner class is a non-static anonymous or nested class. These classes are considered as *"leaking"* because they hold a reference to the containing class. This reference can prevent the garbage collector from freeing the memory space of the outer class even when it is not used anymore, thus causing memory leaks [9, 169].

Relevant Context

Based on the definition and instances of *LIC*, the manual analysis must cover the inner class, which hosts the code smell, and its outer class. During the analysis and coding process, we did not encounter cases where the inspection required analyzing other source code elements.

Instances Description

Our manual analysis of *LIC* showed that the instances can be categorized based on their (i) anonymity and (ii) class type. Hence, we built an analytical guide to analyze the instances. The categories of this coding guide are:

— Anonymity: the subcategories are anonymous classes and nested classes,

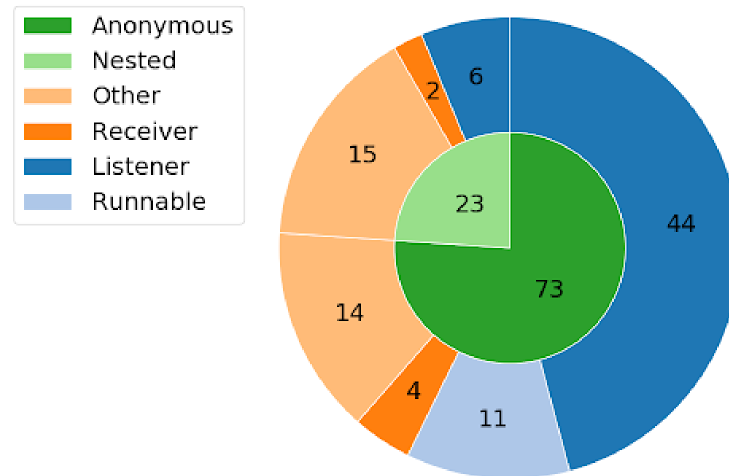

Figure 6.2 A sunburst of *LIC* instances in practice.

— Class type: the subcategories are Listeners, Runnables & Threads, Receivers, and other classes.

Based on this analytical guide, we performed an open-coding of the sample instances. The results of this coding are summarized in the the sunburst of Figure 6.2.

Anonymity: We observed that the code smell *LIC* manifests in:

— *Anonymous classes* (73 instances): this represents Java local classes that do not have a name. These classes are generally used to keep the code concise and only used once;

— *Nested classes* (23 instances): this represents non-static nested classes, which are also called *inner classes*.

This categorization shows that 76% of the analyzed *LIC* instances are anonymous classes and only 23% of them are nested classes.

Class type: Based on the class type, we found that *LIC* instances fall under the following categories:

— *Action Listeners* (50 instances): This represents classes that are commonly used by developers to handle UI-related events. They define the behavior of the app in case of certain user actions;

— *Runnables & Threads* (11 instances): This represents classes used to implement concurrency. In our sample, these classes were used as anonymous inner classes to easily launch parallel tasks;

— *Receivers* (six instances): This represents the Android framework classes that are used for receiving and handling `Intents`;
— *Other classes* (29 instances): This included Android framework classes like `ViewHolder` and `Adapter`, but also other types of classes.

Apart from the class anonymity and type, we also noticed that some instances of *LIC* accessed non-static attributes and methods of the outer class. Listing 6.1 shows a minified example of these instances that we found in the app Open Android Event [43]. In this example, the instance is an anonymous listener that updates an image view based on the user input in a dialog interface. To perform this, the listener accesses `mapImageView`, which is a non-static attribute of the outer class. Because of this non-static access, it is impossible to make the listener static.

Listing 6.1 A *LIC* instance accessing non-static attributes.

```
public class MapFragment extends Fragment {
  TouchImageView mapImageView;
  private void selectDay() {
    AlertDialog.Builder b = new AlertDialog.Builder(getActivity());
    b.setItems(types, new DialogInterface.OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
        switch (which) {
          case 0:
          mapImageView.setImageResource(R.drawable.eventmap_tues_wed);
          break;
          .....
        }
      }
    });
  }
}
```

Relevant Literature

Based on the open-coding results, the relevant literature for our analysis is:
— The Java documentation of nested, inner, and anonymous classes [150, 153];
— Resources about event handling in Android [12].

Discussion

The description shows that, in practice, *LIC* instances are mainly anonymous classes (76%). The *LIC* definition suggests that these classes should be static to avoid memory leaks. However, the Java documentation affirms that anonymous classes cannot be static [150]. Thus, according to the *LIC* definition, every anonymous class in Android is an instance

of *LIC*. The problem with this generalization is that the usage of anonymous classes is a recommended programming fashion in Android as they simplify the management of UI events with listeners [12]. Our categorization based on the class type confirmed this fashion by showing that most of the anonymous classes were listeners. Hence, avoiding anonymous classes can can complex the management of UI-related event. Moreover, as it is impossible to make anonymous classes static, the only possible refactoring for these *LIC* instances would imply their replacement by other structures like static nested classes. Nevertheless, implementing listeners with nested classes can be counter-intuitive for developers.

Our observations also showed that some *LIC* instances are legitimately non-static because they access non-static attributes of outer classes. As a result, considering these classes as code smell instances is inaccurate. This shows that the current *LIC* definition is flawed, as it misses the cases where inner classes need to access their outer classes.

The current definition of *LIC* considers all anonymous classes as bad practices and leads to many confusing instances that are hard to avoid and refactor.

6.2.2 Member Ignoring Method (MIM)

Sample

The *MIM* sample included 96 instances.

Definition

This code smell occurs when a method, which is not a constructor and does not access non-static attributes, is not static. As the invocation of static methods is 15%–20% faster than dynamic invocations, the framework recommends making these methods static [108].

Relevant Context

Based on the code smell definition and sample instances, the analysis should cover the method, which is affected by the code smells, and its container class. However, after the analysis of some *MIM* instances, we realized that the documentation and comments of the container class should also be included in the analysis to allow an accurate judgment of the instances.

Instances Description

In our manual analysis, we observed that *MIM* instances can be categorized based on the content of their host methods. The method content interested us because it revealed insights about the method purposes. Hence, we built an analytical guide with the following category:

— Method content: the subcategories are empty, primitive, and full methods.

We adopted this guide to perform an open-coding of the sample instances.

Content	***Empty***	***Primitive***	***Full***
#Instances	46	27	23
%Instances	48%	28%	24%

Table 6.2 Distribution of *MIM* instances according to the method content.

We report in Table 6.2 the results of this coding and we detail in the following these subcategories:

- *Empty methods* (46 instances): This represents methods that have an empty body—*i.e.*, without instructions. As these methods are empty, they do not access any non-static attributes or methods. Hence, the code smell definition considers them as instances of *MIM*. By analyzing these methods, we found that they can be introduced for the following reasons:
  - *Prototyping*: In the early stages of development, developers can create prototype code that does not include all the functioning details. This code generally includes minimal classes and interfaces with empty methods. An example of this case is the method `parseText()`, which is introduced empty in the app Transistor [79] only for prototyping purposes;
  - *Semantic interfaces*: We found many classes that are semantically abstract even though they are not annotated as such in the source code. For instance, the class `MessagingListener` does not use the keyword `interface`, but it is presented by the comments and documentation [73] as an interface. This practice is used by developers to allow overriding some methods of the interface without the obligation of entirely implementing it. This kind of classes contained many empty methods that consequently represent instances of *MIM*;
  - *Abstract classes*: An example of this case is the abstract class `AsyncTask` [78] that contains many non-abstract methods with empty bodies, like `onPreExecute()`;
- *Primitive methods* (27 instances): This represents methods that only define a minimal behavior—generally a return statement. Some of these methods are also introduced during the prototyping stage and are intended to have a more sophisticated body in the future. For instance, a commit from the Anuto app [85] introduces an initial version of the method `initStatic()` that, instead of initializing an object, only returns `null`;
- *Full methods* (23 instances): Other recurring instances of *MIM* are methods with a full body that do not use any non-static attributes or methods. The method `getImage()` introduced in the TTRSS-Reader app [144] illustrates well these instances. This method only manages the Android resources and does not use any non-static elements, thus it should be static to fasten its execution.

Relevant Literature

Based on the open-coding results, the relevant literature for our analysis is the resources about the *MIM* code smell [108, 156].

Discussion

The manual analysis showed that only 24% of *MIM* instances are fully developed methods. Most of the *MIM* instances are empty or primitive methods (76%). These methods are used as part of several *programming fashions* like prototyping and semantic interfaces. Since these methods are not fully developed, they are not semantically faithful to the definition of *MIM*. Indeed, the *MIM* definition promotes static methods to gain execution time. Nonetheless, empty and primitive methods are unlikely to be executed in their current state. Indeed, empty and primitive methods that are used for prototyping are never released and shipped before being developed with more content. The same goes for empty methods that are used in semantic interfaces and abstract classes. These methods are semantically similar to abstract methods as they are defined only for design purposes and they will never be executed. Hence suggesting these empty and primitive methods to the developers as code smell instances may be *inaccurate*.

The current definition of *MIM* considers empty and primitive methods, which are used for prototyping and other design purposes, as bad practices.

6.2.3 No Low Memory Resolver (NLMR)

Sample

The *NLMR* sample included 94 instances.

Definition

This code smell occurs when an `Activity` does not implement the `onLowMemory()` method, which is called by the system when running low on memory to free allocated and unused memory. If not implemented, the system may kill the process [169].

Relevant Context

Based on the definition and sample instances, the manual analysis should consider the affected activities with their methods and attributes.

Instances Description

As this code smell covers a memory issue, we considered in our analysis memory-related aspects. In particular, we inspected the (i) data structures possessed by the activity (ii) and the mechanisms used by the activity to manage this data.

Data structures: We observed that some *NLMR* instances were activities that hold important memory structures requiring management. Specifically, we found three cases where the activities possessed a memory cache that could be released upon request to free memory. However, despite the cache presence, these activities did not implement a memory release mechanism.

Mechanisms: Besides the `onLowMemory()` method, which is already considered in the definition of *NLMR*, activities can use other mechanisms to free unused memory. In particular:

— The method `onTrimMemory()` can be used to respond to the Android system memory warnings [15];

— Developers can develop their own mechanisms to handle and free unused memory.

From our manual analysis, we found that activities affected by *NLMR* did not follow any alternative approaches for freeing memory.

Relevant Literature

Based on the open-coding results, the relevant literature for our analysis is the documentation of memory management in Android [14, 15].

Discussion

The description showed that *NLMR* instances are activities that did not handle memory usage even when it is severely needed because of cache usage. Cache usage is one of the main reasons why memory release mechanism is provided by the Android SDK [14]. Consequently, instances of *NLMR* in practice represent a real concern as developers manifest *technical unawareness* about the importance of memory management. Our analysis of memory release in Android also highlighted the presence of other mechanisms like the framework method `onTrimMemory()`. The current definition of *NLMR* only includes the `onLowMemory()` method and does not consider other alternatives. Hence, with the current definition, we may detect activities that implement the method `onTrimMemory()` as an instance of *NLMR*, which is *inaccurate*.

Instances of *NLMR* in practice can be real performance concerns. However, the current *NLMR* definition may cause false positives because of its focus on only one memory management mechanism.

6.2.4 HashMap Usage (HMU)

Sample

The *HMU* sample included 96 instances.

Definition

The usage of `HashMap` is inadvisable when managing small sets in Android. Using HashMaps entails the auto-boxing process where primitive types are converted into generic objects. The issue is that generic objects are much larger than primitive types, 16 *vs.* 4 bytes respectively. Therefore, the framework recommends using the `SparseArray` data structure, which is more memory-efficient [7, 169].

Relevant Context

According to the definition and sample instances, the manual analysis of *HMU* should cover the instruction that uses the `HashMap` and also the host method.

Instances Description

Key type	***Wrapper***	***Non-Wrapper***	***Undefined***
#Instances	13	72	11
%Instances	14%	75%	11%

Table 6.3 Distribution of *HMU* instances according to HashMap key type.

From our manual analysis, we found that an interesting insight may emerge from the type of the `HashMap` key in code smell instances. Hence, we built an analytical guide with the following category:

— Key type: the subcategories are Wrapper keys, Non-Wrapper keys, and undefined keys.

Following this guide, we open-coded the *HMU* instances. Table 6.3 reports the results of this open-coding. As shown in Table 6.3, the *HMU* instances have the following key types:

— *Wrapper Keys* (13 instances): This represents situations where the HashMap has a key of a wrapper class—*i.e.*, `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, or `Double`. In our sample, only 14% of the *HMU* instances have wrapper keys.

— *Non-wrapper keys* (72 instances): In this case, the keys of the `HashMap` containers were not of a wrapper class. 75% of the *HMU* instances of our sample have non-wrapper keys.

— *Undefined Keys* (11 instances): In these instances the `HashMap` containers were instantiated without defining the key type. This case includes 11% of the instances that we analyzed.

Relevant Literature

Based on the open-coding results, the relevant literature for our analysis is:
— The documentation of auto-boxing and its performance cost [151];
— Resources about the difference between `SparseArray` and `HashMap` [17].

Discussion

The essence of the *HMU* code smell is that HashMaps are inadvisable to avoid auto-boxing. In Java, the auto-boxing is the process of converting a primitive type, *e.g.*, `int`, into their corresponding object wrapper class, *e.g.*, `Integer`. When using HashMaps, the auto-boxing only happens when the used key is of a wrapper type. Our manual analysis showed that 75% of *HMU* instances do not have a wrapper key, thus they are not a real concern. This shows that the currently adopted definition of *HMU* is *inaccurate* and it should be reviewed to focus on wrapper keys.

The current definition of *HMU* is inaccurate because of its inclusion of HashMaps with non-wrapper keys. As a result of this inaccuracy, 75% of the instances in the analyzed sample were false positives.

6.2.5 UI Overdraw (UIO)

Sample

The *UIO* sample included 81 instances.

Definition

A UI Overdraw is a situation where a pixel of the screen is drawn many times in the same frame. This happens when the UI design consists of unneeded overlapping layers, *e.g.*, hiding backgrounds. To avoid such situations the method `clipRect()` or `quickReject()` should be called to define the view boundaries that are drawable [7, 169].

Relevant Context

Based on the definition and sample instances, the analysis should cover all the drawing operations in which the boundaries were omitted. This includes the Java source code of the view and the XML file which defines it.

Instances Description

From our manual analysis, we observed that *UIO* instances manifest in specific apps where customized views are drawn. We found that these drawings did not use `clipRect()`

and `quickReject()`, but adopted other means to avoid unnecessary drawing. Hence, we built an analytical guide with the following category:

— The mean for avoiding overdrawing: the subcategories are `get{width,height}`, `onSizeChanged`, `getMeasured{width,height}`, `getClipBounds`, and None.

We open-coded the instances using this analytical guide. Table 6.4 reports on the results of this open-coding.

Mean	`get{width,height}`	`onSizeChanged`	`getMeasured{width,height}`	`getClipBounds`	None
#Instances	43	26	3	2	7
%Instances	53%	32%	4%	2%	9%

Table 6.4 Alternative means used in *UIO* instances to avoid unnecessary drawings.

As shown in Table 6.4, *UIO* instances used the following alternative means to avoid overdrawing:

— `get{width,height}()`: These methods return, respectively, the real width and height of the view after drawing at runtime. These dimensions can be used to avoid drawing over other UI elements and thus avoid overdrawing.

— `onSizeChanged()`: In Android, the real size of the view is not predefined, it is calculated at runtime. This method is called when this calculation is done to allow developers, *inter alia*, to define their drawing boundaries.

— `getMeasured{width,height}`: These methods return, respectively, the theoretical width and height of the view, which can be used to limit the drawing area.

— `getClipBounds()`: This method returns the boundaries of the current clip in local coordinates. Similarly to the aforementioned methods, these boundaries can be used to avoid overdrawing.

— None: This represents *UIO* instances where no alternative means are used to avoid overdrawing.

During our manual analysis, we also observed the absence of layers in the views of the analyzed instances. Indeed, we did not find in the analyzed views a case where the UI is composed of layered views where overdrawing can be critical. We observed that in five instances, `clipRect()` is indirectly used via another method call but the tool did not detect the call and considered the case as a code smell instance.

Relevant Literature

Based on the open-coding results, the relevant literature for our analysis is the Android documentation about drawing operations and overdrawing [7, 19].

Discussion

The manual analysis showed that developers use different framework methods to define the view size and boundaries. While these methods do not perform exactly the same actions as the methods `clipRect()` and `quickReject()`, they still can be used to avoid useless drawing. This means that omitting `clipRect()` and `quickReject()` does not necessarily mean that the app has an overdrawing. This shows that the current definition of *UIO* is inaccurate and should consider possible alternatives. On top of that, the observations of the absence of layers and the undetected `clipRect()` calls show that the current detection technique of Paprika is limited.

The current definition of *UIO* omits the possibility of avoiding overdrawing with other framework methods. Thus, many instances of *UIO* are not real overdrawing cases.

6.2.6 Unsupported Hardware Acceleration (UHA)

Sample

The *UHA* sample included 74 instances.

Definition

In Android, most of the drawing operations are executed by the GPU. Drawing operations that are executed by the CPU (*e.g.*, `drawPath()` of `android.graphics.Canvas`) should be avoided to reduce CPU load [106, 143].

Relevant Context

According to the definition and sample instances, the analysis should cover all the drawing operations in which the hardware acceleration was not supported.

Instances Description

In our manual analysis, we noticed that there is a very little variation in the methods causing *UHA* instances. Indeed, we found that the unsupported hardware acceleration is mainly caused by `Path` drawing. Specifically, we found that 69 of the analyzed instances—*i.e.*, 93%, were caused by the methods `drawPath()`, `setPathEffect()`, and `drawTextOnPath()`. These methods are all related to the `Path` class, which allows to draw compound geometric shapes based on straight segments or curves.

Relevant Literature

Based on the open-coding results, the relevant literature for our analysis is the Android documentation about `Path` drawing [11, 16, 154].

Discussion

The analysis showed that *UHA* instances are mainly caused by path drawing. The problem is that methods like `drawPath()` has no alternatives in Android [154]. Thus, in case of need, developers cannot avoid these methods. Moreover, the impact of these methods on CPU performance depends heavily on the use case and the path size [16]. Hence, defining the simple usage of methods, like `drawPath()`, as a code smell instance can be exaggerating.

> 93% **of *UHA* instances are caused by drawings of the `Path` class, which does not have any alternatives in the Android framework.**

6.2.7 Init OnDraw (IOD)

Sample

The *IOD* sample included 51 instances.

Definition

This code smell is also known as DrawAllocation. It occurs when allocations are made inside `onDraw()` routines. The `onDraw()` methods are responsible for drawing `Views` and they are invoked 60 times per second. Therefore, allocations (*init*) should be avoided inside them to avoid memory churn [7, 106].

Relevant Context

Based on the definition and sample instances, the analysis should cover the instruction that performed the initialization and the `onDraw()` method hosting it.

Instances Description

We found that the major aspect that differentiates the instances is the nature of the allocated or initialized object. Hence, we considered this aspect in our analytical guide:

- The nature of the allocated object: the subcategories are drawing objects and other objects.

Based on this guide, we open-coded the sample instances. The results of this coding showed that in *IOD* instances, the allocated objects are of two types:

— Drawing objects (36 instances): We found that 71% of the instances were allocations of `Paint` and `Rect` objects. These classes are the basic components for any drawing in Android. They should be instantiated outside of the method `onDraw()` to avoid recreating them in every frame;

— Different objects (15 instances): We found that the remainder 29% instances were other objects of varying sizes. Indeed, we found cases where developers had a complex processing inside the `onDraw()` method. For instance, an instance in the Muzei app was a an `onDraw()` method with loops including objects creation [83]. This instance represents a severe case of *IOD* as it implies multiple allocations at each call of the `onDraw()` method. Another example is an instance in the Overchan app, which created a runnable inside the `onDraw()` method [84]. This means that the app is creating 60 runnables per second when the concerned view is visible. Beyond the memory impact, this practice heavily affects the CPU consumption and the app responsiveness.

Interestingly, we also found five instances of *IOD* where developers knew that they are violating the framework guidelines. As shown in this instance [81], the developers included an annotation to stop Android Lint from reporting the code smell. This shows that they were notified by the tool about the code smell but they decided to keep it in the codebase anyway.

Relevant Literature

Based on the open-coding results, the relevant literature for our analysis is the Android documentation about drawing and frames [20, 18].

Discussion

Our results showed that the severity of *IOD* instances can heavily vary depending on the allocated object. Indeed, creating drawing objects, *e.g.*, `Paint` and `Rect`, inside the `onDraw()` method can be a simplistic development design for developers. As the drawing occurs in the `onDraw()` method, developers may find it intuitive to allocate the drawing objects inside it. On the other hand, creating runnables and multiple objects inside the `onDraw()` method seems more severe as a practice. Indeed, this bad practice shows that developers were unaware of the drawing process and its performance implications.

The instances of *IOD* vary from allocations of simple drawing objects to massive allocations of heavy memory objects.

6.2.8 Unsuited LRU Cache Size (UCS)

Sample

The *UCS* sample included 18 instances.

Definition

This code smell occurs when an LRU cache is initialized without checking the available memory via the method `getMemoryClass()`. The available memory may vary considerably according to the device so it is necessary to adapt the cache size to the available memory [106, 137].

Relevant Context

Based on the definition and the sample instances, the analysis of *UCS* should cover the cache creation and the cached object.

Instances Description

For this code smell instances, we did not find any categorization criteria. However, we observed the following:

— In 50% of *UCS* instances, the cache was used for storing `Bitmap` objects;

— In 39% of the *UCS* instances, the developers tried to check the available memory using the runtime method `maxMemory()`. This method returns the maximum amount of memory that can be used. However, considering the system constraints, this amount may differ from the amount of memory that the app should use. The latter is rather available through the method `getMemoryClass()`.

Relevant Literature

Based on the open-coding results, the relevant literature for our analysis is the Android documentation about LRU caches [13].

Discussion

Our manual analysis showed that the code smell *UCS* relies on a complex technical detail that makes it hard to guess and avoid. It can be introduced even in cases where developers pay attention to performance concerns. Indeed, 50% of the code smell instances where caches of bitmap images. Using caches for bitmaps was initially a good performance practice, as bitmaps are large objects that can be heavy to load from disk or network. However, as the cache is created without checking the available memory, the practice becomes a code smell instance. This observation is confirmed by the cases where the method `maxMemory()` was used. Developers tried to define the cache size correctly but a technical difference between the two methods led to a code smell introduction.

The code smell *UCS* relies on a technical detail of the Android framework, which makes it hard to guess for developers.

6.2.9 Common Theory

In our discussion of the coding results of each code smell instances, we observed that the inaccuracy of code smell definitions is the most recurrent problem. Indeed, six of the eight studies code smells manifested in their instances flaws related to the adopted definition. We synthesize in Table 6.5 these flaws and their impact on code smell instances.

<table>
<tr><th>Code Smell</th><th>The definition omits</th><th>The definition leads to</th></tr>
<tr><td>No Low Memory Resolver</td><td rowspan="2">Alternative means
to avoid the code smell</td><td rowspan="3">False positives</td></tr>
<tr><td>UI Overdraw</td></tr>
<tr><td>Hashmap Usage</td><td>Technical details</td></tr>
<tr><td>Leaking Inner Class</td><td rowspan="2">Prevalent programming
designs and fashions</td><td rowspan="3">Instances that are:
• inevitable and/or
• hard to refactor</td></tr>
<tr><td>Member Ignoring Method</td></tr>
<tr><td>Unsupported Hardware Acceleration</td><td>The absence of alternatives</td></tr>
</table>

Table 6.5 Common flaws in code smell definitions.

As shown in Table 6.5, the definition flaws lead to two kinds of improper code smell instances:

- **False positives:** This represents instances that are generated by the code smell definition but do not represent a real performance concern. These false positives occur in two cases:
  - When the definition does not cover all the possible means to avoid the performance issue.
    - This flaw is present in the definition of *NLMR*, which does not account for other methods that can release unused memory, *e.g.*, `onTrimMemory()`;
    - This flaw also affects the definition of *UIO* as it does not consider the other framework methods that can be used to avoid overdrawing, *e.g.*, `getClipBounds()`;
  - When the definition is very generalized so it omits important technical details. This is the case of the definition of *HMU*, which considers all the usages of `HashMap` as performance issues and ignores that these structures are harmless when used with non-wrapper keys;
- **Confusing instances:** This represents situations where the code smell instance is inevitable and hard to refactor. These instances occur in two cases:
  - When the definition considers a prevalent programming deign or fashion as a performance issue.
    - This flaw is present in the *LIC* definition, which considers all anonymous classes as a memory leak;
    - This flaw also impacts the *MIM* definition, which considers that all empty and primitive methods must be static;

— When the definition considers an inevitable practice, which does not have alternatives, as a code smell instance. This is the case of the *UHA* definition, which considers any drawing of the `Path` class as a hardware acceleration despite the absence of alternatives to this class.

Six of the studied code smells manifest flaws in their definitions. These flaws lead to false positives and confusing instances that are difficult to avoid and refactor.

6.3 Discussion

We discuss in this section the implications of our study and the potential threats to its validity.

6.3.1 Implications

Based on our findings, the implications of our study are the following.

— Our results show that the currently adopted code smell definitions are flawed. Future studies of mobile-specific code smells must rectify the current definitions and adopt more precise ones.

— Our study shows that a common flaw in code smell definitions is the omission of programming fashions that are common among developers. This finding demonstrates the importance of involving developers in the process of defining code smells. We encourage future research works to validate their definitions and detection tools by developers.

— The flaws in the definitions of *NLMR*, *HMU*, and *UIO* lead to many false positives. This aligns with the complaints raised by developers in our linter study. Hence, we encourage researchers and tool makers to put more efforts into identifying false positives and avoiding them.

— Most of the identified flaws originated from the definitions of Reimann *et al.* [169]. This suggests that the semi-automated process of identifying code smells can lead to many inaccuracies. Hence, we encourage future studies to explore new approaches for identifying and precisely defining code smells.

— Our study shows that the techniques currently used by research tools for detecting *UIO* and *UHA* exhibit some inaccuracies that can hinder the study of these code smells. We invite future works to propose novel techniques and tools that perform a more accurate detection. In particular, the detection of *UIO* should take the UI design into consideration. Also, the detection of *UHA* should be more context-aware and consider the path size and execution time.

6.3.2 Threats to Validity

Transferability: One limitation to the generalizability of our study is the sample size. It would have been preferable to consider more code smell instances. However, the process of manual analysis is time and effort consuming, thus we had to limit the sample size. To limit the impact of this threat, we inductively built our theory from a representative sample of 599 code smell instances. This represents a 95 % statistically significant stratified sample with a 10 % confidence interval of the 180,013 code smell instances detected in a dataset of 324 Android apps. Another threat to generalizability is that our study only concerns eight Android-specific code smells. Without further investigation, these results should not be generalized to other code smells or mobile platforms. Another possible threat is that our study only inspected code smell instances in Android apps written in Java. Other programming languages are available for Android development, notably Kotlin, and it would be preferable to include instances from these languages. We therefore encourage future studies to replicate our work on other datasets and with different code smells, mobile platforms, and programming languages.

Confirmability: One possible threat to the assessment of our results could be the accuracy of our qualitative analysis and particularly the coding step. We used a consensual coding to alleviate this threat and strengthen the reliability of our conclusions. Indeed, each code smell instance was encoded by at least two authors. Initially, every author coded the instance independently to avoid influencing the analysis of other authors. Afterwards, the authors discussed and compared their coding results.

6.4 Summary

We presented in this chapter an exploratory study inspecting the characteristics of code smell instances. The results of this study revealed flaws in the established definitions for Android-specific code smells. In particular, we showed that:

— **Finding 1:** The code smell definitions for *No Low Memory Resolver*, *UI Overdraw*, and *HashMap Usage* omit critical technical details and generate many false positives;

— **Finding 2:** The current definitions of *Leaking Inner Class* and *Member Ignoring Method* consider prevalent practices related to prototyping and the usage of listeners as performance issues. This leads to confusing code smell instances that may be hard to refactor for developers;

— **Finding 3:** The definition of *Unsupported Hardware Acceleration* considers `Path` drawing, which does not have exact alternatives in Android, as a bad practice. This generates confusing code smell instances that may see inevitable for developers.

Our study also provided insights about flaws in code smell detection:

— **Finding 4:** The techniques currently used by research tools for detecting *UI Overdraw* and *Unsupported Hardware Acceleration* exhibit many flaws that can hinder the study of these code smells.

These findings have several implications on the research about Android code smells and performance in general. We highly encourage future works to consider these implications and propose novel catalogs and approaches for a better analysis of mobile apps. We provide all the data of this study in a comprehensible package [94] that can be leveraged in future studies.

Chapter 7

Conclusion and Perspectives

In this chapter, we summarize the contributions of our thesis and we discuss our short- and long-term perspectives.

7.1 Summary of Contributions

We presented in this thesis four contributions to the research in the field of code smells and mobile apps. These contributions provide actionable recommendations for researchers and tools makers who aim to propose solutions that limit code smells and improve the performance of mobile apps.

In Chapter 3, we presented an empirical study that explored code smells in the iOS platform. This study was the first to focus on mobile-specific code smells in a mobile platform other than Android. The first contribution of this study is a novel catalog of six iOS-specific code smells that we identified from developers discussions and the platform documentation. The study also proposed SUMAC, the first code smell detector for iOS apps. SUMAC is open-source and it detects seven code smells in apps written in Objective-C and Swift [97]. Using SUMAC and our catalog of iOS-specific code smells, we conducted an empirical study to explore code smells in the iOS platform. This empirical study, which covered 279 iOS apps and $1,551$ Android apps, showed that some code smells can be generalized to both mobile platforms Android and iOS. Also, it showed that Android apps tend to have more OO and mobile-specific code smells than iOS ones. Furthermore, the discussion of these results suggested that the presence of code smells is more related to the mobile platform than the programming language.

In Chapter 4, we presented our novel tool SNIFFER and three empirical studies about the presence of mobile-specific code smells in the change history. SNIFFER is a toolkit that precisely tracks the history of Android-specific code smells. It tackles many issues raised by the Git mining community like branch and renaming tracking [120]. Using SNIFFER, we performed three studies, (i) an evolution study, (ii) a developers study, and (iii) a survival

study. These large-scale studies covered eight Android code smells, 324 Android apps, $255k$ commits, and contributions from $4,525$ developers. The results of these studies allowed us to discern the motives behind the accrual of Android code smells. In particular, they showed that pragmatism and prioritization choices are not relevant motives of technical debt in the case of Android code smells. Moreover our findings suggest that individual attitudes of Android developers are not the reason behind the prevalence of code smells. The problem is rather caused by the ignorance and oversight, which seem to be common among developers. Finally, we found that the processes adopted by development teams—*e.g.*, the use of tools and their configuration can help in limiting the presence of code smells.

In Chapter 5, we presented a qualitative study about the usage of linters by Android developers. This study, which relied on interviews with 14 experienced Android developers, provided important insights about developers' perception of Android code smells. Specifically, it showed that there is a prevalent ignorance of mobile-specific code smells among developers. Moreover, developers affirmed that they prefer managing performance reactively instead of taking preventive measures. These developers also challenged some bad performance practices, qualifying them as irrelevant and insignificant performance-wise. The results of this study also provided indications about the usage of linters for performance concerns. In particular, we showed that linters can be used to anticipate and debug performance bottlenecks and they are also useful for raising performance culture among developers. Moreover, we showed that besides the intuitive proactive approach for using linters, developers can also use linters reactively in performance sprints. Finally, our analysis showed that the unclarity of categories and severity criteria can dissuade developers from using linters for performance concerns.

In Chapter 6, we presented a study inspecting the characteristics of mobile-specific code smell instances. Contrarily to our previous contributions, this study does not tackle one of the identified lacks in the literature. It was rather motivated by our findings in the qualitative study where developers questioned the relevance of some established bad practices. To address this questioning, we conducted an exploratory study relying on the well-founded approach Grounded Theory [90]. This approach allowed us to build a theory that provided second thoughts about the established foundation about code smell definition and detection. Specifically, it showed that many code smell instances are false positives or confusing instances that are hard to avoid and refactor. These instances are prevalent because the current definitions omit (i) critical technical details about the platform and (ii) programming fashions that are common among developers. The theory also showed that the techniques currently adopted by research tools for detecting code smells exhibit many flaws that can threaten the validity of the related research studies.

7.2 Short-Term Perspectives

In this section we present the short term perspectives of our thesis. We mentioned many perspectives throughout our presentation of the contributions and their implications. The objective of this section is to develop these perspectives and aggregate them.

7.2.1 Rectify Code Smell Definitions

In our linter study, we observed that many developers are dissatisfied with the current code smell definitions. Our evaluation of the code smell instances confirmed this observation and revealed that the currently adopted code smell definitions are flawed. Moreover, these flaws impacted six of the eight studied Android code smells. In our future studies, we aim to rectify these definitions by:

- Adding details that make the definitions more precise,
- Considering the absence of alternatives in the case of some code smells,
- Taking into account the prevalent development fashions among mobile developers.

This perspective is important for the research community since these definitions are adopted by different research studies. Hence, the rectification of these definitions and their original catalog are necessary to assure the validity of future studies about mobile-specific code smells.

7.2.2 Improve Code Smell Detection

Our study showed that the current code smell detection techniques are inaccurate for cases like *UI Overdraw* and *Unsupported Hardware Acceleration*. In our future works, we intend to propose novel techniques for code smell detection. In particular, we aim to augment the current detectors, which are mainly static analyzers, with dynamic analysis. Dynamic analysis can be a better option for code smells that describe execution scenarios. For instance, the detection of the code smell *Unsupported Hardware Acceleration* would be more accurate if it considers the path size, which is generally determined at runtime. This improvement also aligns with the complaints raised by developers about the inadequacy of static analyzers for some performance concerns.

Furthermore, dynamic analysis allows to overcome constraints that we commonly faced in our studies. Specifically, in our different code smell studies, we were always restricted by the available open-source apps. The implementation of dynamic analysis allows to analyze the binaries of mobile apps. Hence, we would be able to cover proprietary software and therefore include more apps in our datasets.

7.2.3 Reevaluate the Impact of Code Smells

A recurrent concern raised by our studies is the real impact of mobile-specific code smells. Many developers claimed that some mobile-specific code smells are insignificant performance-wise. Also, the unawareness observed in our developers study incited us to question the impact of these code smells. Indeed, if these code smells effectively impacted apps in practice, the performance degradation would have attracted developers awareness. For these reasons, we intend in our future works to inspect the impact of mobile-specific code smells on the perceived app performance. Previous studies already assessed the impact of these code smells on performance and energy consumption [40, 107, 157]. Nonetheless, the studies of Carette *et al.* [40] and Palomba *et al.* [157] based most of their claims on cases of combined code smell instances. Also, the study of Hecht *et al.* [107] did not always show important performance impact. Finally, all these studies relied on technical metrics without interpreting the relationship between these metrics and the perceived app performance and the user experience. Our objective is to complement these studies by adopting user-based scenarios and evaluating the impact of mobile-specific code smells on diverse metrics including user experience.

7.2.4 Integrate a Code Smell Detector in the Development Environment

In our future works, we aim to work on the integration of a code smell detector in the development environments commonly used by developers, *e.g.*, Android Studio. In this integration, we intend to carefully follow the recommendations proposed in our linter study. Indeed, our qualitative study proposed the following actionable recommendations:

- Linters must be easy to use and they should blend in the development workflow seamlessly;
- Linters should be more clear and explicit about the categories of checks, the descriptions, and the performance impact of the issues;
- Linters should provide the possibility to write new detection rules. The writing and integration of the new rules should be simple to motivate developers.

Our objective from the integration and these recommendations is to reach developers and help them improve the quality of their mobile apps. In addition, the integration of a code smell detector can help us in communicating about code smells and their impact.

7.2.5 Communicate About Code Smells

A common result of our studies is that there is a prevalent ignorance of mobile-specific code smells among developers. This ignorance implies that the current communication channels about code smells are ineffective. Hence, we suggest that platform and tool makers should communicate more about good and bad development practices. Also, as researchers,

we should put more efforts into communicating the results of our studies to developers. This communication should be less formal and more simplified to successfully reach developers.

In addition, we aim to investigate the impact of channels like documentation and tool descriptions on the perception of code smells. The objective of this investigation is to identify the needed improvements for these channels to enable a better communication about different development issues.

7.3 Long-Term Perspectives

One of the main findings of our thesis is that the current code smell definitions are flawed. In our short-term perspectives, we proposed to rectify the definitions as a first step for alleviating these flaws. However, the rectification does not address the root cause of the issue, which is the identification of code smells. In the following, we present our perspectives for improving the process of identifying code smells in particular, and development practices in general.

7.3.1 Involve Developers in Code Smell Identification

As shown in our literature review, studies relied on different resources to identify bad practices for mobile development. In particular, they relied on:

- documentation,
- developer blogs,
- developer discussions,
- issue reports,
- and source code.

The developer blogs and discussions allowed the identification studies to include the developers' feedback in their code smells. However, this inclusion was limited by the focus on data available online like blogs and discussions. Specifically, none of these studies opted for qualitative approaches to gather data directly from developers.

In our future studies, we intend to involve developers actively in the process of code smell identification. This involvement is important to gather data that may not be available in online discussions and blogs. Moreover, involved developers can incrementally validate the results of the identification process to ensure that the defined code smells are conform to the community needs. We also believe that the expertise of developers can help substantially in improving the quality of the identified code smells. For instance, developers can highlight more abstract practices that reflect their feedback about the development experience. These abstract practices can be a better option than the current code smell definitions. The current code smells are framework-specific and thus they sometimes change or become deprecated with framework updates and upgrades. For instance, we found that the code

smells *Invalidate Without Rect* and *Internal Getter Setter*, which are used in different research studies [40, 107, 156, 157], are now deprecated. The method `invalidate()` was deprecated in API level 28 implying the deprecation of the code smell *Invalidate Without Rect* [8]. Also, since API level 10, internal getters are automatically optimized by Dalvik and thus the code smell *Internal Getter Setter* is not valid anymore [7]. By favoring more abstract code smells, the research community can avoid these deprecations. Hence, we can conduct studies that are more generalizable within the studied platform and also across different development frameworks. Finally, developers can help in defining the severity and priority of the identified code smells. This would allow tool makers to adjust their tools to the perception and needs of developers.

7.3.2 Leverage Big Code in Code Smell Identification

Most of the studies that identified mobile-specific code smells followed a top-down approach. That is, they defined their code smells based on external resources like documentation, discussions, and blogs to build their catalogs and analyze the source code. An alternative to this process is the bottom-up approach in which code smells are defined from the source code itself. The studies of Guo *et al.* [92] and Zhang *et al.* [208] followed this approach, as they profiled and manually analyzed the source code of mobile apps to identify bad practices. However, these studies did not rely on large software repositories, only 98 and 15 apps, respectively. In our future studies, we aim to leverage the source code of the thousands of mobile apps, which are available in the app stores, to learn new development practices. In particular, we aim to combine machine learning techniques and crowd-sourcing to identify new development practices. This perspective entails three main challenges:

— **Define a representation for mobile apps:** Our objective is to define a data structure that represents the source code of mobile apps and that is suitable for unsupervised learning techniques. The literature proposes a few representations for mobile-specific code smells. Notably, Paprika adopts a graph representation where nodes are the source code entities (*e.g.*, classes, methods, variables) and edges are the different relationships between them (*e.g.*, ownership, coupling, etc). We conducted some experiments on these graphs and our initial results showed that these representations are not adequate for learning techniques. Specifically, the algorithms generally expect mono-type nodes and edges, which is not the case of the Paprika graph. In our future works, we intend to explore other representations that fit with the learning algorithms and also express the different specificities of mobile apps;

— **Design a detection technique:** We aim to build a technique that unsupervisedly identifies patterns from the source code of mobile apps. We have already conducted experiments with Graph Edit Distance [70] and identified redundant patterns in the source code. Graph Edit Distance is a pure pattern recognition method. In the next

steps, we aim to experiment with impure and extreme pattern recognition techniques, like Learning Vector Quantization [119] and Graph Embedding [204];

— **Design an assessment oracle:** Our aim is to build an oracle that evaluates the relevance and the quality of the identified patterns. For this purpose, we intend to rely on a multi-objective approach that considers the performance impact, the software quality metrics, and the usefulness of the pattern in practice. To define the performance and quality metrics, we will build on previous works about mobile and software systems in general. As for the usefulness, we will adopt a crowd-sourcing approach where developers will assess the practicality of the patterns and their usefulness in development.

7.4 Final Words

This thesis summarizes the research works that we conducted during three years in order to build an understanding of mobile-specific code smells. We believe that our works contributes effectively in developing this understanding and helps in designing adequate solutions for mobile-specific code smells. Our work also opens many perspectives for both researchers and tool makers. We summarize in Table 7.1 these perspectives.

#	***Perspectives***
A	Rectify code smell definitions
B	Improve code smell detection
C	Reevaluate the impact of code smells
D	Integrate a code smell detector in the development environment
E	Communicate about code smells
F	Involve developers in code smell identification
G	Leverage big code in code smell identification

Table 7.1 The short and long-term perspectives of our thesis

Bibliography

[1] Aalen, O. O. (1989). A linear regression model for the analysis of life times. *Statistics in medicine*, 8(8):907–925.

[2] Abbes, M., Khomh, F., Guéhéneuc, Y. G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 181–190.

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

[4] Afjehei, S. S., Chen, T.-H. P., and Tsantalis, N. (2019). iperfdetector: Characterizing and detecting performance anti-patterns in ios applications. *Empirical Software Engineering*.

[5] Agiletribe (2013). Don't abuse singleton pattern. `https://agiletribe.wordpress.com/2013/10/08/dont-abuse-singleton-pattern/`. accessed: 2017-01-06.

[6] Akamai (2015). Mobile stats. `https://www.soasta.com/blog/22-mobile-web-performance-stats/`. [Online; accessed April-2018].

[7] Android (2017a). Android lint checks. `https://sites.google.com/a/android.com/tools/tips/lint-checks`. [Online; accessed August-2017].

[8] Android (2017b). Deprecation of invalidate with rect. `https://developer.android.com/reference/android/view/View.html#invalidate()`. [Online; accessed January-2019].

[9] Android (2018). Android lint. `https://developer.android.com/studio/write/lint.html`. [Online; accessed March-2018].

[10] Android (2019a). Android versioning. `https://developer.android.com/studio/publish/versioning`. [Online; accessed January-2019].

[11] Android (2019b). Canvas. `https://developer.android.com/reference/android/graphics/Canvas`. [Online; accessed September-2019].

[12] Android (2019c). Input events overview. `https://developer.android.com/guide/topics/ui/ui-events`. [Online; accessed September-2019].

[13] Android (2019d). Lrucache. `https://developer.android.com/reference/android/util/LruCache`. [Online; accessed September-2019].

[14] Android (2019e). OnLowMemory() and caches. [Online; accessed January-2019].

[15] Android (2019f). ontrimmemory. `https://developer.android.com/reference/android/content/ComponentCallbacks2#onTrimMemory(int)`. [Online; accessed September-2019].

[16] Android (2019g). Slow rendering. https://developer.android.com/topic/performance/vitals/render. [Online; accessed September-2019].

[17] Android (2019h). Sparsearray family ties. https://www.youtube.com/watch?v=I16lz26WyzQ. [Online; accessed September-2019].

[18] Android (2019i). Test ui performance. https://developer.android.com/training/testing/performance. [Online; accessed September-2019].

[19] Android (2019j). View. https://developer.android.com/reference/android/view/View. [Online; accessed September-2019].

[20] Android (2019k). Why 60fps? https://www.youtube.com/watch?v=CaMTIgxCSqU&index=25&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE. [Online; accessed September-2019].

[21] Apple (2014). Threading programming guide. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Multithreading/Introduction/Introduction.html. accessed: 2017-01-06.

[22] Apple (2015a). Performance tips. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/PerformanceTips/PerformanceTips.html. accessed: 2017-01-06.

[23] Apple (2015b). The Singleton pattern in Cocoa. https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Singleton.html#//apple_ref/doc/uid/TP40008195-CH49-SW1. accessed: 2016-10-20.

[24] Apple (2015c). Work less in the background. https://developer.apple.com/library/ios/documentation/Performance/Conceptual/EnergyGuide-iOS/WorkLessInTheBackground.html#//apple_ref/doc/uid/TP40015243-CH22-SW1. accessed: 2016-10-20.

[25] Apple (2017). Apple developer documentation. https://developer.apple.com/develop/. accessed: 2017-01-09.

[26] Arcelli Fontana, F. and Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324.

[27] Arcoverde, R., Garcia, A., and Figueiredo, E. (2011). Understanding the longevity of code smells: preliminary results of an explanatory survey. In *Proceedings of the 4th Workshop on Refactoring Tools*, pages 33–36. ACM.

[28] Arturbosch (2018). Detekt. https://github.com/arturbosch/detekt. [Online; accessed April-2018].

[29] Ash, F. (2014). Introduction to mvvm. https://www.objc.io/issues/13-architecture/mvvm/. accessed: 2017-01-06.

[30] Atom (2016). Linter swiftc. https://atom.io/packages/linter-swiftc. accessed: 2017-01-09.

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

[32] Baltes, S. and Diehl, S. (2018). Towards a theory of software development expertise. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 187–200. ACM.

[33] Barbaschow, A. (2019). Smartphone market and tablet sales. `https://www.zdnet.com/article/smartphone-market-a-mess-but-annual-tablet-sales-are-also-down/`. [Online; accessed September-2019].

[34] Bartel, A., Klein, J., Le Traon, Y., and Monperrus, M. (2012). Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proc. of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM.

[35] Beller, M., Bholanath, R., McIntosh, S., and Zaidman, A. (2016). Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 470–481. IEEE.

[36] Bland, J. M. and Altman, D. G. (1998). Survival probabilities (the kaplan-meier method). *Bmj*, 317(7172):1572–1580.

[37] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., et al. (2010). Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM.

[38] Brown, W. H., Malveau, R. C., McCormick, H. W. S., and Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition.

[39] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. (2015). Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer.

[40] Carette, A., Younes, M. A. A., Hecht, G., Moha, N., and Rouvoy, R. (2017). Investigating the energy impact of android smells. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 115–126. IEEE.

[41] Carroll, A. and Heiser, G. (2010). An analysis of power consumption in a smartphone. *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 21–21.

[42] Cast (2017). Reduce technical debt. https://www.castsoftware.com/research-labs/technical-debt. [Online; accessed January-2019].

[43] Championswimmer (2015). Commit from open event android. `https://github.com/fossasia/open-event-android/commit/88105bdd88eee3f1222b1f25bc76d5ef4125707a`. [Online; accessed December-2018].

[44] Charmaz, K. and Belgrave, L. L. (2007). Grounded theory. *The Blackwell encyclopedia of sociology*.

[45] CheckStyle (2018). Checkstyle. `http://checkstyle.sourceforge.net/`. [Online; accessed April-2018].

[46] Chris, E. (2013). Lighter view controllers. `https://www.objc.io/issues/1-view-controllers/lighter-view-controllers/`. accessed: 2017-01-06.

[47] Christakis, M. and Bird, C. (2016). 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. ACM.

[48] Cingolani, P. and Alcalá-Fdez, J. (2013). jfuzzylogic: a java library to design fuzzy logic controllers according to the standard for fuzzy control programming. *Int. J. Comput. Intell. Syst.*, 6:61–75.

[49] Clang (2016). Clang static analyzer. `http://clang-analyzer.llvm.org/`. accessed:2016-10-20.

[50] Cliff, N. (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494.

[51] Cohen, J. (1977). *Statistical power analysis for the behavioral sciences (rev.* Lawrence Erlbaum Associates, Inc.

[52] Cohen, J. (1992). A power primer. *Psychological bulletin*, 112(1):155.

[53] Colin, W. (2011). Singletons: You're doing them wrong. `http://cocoasamurai.blogspot.fr/2011/04/singletons-your-doing-them-wrong.html`. accessed: 2017-01-06.

[54] Cook, T. and Campbell, D. (1979). *Quasi-experimentation: Design & analysis issues for field settings*. Houghton Mifflin Company.

[55] Cox, D. R. (1972). Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, 34(2):187–202.

[56] Creswell, J. W. and Creswell, J. D. (2017). *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.

[57] Crowd (2016). Collaborative list of open-source ios apps. `https://github.com/dkhamsing/open-source-ios-apps`. accessed: 2016-10-20.

[58] Cunningham, W. (1993). The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30.

[59] da Silva Maldonado, E., Shihab, E., and Tsantalis, N. (2017). Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062.

[60] Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. (2012). Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM.

[61] Davidson-Pilon, C. (2014). Lifelines. `https://lifelines.readthedocs.io/en/latest/Survival%20Analysis%20intro.html`. [Online; accessed January-2019].

[62] Dazeinfo (2016). Mobile app retention challenge. `https://dazeinfo.com/2016/05/19/mobile-app-retention-churn-rate-smartphone-users/`. [Online; accessed April-2018].

[63] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. (2011). PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*.

[64] ESLint (2018). Eslint. https://eslint.org/. [Online; accessed April-2018].

[65] FauxPas (2016). Faux pas. http://fauxpasapp.com/#highlights. accessed: 2017-01-09.

[66] FindBugs (2018). Findbugs. http://findbugs.sourceforge.net/. [Online; accessed April-2018].

[67] Fontana, F. A., Ferme, V., and Spinelli, S. (2012). Investigating the impact of code smells debt on quality code evaluation. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 15–22. IEEE Press.

[68] Forsblom, N. (2019). Sensors in a smartphone. https://blog.adtile.me/2015/11/12/were-you-aware-of-all-these-sensors-in-your-smartphone/. [Online; accessed September-2019].

[69] Fowler, M. (1999). *Refactoring: improving the design of existing code*. Pearson Education India.

[70] Gao, X., Xiao, B., Tao, D., and Li, X. (2010). A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129.

[71] Geiger, F.-X., Malavolta, I., Pascarella, L., Palomba, F., Di Nucci, D., and Bacchelli, A. (2018). A graph-based dataset of commit history of real-world android apps. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 30–33. ACM.

[72] Geoffrey, H. (2014). Paprika source code. https://github.com/GeoffreyHecht/paprika. [Online; accessed January-2019].

[73] GitHub (2008). Interface mim. https://github.com/k9mail/k-9/blob/ffd7766384f89d54e1502a6e7e7559c24ebcfdd0/src/com/android/email/MessagingListener.java. [Online; accessed January-2019].

[74] GitHub (2009). Mention memory improvement. https://github.com/k9mail/k-9/commit/909f677f912ed1a01b4ef39f2bd7e6b068d1f19e. [Online; accessed January-2019].

[75] GitHub (2010). Fix lic. https://github.com/connectbot/connectbot/commit/32bc0edb89e708b873533de94d3e58d5099cc3ba. [Online; accessed January-2019].

[76] GitHub (2012a). Remove nlmr. https://github.com/SilenceIM/Silence/commit/3d9475676f80a3dbd1b29f83c59e2c132fb135b5. [Online; accessed January-2019].

[77] GitHub (2012b). Remove nlmr with modifications. https://github.com/k9mail/k-9/commit/bbcc4988ba52ca5e8212a73444913d35c23cebc4. [Online; accessed January-2019].

[78] GitHub (2013a). Abstract class mim. https://github.com/nilsbraden/ttrss-reader-fork/blob/9de6e6257a0feab73d23a9df570db0c448772dae/ttrss-reader/src/org/ttrssreader/utils/AsyncTask.java. [Online; accessed January-2019].

[79] GitHub (2013b). Empty method mim. `https://github.com/y20k/transistor/blob/56101672d6ff528a48c73df23789bc5ea51dddf7/libraries/ExoPlayer-r2.5.4/library/smoothstreaming/src/main/java/com/google/android/exoplayer2/source/smoothstreaming/manifest/SsManifestParser.java`. [Online; accessed January-2019].

[80] GitHub (2013c). Remove lic. `https://github.com/haiwen/seadroid/commit/74112f7acba3511a650e113aa3483dcd215af88f`. [Online; accessed January-2019].

[81] GitHub (2013d). Suppress draw allocation. `https://github.com/klassm/andFHEM/blob/d30e276494afe2a7a069de93a34b14686ea1eb7c/app/src/main/java/com/chiralcode/colorpicker/ColorPicker.java`. [Online; accessed January-2019].

[82] GitHub (2014a). Commit from shopping list app. `https://github.com/openintents/shoppinglist/commit/efa2d0b2214349c33096d1d999663585733ec7b7#diff-7004284a32fa052399e3590844bc917f`. [Online; accessed December-2018].

[83] GitHub (2014b). Complex ondraw. `https://github.com/romannurik/muzei/blob/82d3b0f223adb1a407b080c19414897da80ade6a/main/src/main/java/com/google/android/apps/muzei/util/AnimatedMuzeiLogoView.java`. [Online; accessed January-2019].

[84] GitHub (2015). Runnable in ondraw. `https://github.com/miku-nyan/Overchan-Android/blob/efcac93b9d9ba7fe3c5d92a80ba92299b7d3f704/src/nya/miku/wishmaster/lib/gallery/FixedSubsamplingScaleImageView.java`. [Online; accessed January-2019].

[85] GitHub (2017). Primitive method mim. `https://github.com/reloZid/android-anuto/tree/c4374d5785b5e24657a21b78ae5a14d8a8af6821/app/src/main/java/ch/logixisland/anuto/engine/logic/Entity.java`. [Online; accessed January-2019].

[86] Gitster (2016). Git diff core delta algorithm. `https://github.com/git/git/blob/6867272d5b5615bd74ec97bf35b4c4a8d9fe3a51/diffcore-delta.c`. [Online; accessed November-2018].

[87] Gjoshevski, M. and Schweighofer, T. (2015). Small Scale Analysis of Source Code Quality with regard to Native Android Mobile Applications. In *4th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications*, pages 2–10, Maribor, Slovenia. CEUR Vol. 1375.

[88] Glaser, B. G. and Holton, J. (2007). Remodeling grounded theory. *Historical Social Research/Historische Sozialforschung. Supplement*, pages 47–68.

[89] Glaser, B. G., Strauss, A. L., et al. (1998). Grounded theory. *Strategien qualitativer Forschung. Bern: Huber*, 4.

[90] Glaser, B. G., Strauss, A. L., and Strutzel, E. (1968). The discovery of grounded theory; strategies for qualitative research. *Nursing research*, 17(4):364.

[91] Gottschalk, M., Jelschen, J., and Winter, A. (2014). Saving Energy on Mobile Devices by Refactoring. *28th International Conference on Informatics for Environmental Protection (EnviroInfo 2014)*.

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

[93] Habchi, S. (2019a). Companion artifacts for developers study. `https://figshare.com/s/26d66505a75a772dd994`. [Online; accessed September-2019].

[94] Habchi, S. (2019b). Exploratory study of code smell instances. `https://figshare.com/s/2711b65e456952d44150`. [Online; accessed September-2019].

[95] Habchi, S. (2019c). Study artifacts for evolution study. `https://figshare.com/s/790170a87dd81b184b0a`. [Online; accessed September-2019].

[96] Habchi, S. (2019d). Study artifacts for survival study. `https://figshare.com/s/9977eeabd9265d713bdb`. [Online; accessed September-2019].

[97] Habchi, S. (2019e). Sumac source code. `https://github.com/HabchiSarra/Sniffer`. [Online; accessed September-2019].

[98] Habchi, S., Blanc, X., and Rouvoy, R. (2018a). On adopting linters to deal with performance concerns in android apps. In *ASE18-Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, volume 11. ACM Press.

[99] Habchi, S., Blanc, X., and Rouvoy, R. (2018b). Technical report for linter study. `https://zenodo.org/record/1320453`.

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

[101] Habchi, S., Moha, N., and Rouvoy, R. (2019a). The rise of android code smells: Who is to blame? In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, pages 445–456, Piscataway, NJ, USA. IEEE Press.

[102] Habchi, S., Rouvoy, R., and Moha, N. (2019b). Database of full code smell history. `https://figshare.com/s/d9e1782ba36dac7b780b`. [Online; accessed January-2019].

[103] Habchi, S., Rouvoy, R., and Moha, N. (2019c). On the survival of android code smells in the wild. In *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '19, pages 87–98, Piscataway, NJ, USA. IEEE Press.

[104] Habchi, S. and Veuiller, A. (2019). Sniffer source code. `https://github.com/HabchiSarra/Sniffer`. [Online; accessed January-2019].

[105] Harrington, D. P. and Fleming, T. R. (1982). A class of rank test procedures for censored survival data. *Biometrika*, 69(3):553–566.

[106] Hecht, G. (2017). *Détection et analyse de l'impact des défauts de code dans les applications mobiles*. PhD thesis, Université du Québec à Montréal, Université de Lille, INRIA.

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

[108] Hecht, G., Omar, B., Rouvoy, R., Moha, N., and Duchien, L. (2015a). Tracking the software quality of android applications along their evolution. In *30th IEEE/ACM International Conference on Automated Software Engineering*, page 12. IEEE.

[109] Hecht, G., Rouvoy, R., Moha, N., and Duchien, L. (2015b). Detecting Antipatterns in Android Apps. Research Report RR-8693, INRIA Lille ; INRIA.

[110] Hector, M. (2015). Defeating the antipattern bully. `https://krakendev.io/blog/antipatterns-singletons`. accessed: 2017-01-06.

[111] Hove, S. E. and Anda, B. (2005). Experiences from conducting semi-structured interviews in empirical software engineering research. In *Software metrics, 2005. 11th ieee international symposium*, pages 10–pp. IEEE.

[112] Infer, F. (2018). Infer. `http://fbinfer.com/`. [Online; accessed April-2018].

[113] Jeff, G. and Conrad, S. (2014). Architecting ios apps with viper. `https://www.objc.io/issues/13-architecture/viper/`. accessed: 2016-10-20.

[114] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press.

[115] Kessentini, M. and Ouni, A. (2017). Detecting android smells using multi-objective genetic programming. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 122–132. IEEE Press.

[116] Khomh, F., Di Penta, M., and Gueheneuc, Y.-G. (2009). An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 75–84. IEEE.

[117] Kim, M. and Notkin, D. (2006). Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64. ACM.

[118] Kleinbaum, D. G. and Klein, M. (2010). *Survival analysis*, volume 3. Springer.

[119] Kohonen, T. (1995). Learning vector quantization. In *Self-organizing maps*, pages 175–189. Springer.

[120] Kovalenko, V., Palomba, F., and Bacchelli, A. (2018). Mining file histories: should we consider branches? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM*, pages 202–213.

[121] Krutz, D. E., Mirakhorli, M., Malachowsky, S. A., Ruiz, A., Peterson, J., Filipski, A., and Smith, J. (2015). A dataset of open-source android applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 522–525. IEEE Press.

[122] Kyle, C. (2013). Why singletons are bad. `http://www.kyleclegg.com/blog/9272013why-singletons-are-bad`. accessed: 2017-01-06.

[123] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D. (2014a). 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. ACM.

[124] Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D., and Guéhéneuc, Y.-G. (2014b). Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In *Proc. of the 22nd International Conference on Program Comprehension*, pages 232–243. ACM.

[125] Linares-Vásquez, M., Vendome, C., Luo, Q., and Poshyvanyk, D. (2015). How developers detect and fix performance bottlenecks in android apps. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 352–361. IEEE.

[126] Liu, Y., Xu, C., and Cheung, S.-C. (2013). Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, pages 2–10. IEEE.

[127] Liu, Y., Xu, C., and Cheung, S.-C. (2014). Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM.

[128] Macbeth, G., Razumiejczyk, E., and Ledesma, R. D. (2011). Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10(2):545–555.

[129] Mannan, U. A., Ahmed, I., Almurshed, R. A. M., Dig, D., and Jensen, C. (2016). Understanding code smells in android applications. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, pages 225–234. ACM.

[130] Marcelo, F. (2013a). 25 ios app performance tips and tricks. `http://www.raywenderlich.com/31166/25-ios-app-performance-tips-tricks#memwarnings`. accessed: 2017-01-06.

[131] Marcelo, F. (2013b). 25 ios app performance tips and tricks. `https://www.raywenderlich.com/31166/25-ios-app-performance-tips-tricks#mainthread`. accessed: 2017-10-06.

[132] Marcelo, F. (2013c). 25 ios app performance tips and tricks. `https://www.raywenderlich.com/31166/25-ios-app-performance-tips-tricks#cache`. accessed: 2017-01-06.

[133] Marcus, Z. (2015). Massive view controller. `http://www.cimgf.com/2015/09/21/massive-view-controllers/`. accessed: 2017-01-06.

[134] Mateus, B. G. and Martinez, M. (2018). An empirical study on quality of android applications written in kotlin language. *Empirical Software Engineering*, pages 1–38.

[135] Matteo, M. (2014). How to keep your view controllers small for a better code base. `http://matteomanferdini.com/how-to-keep-your-view-controllers-small-for-a-better-code-base/`. accessed: 2017-01-06.

[136] Maxwell, K. (2002). *Applied statistics for software managers.* Prentice Hall.

[137] McAnlis, C. (2015). The magic of lru cache (100 days of google dev). `https://youtu.be/R5ON3iwx78M`. [Online; accessed January-2019].

[138] McIlroy, S., Ali, N., and Hassan, A. E. (2016). Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empirical Software Engineering*, 21(3):1346–1370.

[139] Moha, N., Guéhéneuc, Y.-G., Duchien, L., and Le Meur, A. (2010). Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36.

[140] Morales, R., Saborido, R., Khomh, F., Chicano, F., and Antoniol, G. (2017). Earmo: An energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*.

[141] Neo4J (2017). Cypher. `http://neo4j.com/developer/cypher-query-language`. [Online; accessed August-2017].

[142] Neo4j (2017). Neo4j. `http://neo4j.com`. [Online; accessed August-2017].

[143] Ni-Lewis, I. (2015). Custom views and performance (100 days of google dev). `https://youtu.be/zK2i7ivzK7M`. [Online; accessed January-2019].

[144] NilsBraden (2011). Full mim instance. `https://github.com/nilsbraden/ttrss-reader-fork/tree/5e9c3bc75ad23f30f2a1ebceffcbe2072e4b564a/ttrss-reader/src/org/ttrssreader/model/CategoryAdapter.java`. [Online; accessed January-2019].

[145] Nittner, G. (2016). Chanu - 4chan android app. `https://github.com/grzegorznittner/chanu`. [Online; accessed January-2019].

[146] Objc.io (2017). Objc.io. `https://www.objc.io/`. accessed: 2017-01-09.

[147] OCLint (2016). Oclint. `http://oclint.org/`. accessed:2016-04-29.

[148] Oliver, D. G., Serovich, J. M., and Mason, T. L. (2005). Constraints and opportunities with interview transcription: Towards reflection in qualitative research. *Social forces*, 84(2):1273–1289.

[149] OpenTheFile (2015). What is an ipa file and how do i open an ipa file? `http://www.openthefile.net/extension/ipa`. accessed: 2016-05-17.

[150] Oracle (2017a). Anonymous Classes. `https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html`. [Online; accessed January-2019].

[151] Oracle (2017b). Autoboxing. `https://docs.oracle.com/javase/8/docs/technotes/guides/language/autoboxing.html`. [Online; accessed September-2019].

[152] Oracle (2017c). Java platform, micro edition (java me). `http://www.oracle.com/technetwork/java/embedded/javame/index.html`. [Online; accessed August-2017].

[153] Oracle (2017d). Nested classes. `https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`. [Online; accessed September-2019].

[154] Overflow, S. (2104). drawpath() alternatives. [Online; accessed January-2019].

[155] Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., and Lucia, A. D. (2014). Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110.

[156] Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., and De Lucia, A. (2017a). Lightweight detection of android-specific code smells: The adoctor project. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 487–491. IEEE.

[157] Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., and De Lucia, A. (2019). On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, 105:43–55.

[158] Palomba, F., Nucci, D. D., Panichella, A., Zaidman, A., and Lucia, A. D. (2017b). Lightweight detection of android-specific code smells: The adoctor project. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017*, pages 487–491. IEEE Computer Society.

[159] Pathak, A., Hu, Y. C., and Zhang, M. (2011). Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 5. ACM.

[160] Pawlak, R. (2006). Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11).

[161] Peters, R. and Zaidman, A. (2012). Evaluating the lifespan of code smells using software repository mining. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 411–416. IEEE.

[162] Phonegg (2019a). Highest internal memory. `https://www.phonegg.com/top/61-Highest-Internal-Memory`. [Online; accessed September-2019].

[163] Phonegg (2019b). Highest ram. `https://www.phonegg.com/top/60-Highest-RAM`. [Online; accessed September-2019].

[164] PMD (2017). Pmd source code analyzer. `http://pmd.sourceforge.net/`.

[165] Raymond, L. (2015). Clean swift ios architecture for fixing massive view controller. `http://clean-swift.com/clean-swift-ios-architecture/`. accessed: 2017-01-06.

[166] Raywenderlich (2016). Raywenderlich. `https://www.raywenderlich.com/`. accessed: 2017-01-09.

[167] Realm (2016). Swift lint. `https://github.com/realm/SwiftLint`. accessed: 2017-01-09.

[168] Reddit (2018). Android dev subreddit. `https://www.reddit.com/r/androiddev/`. [Online; accessed April-2018].

[169] Reimann, J., Brylski, M., and Aßmann, U. (2014). A Tool-Supported Quality Smell Catalogue For Android Developers. In *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung – MMSM 2014*.

[170] Romano, J., Kromrey, J. D., Coraggio, J., and Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33.

[171] Sadowski, C., Van Gogh, J., Jaspan, C., Söderberg, E., and Winter, C. (2015a). Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 598–608. IEEE Press.

[172] Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., and Winter, C. (2015b). Tricorder: Building a program analysis ecosystem. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 1:598–608.

[173] Schmidt, C. (2004). The analysis of semi-structured interviews. *A companion to qualitative research*, pages 253–258.

[174] Sheskin, D. J. (2003). *Handbook of parametric and nonparametric statistical procedures*. crc Press.

[175] Shyiko (2018). Ktlint. https://github.com/shyiko/ktlint. [Online; accessed April-2018].

[176] Silence (2013). Renaming example from silence app. https://github.com/SilenceIM/Silence/commit/303d1acd4554d87d920a265fec7cc167a8bb094e. [Online; accessed January-2019].

[177] SonarQube (2015). Sonarqube. http://www.sonarqube.org/. accessed: 2016-10-20.

[178] Soroush, K. (2015). Massive view controller. http://khanlou.com/2015/12/massive-view-controller/. accessed: 2017-01-06.

[179] stackoverflow (2017). Stack overflow. http://stackoverflow.com/. accessed: 2017-01-09.

[180] Statista (2019a). Apps in leading stores. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. [Online; accessed September-2019].

[181] Statista (2019b). Downloads from the apple store. https://www.statista.com/statistics/263794/number-of-downloads-from-the-apple-app-store/. [Online; accessed September-2019].

[182] Statista (2019c). Downloads from the play store. https://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/. [Online; accessed September-2019].

[183] Stephen, P. (2014). Avoiding singleton abuse. https://www.objc.io/issues/13-architecture/singletons/#avoiding-singletons. accessed: 2017-01-06.

[184] Strauss, A. and Corbin, J. (1994a). Grounded theory methodology. *Handbook of qualitative research*, 17:273–85.

[185] Strauss, A. and Corbin, J. (1994b). Grounded theory methodology: An overview. In Denzin, N. K. and Lincoln, Y. S., editors, *Handbook of Qualitative Research*, pages 273–285, Thousand Oaks, CA. Sage Publications.

[186] Syer, M. D., Nagappan, M., Adams, B., and Hassan, A. E. (2015). Replicating and re-evaluating the theory of relative defect-proneness. *IEEE Transactions on Software Engineering*, 41(2):176–197.

[187] Szydlowski, M., Egele, M., Kruegel, C., and Vigna, G. (2012). Challenges for dynamic analysis of iOS applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7039 LNCS:65–77.

[188] Tailor (2016). Tailor swift static analyzer. https://tailor.sh/. accessed: 2016-10-20.

[189] Terence, P. (2015). Objc.g4. https://github.com/antlr/grammars-v4/tree/master/objc. accessed:2017-01-06.

[190] Terence, P. (2016). Swift.g4. https://github.com/antlr/grammars-v4/blob/master/swift/Swift.g4. accessed:2017-01-06.

[191] Tom, E., Aurum, A., and Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516.

[192] Tómasdóttir, K. F., Aniche, M., and Deursen, A. v. (2017). Why and how javascript developers use linters. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 578–589. IEEE Press.

[193] Tsantalis, N. (2019). Jdeodorant. https://github.com/tsantalis/JDeodorant. [Online; accessed September-2019].

[194] Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008). Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE.

[195] Tsantalis, N., Mazinanian, D., and Rostami, S. (2017). Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*, pages 60–70. IEEE Press.

[196] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 403–414. IEEE Press.

[197] Tufano, M., Palomba, F., Oliveto, R., Penta, M. D., Lucia, A. D., and Poshyvanyk, D. (2017). When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, PP.

[198] Tukey, J. W. (1977). *Exploratory Data Analysis.* Addison-Wesley.

[199] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (1999). Soot-a java bytecode optimization framework. In *Proc. of the conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press.

[200] Verloop, D. (2013). *Code Smells in the Mobile Applications Domain.* PhD thesis, TU Delft, Delft University of Technology.

[201] Weisstein, E. W. (2004). Bonferroni correction.

[202] Wikipedia, C. (2019). Original iphone. https://en.wikipedia.org/wiki/IPhone_(1st_generation). [Online; accessed September-2019].

[203] Yamashita, A. and Moonen, L. (2013). Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251. IEEE.

[204] Yan, S., Xu, D., Zhang, B., Zhang, H.-J., Yang, Q., and Lin, S. (2006). Graph embedding and extensions: A general framework for dimensionality reduction. *IEEE transactions on pattern analysis and machine intelligence*, 29(1):40–51.

[205] Yanagiba (2015). Swift AST. https://github.com/yanagiba/swift-ast. [Online; accessed August-2019].

[206] Zadeh, L. A. (1974). Fuzzy logic and its application to approximate reasoning. In *IFIP Congress*, pages 591–594.

[207] Zaman, S., Adams, B., and Hassan, A. E. (2012). A qualitative study on performance bugs. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 199–208. IEEE.

[208] Zhang, L., Gordon, M. S., Dick, R. P., Mao, Z. M., Dinda, P., and Yang, L. (2012). Adel: An automatic detector of energy leaks for smartphone applications. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 363–372. ACM.

Appendix A

Sumac: A Code Smell Detector for iOS Apps

We present in this appendix the details of our tool SUMAC. Table A.1 describes the entities used to build the app model in *Sumac*.

Entity	***Description***
App	An application
Class	A class of the app
ExternalClass	A library class
Method	A method of a class
ExternalMethod	A method of library class
Argument	A method parameter
ExternalArgument	A parameter of an external method
Variable	An attribute of a class
Function	A function of an app
GlobalVariable	A global variable of an app

Table A.1 The entities used by SUMAC

Table A.2 summarizes the metrics used to describe Objective-C apps in *Sumac*.
Table A.3 summarizes the metrics used to describe Swift apps in *Sumac*.

Metric	*Concerned entities*
NumberOfClasses	App
NumberOfVariables	App, Class
NumberOfMethods	App, Class
NumberOfChildren	Class
ClassComplexity	Class
IsStatic	Method
NumberOfArguments	Method
NumberOfDirectCalls	Method
NumberOfCallers	Method
CyclomaticComplexity	Method
NumberOfLines	Class, Method
CAMC (Cohesion Among Methods in A Class)	Class
IsProtocol	Class
IsCategory	Class
NumberOfProtocols	App
NumberOfViewControllers	App
IsViewController	Class
IsInteractor	Class
IsRouter	Class
IsPresenter	Class
IsView	Class

Table A.2 The metrics used by SUMAC for analyzing Objective-C apps.

Metric	*Concerned entities*
NumberOfClasses	App
NumberOfVariables	App, Class
NumberOfMethods	App, Class
NumberOfChildren	Class
ClassComplexity	Class
IsStatic	Method
NumberOfArguments	Method
NumberOfDirectCalls	Method
NumberOfCallers	Method
CyclomaticComplexity	Method
NumberOfLines	Class, Method
IsProtocol	Class
IsCategory	Class
NumberOfProtocols	App
NumberOfViewControllers	App
IsViewController	Class
IsInteractor	Class
IsRouter	Class
IsPresenter	Class
IsView	Class
NumberOfExtensions	App
NumberOfStructs	App
NumberOfEnums	App
IsExtension	Class
IsStruct	Class
IsEnum	Class

Table A.3 The metrics used by SUMAC for analyzing Swift apps.

Appendix B

A Preliminary Survey for the Linter Study

In order to lay the foundation for our linter study, we conducted a short survey to ask developers about their linter usage for Android. Figures B.1 and B.2 present the questions of this survey. The questions of this survey go under three categories:

- The usage of Android linters: This represents the main questions of our survey. Their objective is to:
  - Check if the respondents use a linter specific to the Android platform or not;
  - Identify the main Android linters used by the respondents;
  - Identify the purposes/reasons for which the respondents used Android linters.
- The experience and expertise: The objective of these questions is to describe the background of the respondents. In particular, they identify:
  - The number of years of experience in Android development;
  - The level of expertise in Android development;
  - The context surrounding their development activity.
- Feedback: This represents a final question about potential comments regarding linters and the conducted survey.

We published this survey in a mobile development forum [168] to focus on developers specialized in mobile frameworks. We also selected this widely used forum to reach mobile developers with different backgrounds. The publication of the form allowed us to collect anonymous answers from 94 Android developers.

Figure B.3 shows developers' answers regarding the use of linters. The chart shows that 72% of the respondents used a linter specific to the Android platform.

Figure B.4 shows the developers' answers regarding the adopted linters. The chart shows that 97% of the respondents used Android Lint. Other linters like Infer and ktlint were also mentioned by developers, but only in low proportions.

Android linters

This survey is conducted by Sarra Habchi from Inria Lille.

A lint or linter is a program that analyzes source code to identify potential problems. Lints usually target a specific programming language or platform. This short survey focuses on Android-specific lints to identify the most-used ones and their applications.

For more information, feel free to contact us at sarra.habchi@inria.fr
Thanks in advance for participating!

* Required

1. **Do you use a lint in your Android developments?** *
   Mark only one oval.
   - ( ) Yes
   - ( ) No *Skip to question 4.*

Android Lints

2. **Which lints do you use for Android ?** *
   Check all that apply.
   - [ ] Android Studio Lint
   - [ ] Infer
   - [ ] Other: ____________

3. **For what reasons do you generally use it?** *
   Check all that apply.
   - [ ] Check correctness
   - [ ] Ensure coding conventions
   - [ ] Improve performance (energy, responsiveness, memory, network)
   - [ ] Improve security
   - [ ] Find potential bugs
   - [ ] Other: ____________

Experience and expertise

These will help us with classification. Completely anonymous.

Figure B.1 The form used to prepare the linter study (Part 1).

4. **For how many years have you been developing with Android?** *
Mark only one oval.

- <1
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

5. **How do you evaluate your expertise in Android?** *
Mark only one oval.

	1	2	3	4	5	
Beginner						Expert

6. **What is the context of your Android development activity?** *
Check all that apply.

- Open source projects
- Commercial projects
- Personal projects
- Other: ____________

7. **Any comments about Android lints, this survey or the research study?**

Figure B.2 The form used to prepare the linter study (Part 2).

Do you use a lint in your Android developments?

94 responses

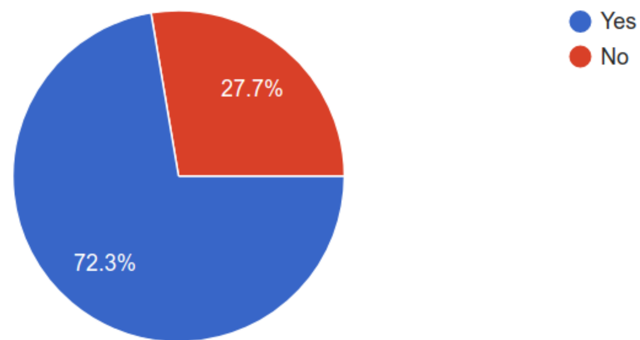

Figure B.3 Responses about the linter usage.

Which lints do you use for Android ?

68 responses

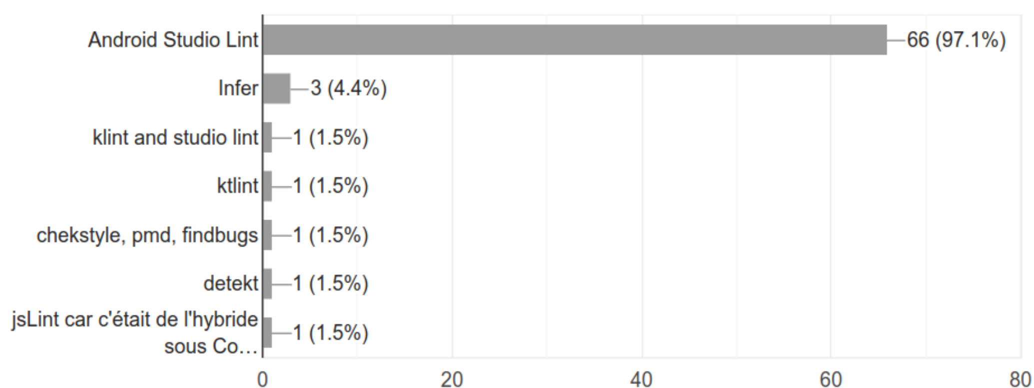

Figure B.4 Responses about the adopted used Android linters.

Figure B.5 shows the developers' answers regarding the reasons behind their linter usage. The chart shows that the respondents used linters mainly for checking the correctness of their code and finding bugs. In addition, the chart shows that only 51% of the respondents used linters for performance purposes.

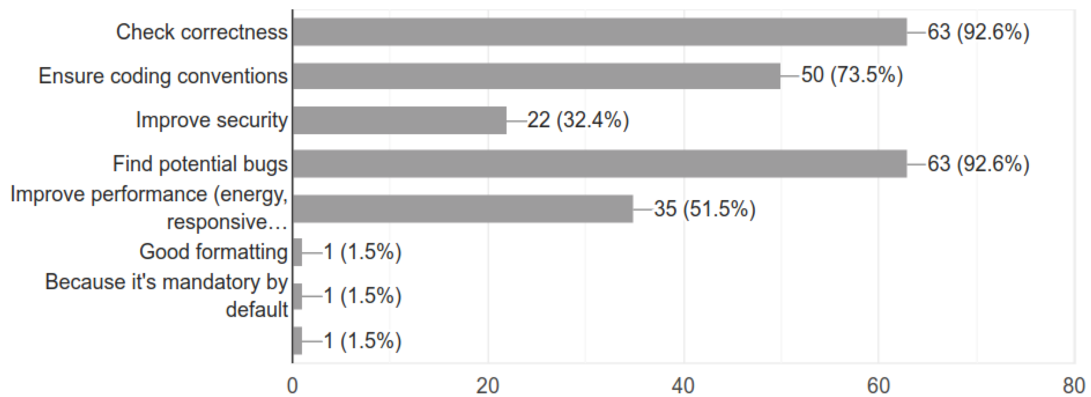

Figure B.5 Responses about the reasons for using an Android linter.

Figure B.6 describes the experience of the respondents in the field of Android development. We observe that most of the surveyed developers have more than one year experience in Android.

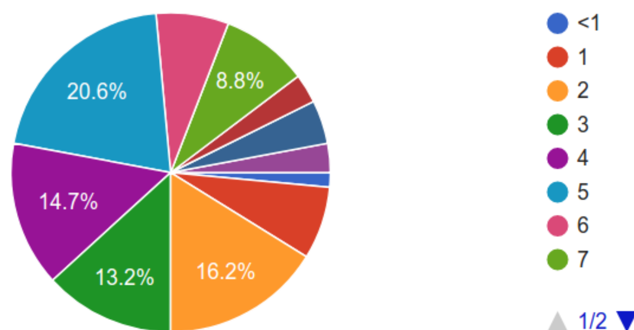

Figure B.6 The respondents' experience in Android.

Figure B.6 describes the expertise of the respondents in the field of Android development. The chart shows that 69% of the respondents consider that they have a higher than average expertise in Android.

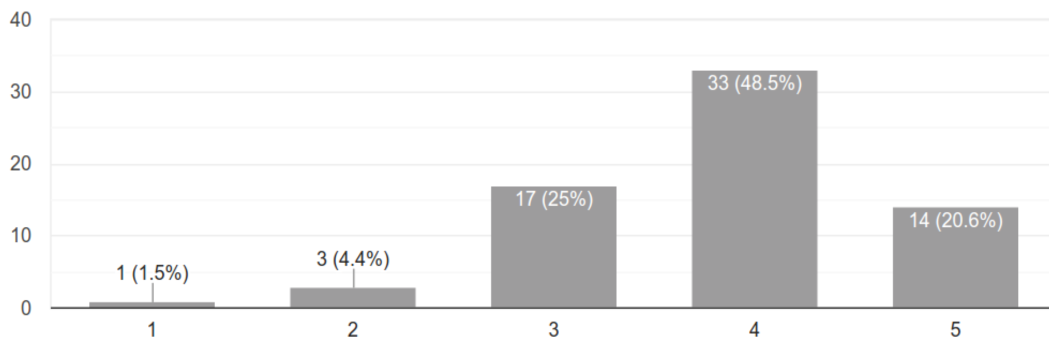

Figure B.7 The respondents' expertise in Android.

Figure B.6 describes the context surrounding the app development among respondents. The chart shows that 88% of the respondents developed apps for commercial projects.

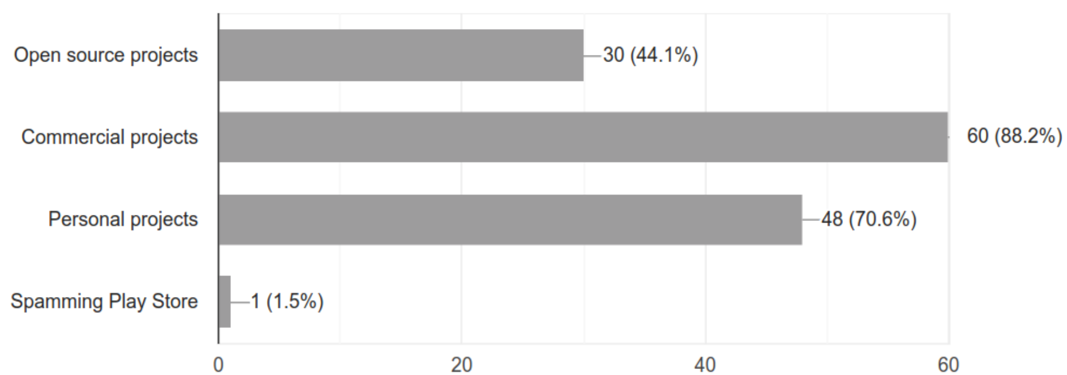

Figure B.8 The context of app development.

Figure B.6 shows the respondents' comments about Android linters.

Any comments about Android lints, this survey or the research study?

4 responses

lint with auto correct suggetions should be more.it helps very much.
Android lint has lots of advantages but it is not enough visual in report. It could be more like Sonar giving a level of Quality not just listing erros,bugs...
pratique
The things that i would want in Android Lint would be a global rating of some metrics (quality, dulications... like Sonar) and more lint checks

Figure B.9 Comments about the linter.