An Empirical Study on the Impact of Refactoring in Android Applications

by

Oumayma Hamdi

MANUSCRIPT-BASED THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS IN SOFTWARE ENGINEERING
M.A.Sc.

MONTREAL, DECEMBER 17, 2021

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

ACKNOWLEDGEMENTS

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

First and foremost, I would like to thank God, who has given me strength and encouragement throughout all the challenging moments of completing this work.

I would like to express the deepest appreciation for my advisor, Dr. Ali Ouni, for offering me the opportunity to carry out this work. Thank you for the many advices and discussions that helped me achieve this work. Further, I extend my gratitude to Dr. Mohamed Wiem Mkaouer, Dr. Eman Abdullah AlOmar and Dr. Mel Ó Cinnéide for the collaboration and for their insightful comments, suggestions and writing improvements of the published papers.

I would like to thank Professor Naouel Moha for agreeing to be my examiner and for making an effort to revise my dissertation. My sincere thanks go also to the president of the jury, Chamseddine Talhi, for the honor he did me by agreeing to judge my work.

I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Mitacs and École de Tchnologie Supérieure for the trust they have placed in me by funding my research.

Finally, I would like to thank my family, especially my parents and my two brothers who have always been there for me and who had to endure the distance. Also, thanks to my friends who have provided me through moral and emotional support in my life, especially Younes who is without doubt the most striking meeting I had during this journey.

Une étude empirique sur l'impact du refactoring dans les applications Android

Oumayma Hamdi

RÉSUMÉ

Les applications mobiles (apps) deviennent des logiciels complexes qui doivent être développés rapidement tout en évoluant de manière continue afin de répondre aux nouveaux besoins des utilisateurs ainsi qu'aux mises à jour régulières. Cette évolution rapide peut provoquer la présence de mauvais choix d'implémentions ou de conception qui se manifestent par ce qu'on appelle défauts de code ou code smells. La présence de ces défauts au sein d'une application peut dégrader la qualité et les performances en compliquant les tâches de maintenance et d'évolution. Il est alors important de connaître ces défauts mais aussi de pouvoir les détecter et les corriger, afin de permettre aux développeurs d'améliorer la qualité et les performances de leur application. Pour corriger les défauts de code et améliorer la qualité des apps, refactoring, est une technique clé largement acceptée qui consiste à modifier la structure interne du code source tout en préservant son comportement externe.

Bien que la plupart des études existantes ont étudié l'impact des activités de refactoring dans les applications orientées objet, peu d'attention a été accordée aux applications mobiles. En particulier, il y a un manque de connaissances sur l'impact de refactoring sur différents aspects de la qualité du logiciel, y compris les défauts de code et les métriques de qualité. De plus, la plupart des recherches précédentes se sont concentrées sur l'étude des caractéristiques des défauts de code orientées objet (OO) traditionnelles affectant les fichiers de code source dans les systèmes logiciels traditionnels, et ont préconisé que l'interaction et la co-présence des défauts de code réduisent la capacité des développeurs à comprendre et maintenir le code source. Cependant, peu de connaissances sont disponibles sur les catégories émergentes de défauts de code spécifiques à Android et leurs interactions, c'est-à-dire les cooccurrences, avec les défauts OO traditionnelles, dans le contexte des applications Android.

Par conséquent, dans cette mémoire, nous menons une série d'études empiriques pour mieux comprendre le concept de refactoring et de défauts de code dans les applications mobiles. Premièrement, nous menons une étude longitudinale sur l'historique d'évolution de cinq applications Android pour étudier l'impact de refcatoring sur les défauts de code orientés objets et spécifique à Android. Ensuite, nous performons une étude empirique sur un vaste ensemble de données composé de 1,923 applications Android pour étudier la co-occurrence des défauts de code. Finalement, nous réalisons une étude sur un ensemble de données composé de 300 applications Android pour analyser l'impact du refactoring sur les métriques de qualité.

Nos résultats montrent que (1) les défauts de code sont répandues dans les applications Android, mais les classes contenant des défauts de code ne sont pas particulièrement ciblées par les activités de refactoring et, lorsqu'elles le sont, il est rare que le refactoring élimine réellement lé défaut, (2) le phénomène de co-occurrence est en effet répandu dans les applications Android et plusieurs types de défauts de code ont de fortes associations, et (3) lorsque le refactoring affecte les métriques, elle les améliore généralement. Dans de nombreux cas, le refactoring n'a pas d'impact significatif sur les métriques, alors qu'une métrique (LCOM) se détériore globalement à la suite du refactoring.

Les résultats de ces études empiriques ont révélé des résultats de recherche importants. Notamment, sur la pratique actuelle du refactoring dans le contexte de développement d'applications mobiles ainsi que sur les défauts de code et leur interaction. Ces résultats nous ont permis d'élaborer quelques recommandations pour les chercheurs et les développeurs d'outils pour concevoir des outils de détection et de refactoring des défauts de code mobile. Nos résultats ouvert des perspectives pour des travaux de recherche sur l'identification des défauts de code mobiles et sur les pratiques de développement en général.

Mots-clés: applications mobiles, refactoring, défauts de code, métriques de qualité, étude empirique

An Empirical Study on the Impact of Refactoring in Android Applications

Oumayma Hamdi

ABSTRACT

Android applications (apps) are becoming complex software that must be developed quickly while continuously evolving to meet new user needs as well as regular updates. This rapid evolution may lead to poor implementation and design choices, known code smells. The presence of these smells within an application may degrade the quality and performance by complicating maintenance and evolution tasks. Thus, it is important to know these smells but also to detect and correct them, in order to allow developers to improve the quality and performance of their applications. Code refactoring is a key practice that is employed to ensure that the intent of a code change is properly achieved without compromising internal software quality.

While most of existing studies have investigated the impact of refactoring activities in desktop applications, little attention has been paid to mobile applications. In particular, there is a lack of knowledge about the impact of refactoring on different software quality aspects, including code smells and quality metrics. Moreover, most of previous research focused on studying the characteristics of traditional object-oriented (OO) code smells affecting source code files in desktop software systems, and advocated that the interaction and co-presence of code smells reduce the ability of developers to understand and maintain source code. However, little knowledge is available on emerging categories of Android-specific code smells and their interactions, i.e., co-occurrences, with traditional OO smells, in the context of Android apps.

Therefore, in this work we conduct a series of empirical studies to better understand the concept of refactoring and code smells in mobile apps. First, we conduct a longitudinal empirical study that analyses the evolution history of five open-source mobile apps to investigate the impact of refactoring on code smells. Then, we examined the prevalence of code smell co-occurrences and determine which code smell types tend to co-exist more frequently on a large dataset composed of 1,923 open-source mobile apps. Finally, we perform a study on a dataset composed of 300 open-source mobile apps to analyze the impact of refactoring on quality metrics.

Our results show that (1) code smells are widespread across Android applications, but smelly classes are not particularly targeted by refactoring activities and, when they are, it is rare for refactoring to actually remove a smell, (2) the co-occurrence phenomenon is indeed prevalent in Android apps and several smell types have strong associations, and (3) when refactoring affects the metrics it generally improves them. In many cases refactoring has no significant impact on the metrics, whereas one metric (LCOM) deteriorates overall as a result of refactoring.

The results of these studies revealed important research findings. In particular, on the current practice of refactoring and code smells in the context of mobile applications. These results allowed us to develop some recommendations for researchers and tool creators to design tools for detecting and refactoring mobile code smells. Our results opened perspectives for research works about the identification of code smells in mobile codes and development practices in general.

Keywords: mobile apps, refactoring, code smell, quality metrics, empirical study

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Apps	Applications
BC	Blob Class
CC	Complex Class
CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
DTWC	Data Transmission Without Compression
DR	Debuggable Release
DWL	Durable Wakelock
ETS	École de Technologie Supérieure
FE	Feature Envy
IDFP	Inefficient Data Format and Parser
IDS	Inefficient Data Structure
ISQLQ	Inefficient SQL Query
IGS	Internal Getter and Setter
LC	Lazy Class
LM	Long Method
LOC	Lines of code
LCOM	Lack Of Cohesion of Methods
LIC	Leaking Inner Class

LPL	Long Parameter List
LT	Leaking Thread
MC	Message Chain
MIM	Member Ignoring Method
NOSI	Number Of Static Invocations
NLMR	No Low Memory Resolver
OO	Object-Oriented
PD	Public Data
RAM	Rigid Alarm Manager
RFC	Response For a Class
RR	Refused Bequest
SC	Spaghetti Code
SG	Speculative Generality
SL	Slow Loop
UC	Unclosed Closable
VQTY	Variable Quantity
WMC	Weight Method Class

INTRODUCTION

Context

Android is the dominant operating system for mobile devices Research (2020) and has experienced a tremendous expansion of its user base over the past number of years. It currently runs on approximately 86% of smart phones worldwide, with over 2.5 billion monthly active users [1]. The vibrancy of the Android operating system can be partially attributed to its openness to vendor customizations, along with the richness of the functionalities it provides in its platform that eases the development and evolution of apps that cover ever-broadening domains of applications Zhang & Cai (2019). As a consequence of this popularity, the Android app marketplace is extremely congested and strong competition makes it necessary to build mobile apps rapidly and evolve them continuously to meet the needs of users. Such activities may lead to poor implementation and design choices, known *code smells* that decrease the quality of code. To face this challenge, developers have to pay careful attention to the quality of their mobile apps by applying *refactoring* which is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design. Indeed, 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 traditional software system.
- Another particularity is that mobile apps handle sensors that respond to device movement, numerous gestures, global positioning system, cameras and multiple networking protocols. 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 quality. Hence, quality can be considered as a key challenge of mobile development.

Problem statement

[1] https://venturebeat.com/2019/05/07/android-passes-2-5-billion-monthly-active-devices

1. **Problem 1: Lack of knowledge about the impact of refactoring on code smells in mobile applications**

   Mobile apps differ significantly from traditional software systems Minelli & Lanza (2013); Mannan, Ahmed, Almurshed, Dig & Jensen (2016); Kessentini & Ouni (2017) in having to deal with limitations on specific hardware resources like memory, CPU, display size, etc., as well as the highly dynamic nature of the mobile app market and the ever-increasing user requirements. These differences can play an important role in mobile app development and evolution. Indeed, unlike object-oriented (OO) software systems AlOmar, Mkaouer, Ouni & Kessentini (2019); Ó Cinnéide, Tratt, Harman, Counsell & Hemati Moghadam (2012); Alshayeb (2009); Bavota, De Lucia, Di Penta, Oliveto & Palomba (2015a); Cedrim, Sousa, Garcia & Gheyi (2016); Stroggylos & Spinellis (2007); Tahir, Dietrich, Counsell, Licorish & Yamashita (2020), the impact of refactoring on code smells in mobile apps has received little attention. Hence, much uncertainly exists about the relationship between refactoring and code smells in mobile apps.

   Proving the existence of an impact will help tool creators, practitioners and researcher. For tool creators, it help them to develop practical and reliable refactoring tools for mobile apps. As for practitioners and researchers discovering the relationship can help them to understand current refactoring practices and their impact on code smells.

2. **Problem 2: The presence and impact of the code smells co-occurrence phenomenon is unknown in mobile applications**

   Most of previous research focused on studying the characteristics of traditional object-oriented (OO) code smells affecting source code files in desktop software systems, and advocated that the interaction and co-presence of code smells reduce the ability of developers to understand and maintain source code. However, little knowledge is available on emerging categories of Android-specific code smells and their interactions, i.e., co-occurrences, with traditional OO smells, in the context of Android apps.

This knowledge is particularly important for developers researchers and tool creators. For Android developers, discovering such relationships will help them focusing their attention by getting a high priority in refactoring the smells that frequently co-occur together which may lead to better monitoring the quality of their apps. Moreover, it can help them to save time and effort when refactoring their code and increase their awareness and understanding of their apps. As for researchers, it can be a starting point for a deep investigation of the relation between Android smells and traditional code smells. Also, such knowledge can help researchers designing Android-specific refactoring techniques and prototypes that take into consideration the hidden dependencies between such smells.

3. **Problem 3: Lack of knowledge about the impact of refactoring on quality metrics in mobile applications**

   This problem is similar to the first one. Indeed, the impact of refactoring on quality metrics in mobile apps has received little attention counter to object-oriented (OO) software systems AlOmar *et al.* (2019); Ó Cinnéide *et al.* (2012); Alshayeb (2009); Bavota *et al.* (2015a); Cedrim *et al.* (2016); Stroggylos & Spinellis (2007).

   To develop efficient and reliable refactoring support tools for mobile apps, there is a need to better understand the current refactoring practice and its impact on structural quality.

Contributions

1. **Contribution 1: Impact of refactoring on code code smells in Android apps**

   The first contribution of this work is an empirical study that aims at understanding the impact of refactoring on code smells by analyzing the evolution history of five open-source Android apps exhibiting a total of 9,600 refactoring operations. We consider 15 common Android smell types and 10 common traditional Object-Oriented (OO) code smell types. We started with a preliminary study of the prevalence and co-occurrences of a number of Android-specific and traditional code smells, and then investigated the impact of different refactoring operations on these code smells.

The study delivers several important findings. Firstly, results indicate that code smells are very prevalent in Android apps, with 68% of classes being affected by Android specific smells and 63% of classes affected by traditional OO smells. Secondly, developers are more likely to apply refactoring operations to *non-smelly* code elements. A total of 25% of refactorings were applied to traditional OO smells, while 23% of refactorings were applied to Android smells. Thirdly, these refactoring activities removed only 5% of traditional OO smells and 1.5% of Android smells.

Moreover, it is worth noting that we observed from the analyzed smell instances that some classes are often affected by multiple types of code smells. To have detailed analysis of smells prevalence, we thus assess the phenomenon of smells co-occurrence in the second contribution.

2. **Contribution 2: Co-occurrences of code smells in Android apps**

   The second contribution of this work is an empirical study where we investigated the co-occurrence of code smells in Android apps on a large dataset composed of 1,923 open-source apps, 15 types of Android smells and 10 types of OO code smells. We jointly analyzed (1) the prevalence of the co-occurrence phenomenon, and (2) code smell pairs that most tend to co-occur.

   The key findings of our study indicate that *(1)* the co-occurrence phenomenon is quite prevalent in Android apps with 34%, 26% and 51% of classes are affected respectively by more than one Android, OO and both (i.e., OO and Android) smell types, and *(2)* there exist 14 smell pairs frequently co-occur together.

   Indeed, as the refactoring can be applied to delete code smells, it can also be applied to improve the software quality, in particular software quality metrics. Thus, in the last contribution we used a non-smell model which is the software quality metrics, to determine if they can better explain the refactoring activities that take place during Android app development.

3. **Contribution 3: Impact of refactoring on quality metrics in Android apps**

   The last contribution is a study that aims at investigating the impact of refactoring on quality metrics in Android apps. We mined 300 open-source apps containing 42,181 refactoring operations in total. We determined the effect each refactoring had upon the 10 chosen software quality metrics, and employed the difference-in-differences (DiD) model to determine the extent to which the metric changes brought about by refactoring differ from the metric changes in non-refactoring commits.

   The results indicate that for most refactoring type and metric combinations, the refactoring produced no significant change in the metric. On the other hand, some refactoring types yielded a broad improvement in several metric values. LCOM stood out as the least consistent metric, improving for some refactoring types and disimproving for others. For the non-refactoring commits, the metrics exhibit no significant change, other than (unsurprisingly) the design size metrics.

Outline

Knowing that this document is structured by articles, we present a general review of the literature followed by three chapters dedicated to each of the contribution. In each chapter, you will find a discussion of the results and general conclusion. The conclusion and the future work is presented at the end of this thesis.

Publications

- Oumayma Hamdi, Ali Ouni, Mel Ó Cinnéide and Mohamed Wiem Mkaouer. **A Longitudinal Study of the Impact of Refactoring in Android Applications**. In Information and Software Technology (IST) journal, 2021.
- Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel Ó Cinnéide and Mohamed Wiem Mkaouer. **A large empirical study on the impact of refactoring on quality attribute in Android applications**. In the 8th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2021.

- Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar and Mohamed Wiem Mkaouer. **An Empirical Study on Code Smells Co-occurrences in Android Applications**. In the 5th International Workshop on Refactoring (IWoR), 2021.

CHAPTER 1

STATE OF THE ART

In this chapter, we introduce the necessary background and concepts for understanding the subject and as well as the most relevant works for our topic. This includes *(i)* the mobile system on which we have worked throughout this thesis which is the Android system, (*ii*) code smells and their prevalence, evolution, and co-occurrences and (*iii*) the impact of refactoring on different software quality aspects, including code smells and quality metrics.

1.1 Basic concepts

1.1.1 Android System

Android is an operating system developed by Google for mobile devices. It is based on a Linux kernel and Android applications are usually developed in Java. We chose to work on Android because it is a free operating system. It is also the most popular mobile system. Google says that in 2020 there are more than 2.5 billion Android devices worldwide[2]. Moreover, this system offered us the opportunity to access a large number of applications more simply than its competitors.

The Android architecture is summarized in Figure 1.1. In our work, we are interested in the JAVA API framework in green on which all the applications are based. Thus, most of the code smells that we address in this thesis are linked to this application framework, in particular because they appear in classes which inherit from this framework.

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

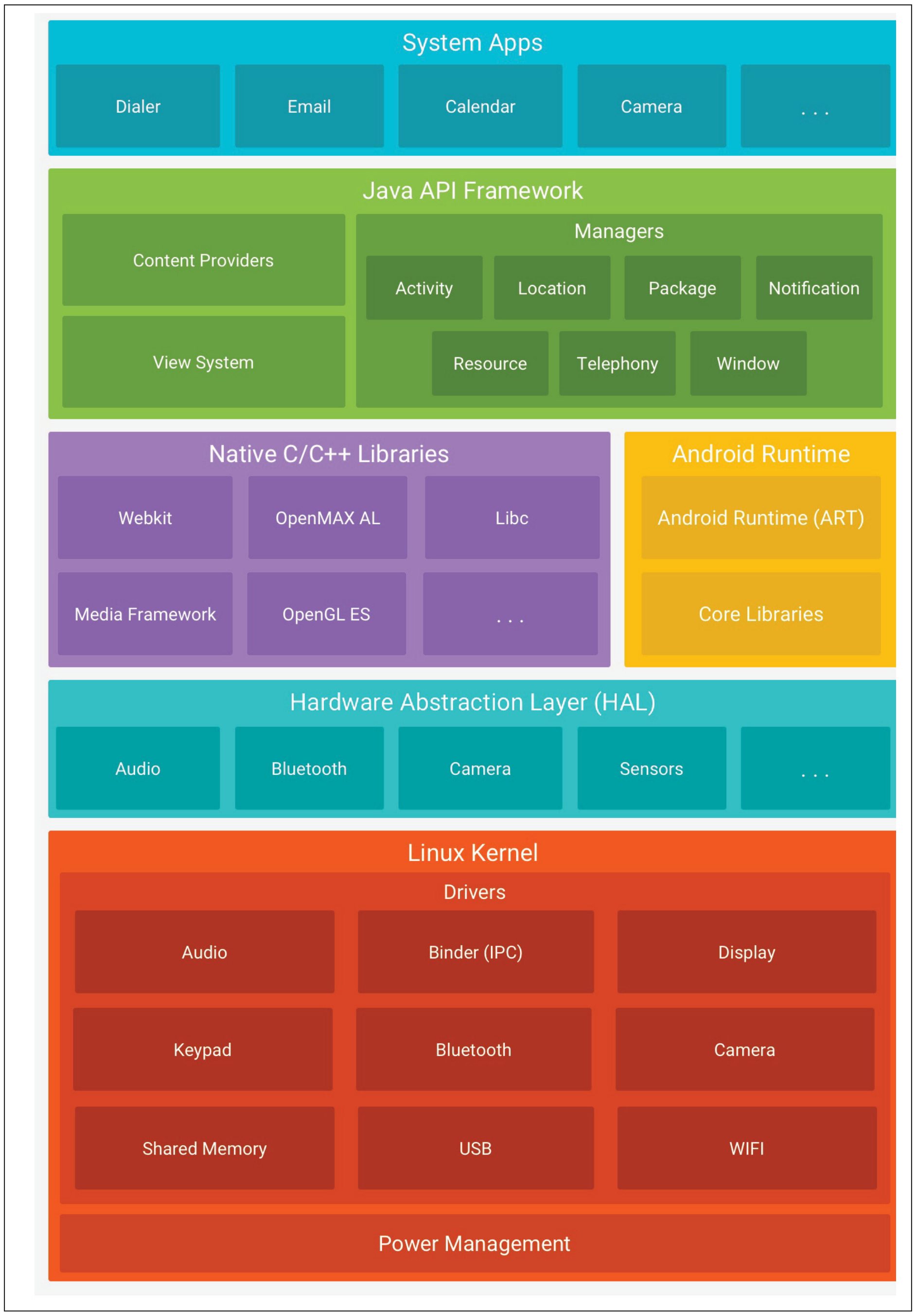

Figure 1.1 Android Architecture
Taken from Android Developers (2021)

1.1.2 Refactoring

Refactoring is the process of reworking the program's source code without changing its functionality (e.g., preserving its external behavior) to improve it structure in terms of readability, complexity, maintainability, extensibility, reusability, etc. The concept of refactoring was introduced by Opdyke (1992) and popularized later in Fowler's well-known book Fowler, Beck, Brant, Opdyke & Roberts (1999). Fowler first provided a comprehensive list of code smells along with a set of possible refactoring operations to fix each smell type. In this work, we focus on 12 common refactoring operations, as indicated in Table 1.1.

Table 1.1 The list of studied refactoring operations

Ref	**Refactoring**	**Description**	**Level**
MM	Move Method	Moves a method from a class to another class.	Method
EM	Extract Method	Creates a new method from an existing fragment of code.	Method
IM	Inline Method	Replaces calls to the method with the method's content and delete the method itself.	Method
RM	Rename Method	Renames Method.	Method
EMM	Extract and Move Method	Extracts and moves method.	Method
PDM	Push Down Method	Moves a method from a class to those subclasses that require it.	Method
PUM	Pull Up Method	Moves a method from a class(es) to its immediate superclass.	Method
MA	Move Attribute	Moves Attribute from a class to another class.	Attribute
PDA	Push Down Attribute	Moves an attribute from a class to those subclasses that require it.	Attribute
PUA	Pull Up Attribute	Moves an attribute from a class(es) to their immediate superclass.	Attribute
ESC	Extract Super Class	Creates a superclass from two classes with common attributes and methods.	Class
RC	Rename Class	Renames a class.	Class

1.1.3 Code smells

Code smells are implementation problems that come from poor design and/or coding choices. They are viewed as problem symptoms that can make the maintenance and evolution of software more challenging Fowler *et al.* (1999); Brown, Malveau, McCormick & Mowbray (1998a). Software developers may introduce code smells during the initial development or during the evolution of the system. Such smells may increase the risk of bugs and failures Palomba *et al.* (2018a); Khomh, Di Penta, Guéhéneuc & Antoniol (2012).

In the context of Android applications, there are two types of code smells that are of concern in this study:

- *Object-Oriented smells*: This include regular OO smells such *Long Method*, *Feature Envy*, *God Class*, as defined by Fowler *et al.* (1999) can exist in Android apps. A non-exhaustive list that contains 10 of the most common OO code smells is reported in Table 1.2.
- *Android-specific smells*: Certain specific smells are prevalent only in Android apps, e.g., *Slow Loop*, *Leaking Thread*, and *Durable Wakelock*. A list of 15 common Android-specific code smells proposed by Reimann, Brylski & Aßmann (2014) and Palomba, Di Nucci, Panichella, Zaidman & De Lucia (2017a) is described in Table 1.3.

Table 1.2 List of traditional Oriented-Object code smells
Taken from Bavota et al. (2015, p.3) and Palomba et al. (2018, p.3)

Abbreviation	**Code smell**	**Description**
BC	Blob Class	A large class implementing different responsibilities and centralizing most of the system processing.
CC	Complex Class	A class having at least one method having a high cyclomatic complexity.
FE	Feature Envy	A method is more interested in a class other than the one it actually is in.
LC	Lazy Class	A class having very small dimension, few methods and low complexity. It does not do enough to justify its existence.
LM	Long Method	A method that is unduly long in terms of lines of code.
LPL	Long Parameter List	A method having a long list of parameters some of which are avoidable
MC	Message Chain	A long chain of method invocations is performed to implement a class functionality.
RR	Refused Bequest	A class that uses only some of its inherited properties while redefining most of the inherited methods, thus signaling a poorly-designed hierarchy.
SC	Spaghetti Code	A class implementing complex methods interacting between them, with no parameters, using global variables.
SG	Speculative Generality	A class declared as abstract having very few children classes using its methods.

Table 1.3 List of Android-specific code smells
Taken from Reimann et al. (2014, p.2) al. and Palomba et al. (2017, p.2)

Abbreviation	**Android smell**	**Description**
DTWC	Data Transmission Without Compression	A method that transmits a file over a network infrastructure without compressing it.
DR	Debuggable Release	Leaving the attribute *android:debuggable* true when the app is released.
DWL	Durable Wakelock	A method using an instance of the class *WakeLock* acquires the lock without calling the release.
IDFP	Inefficient Data Format and Parser	A method using `treeParser`, slows down the app, and should be avoided and replaced with other more efficient parsers (e.g., `StreamParser`) Reimann *et al.* (2014).
IDS	Inefficient Data Structure	A method using *HashMap <Integer,Object>*.
ISQLQ	Inefficient SQL Query	A method defining a JDBC connection and sending an SQL query to a remote server.
IGS	Internal Getter and Setter	Accessing internal fields via getters and setters is expensive in Android development and, thus, internal fields should be accessed directly.
LIC	Leaking Inner Class	A non-static nested class holding a reference to the outer class.
LT	Leaking Thread	An Activity starts a thread and does not stop it
MIM	Member Ignoring Method	Non-static methods that do not access any internal properties.
NLMR	No Low Memory Resolver	A mobile app that does not contain the method *onLowMemory*.
PD	Public Data	A class that does not define the context or define the context as non-private.
RAM	Rigid Alarm Manager	A class using an instance of *AlarmManager* does not define the method *setInexact−Repeating*.
SL	Slow Loop	Using the for-loop version.
UC	Unclosed Closable	A class that does not call such the close method to release resources that an object is holding.

1.1.4 Quality metrics

Various quality metrics are used to measure the quality of a software system. One of the most widely used metric suites is that defined by Chidamber and Kemerer. We selected and used in our study a non-exhaustive list of eight common quality metrics as described in Table 1.4.

Table 1.4 The list of quality metrics employed in this study

Abbreviation	**Metrics**	**Description**
CBO	Coupling Between Objects	Number of classes that are coupled to a particular class Chidamber & Kemerer (1994).
DIT	Depth of Inheritance Tree	Number of classes that a particular class inherits from Chidamber & Kemerer (1994).
LOC	Lines of code	Number of lines of code ignoring spaces and comments Aniche (2016).
LCOM	Lack Of Cohesion of Methods	Numbers of pairs of methods that shared references to instance variables Chidamber & Kemerer (1994).
NOSI	Number Of Static Invocations	Number of invocations of static methods Aniche (2016)
RFC	Response For a Class	Number of method invocations in a class Chidamber & Kemerer (1994).
VQTY	Variable Quantity	Number of declared variables Aniche (2016).
WMC	Weight Method Class	The sum of all the complexities of the methods (McCabe's cyclomatic complexity) in c the class Chidamber & Kemerer (1994).

1.2 Related Work

1.2.1 Studies about the evolution and diffuseness of code smells

A first study on code smells during the evolution of software systems has been conducted by Chatzigeorgiou and Manakos Chatzigeorgiou & Manakos (2010). The study shows that the number of instances of code smells increases as system evolves and persists up to the latest examined version, and developers do not perform refactorings in order to remove them. Similarly, a study by Peters and Zaidman Peters & Zaidman (2012) shows that even when developers are aware of the presence of code smells in the source code, they are unlikely to invest time on refactoring to remove them. The reasons of such practice are explored by Arcoverde et al. Arcoverde, Garcia & Figueiredo (2011), who reported a survey in order to understand the longevity of code smells and showed that code smells often remain in source code for a long time and the main reason to postpone their removal through refactoring activities is to avoid API modifications Arcoverde *et al.* (2011).

The evolution of code smells has also been studied by Olbrich et al. Olbrich, Cruzes, Basili & Zazworka (2009) who analyzed the historical data of two projects, namely Lucene and Xerces, over several years and concluded that *God Classes* and *Shotgun Surgery* have a higher change frequency than other classes noting that they neither performed an analysis to control the effect of the size on their results nor studied the kinds of changes affecting these OO code smells. Moreover, Vaucher et al.Vaucher, Khomh, Moha & Guéhéneuc (2009) considered in their study on *God Class* code smells to investigate whether they affect software systems for long periods of time and making a comparison with whether the code smell is refactored.

Habch et al. Habchi, Rouvoy & Moha (2019b) investigated the survival of eight types of Android code smells. The study reports that while in terms of time Android code smells can remain in the codebase for years before being removed, it takes an average of 34 effective commits to be removed. However, there is no investigation on what types of refactoring changes are performed to remove Android smells. Recently, Bessghaier et al. Bessghaier, Ouni & Mkaouer (2020) studied a new emerging category of code smells in modern Web applications and analyzed their prevalence and found that such smells do increase code change proneness. Delchev and Harun Delchev & Harun (2015) were interested in the frequency of code smells and the severity of their effects. They conducted a survey with 73 developers, on 10 traditional code smells, about how frequently developers encountered a smell and how likely they were to refactor such smells. With regards to Android projects, the survey found that developers faced *Long Method* smells more often than other smells, but Shotgun Surgery was more likely to be refactored. Furthermore, frequency and severity vary relatively to the programming language. As for developer experience, the study indicates that the more experienced the developers, the less likely they face smells. However, when these more experienced developers do face a smell instance, they had a higher tendency to refactor that smell.

Vidal et al. Vidal, Marcos & Díaz-Pace (2016) observed that the number of code smells suggested by existing metric-based tools usually exceed the number of design problems that developers can deal with. For this reason, they proposed a prioritization approach based on previous modifications of a class, important modifiability scenarios for the considered system, and the relevance of the code smell type. Sharma et al. Sharma, Singh & Spinellis (2020) investigated the relationship between design and architecture smells in OO software systems and found that the density of code smells does not depend on the project size and that architecture smells and design smells are highly correlated. The study results suggest that design and architecture smell pairs do not generally co-occur.

Recently, Peruma et al. Peruma *et al.* (2020, 2019) conducted an empirical study on the occurrences and distribution of test smells in Android apps. Their findings show a widespread occurrence of test smells in apps. The study shows that Android apps tend to exhibit test smells early in their lifetime with different degrees of co-occurrences on different smell types. The study also demonstrates that test smells can be used as an indicator of necessary preventive software maintenance for test suites. Bavota et al. Bavota, Qusef, Oliveto, De Lucia & Binkley (2012) analyzed the distribution of unit test smells in 18 software systems providing evidence that they are widely spread, but also that most of them have a strong negative impact on code comprehensibility. The diffusion of Android smells has been recently studied by Palomba et al. Palomba, Di Nucci, Panichella, Zaidman & De Lucia (2019) who conducted a large empirical study on Android apps. Their findings show that *Leaking Thread*, *Member Ignoring Method*, *Slow Loop* and, *Data Transmission Without Compression* Android smells occur more frequently than others. Tufano et al. Tufano *et al.* (2016) obtained similar results by reporting a large-scale empirical study, which showed that test smells are usually introduced by developers when the corresponding test code is committed in the repository for the first time and they tend to remain in a system for a long time. The study is complementary to the one by Tufano et al., since it is focused on the analysis of the design flaws arising in the production code. Mannan et al. Mannan *et al.* (2016) compared the distribution of OO code smells in Android applications to the ones in desktop apps. Despite the different nature of mobile applications, they did not observe any major differences in terms of density.

Synthesis: Our study on the prevalence of Android and traditional OO code smells is complementary to these studies discussed above. However, our goal is to go in deeper and analyze the extent to which developers apply refactoring to fix Android and OO smells as well as the effectiveness of refactoring at removing such smells.

1.2.2 Studies about the co-occurences of code smells

Palomba et al. Palomba *et al.* (2018b) conducted a large-scale empirical study aimed at quantifying the diffuseness of the co-occurrence phenomenon in terms of how frequently code smells occur together. The results of this study indicate 59% of smelly-classes are affected by more than one smell. In particular, six smell types frequently co-occur together (e.g., Complex Class and Message Chains). In the same context, Garg et al. Garg, Gupta, Bansal, Mishra & Bajpai (2016) investigate the co-occurrence of code smells in two open-source software, Mozilla and Chromium. They observed that co-occurrence patterns are presented in both the software with a small variation in their co-occurrence percentage. Some code smells are more common such as *Data Clumps*, *Internal Duplication*, and *External Duplication*. Similarly, a study by Fontana et al. Fontana, Ferme & Zanoni (2015) examined code smells co-occurrence in a set of 111 open source systems. They observe that *Brain Method* has the largest share of co-occurrences. However, they found no co-occurrence between *God Class* and *Data Class*. Recently, Muse et al. Muse *et al.* (2020) studied a new category of SQL code smells data-intensive systems finding that some traditional code smells have a higher association with bugs compared to SQL code smells.

As for studies investigating the effects between code smell co-occurrences and code maintainability. Abbes et al. Abbes, Khomh, Gueheneuc & Antoniol (2011) examined the interactions between code smells and their effects. The authors concluded that when code smells appeared isolated, they had no impact on maintainability, but when they appeared interconnected, they brought a major maintenance effort. Yamashita et al. Yamashita, Zanoni, Fontana & Walter (2015) presented an extension study on inter-smell relations in both open and industrial systems, finding that the relation between smells vary depending on the type of system taken into account. Yamashita and Moonen Yamashita & Moonen (2013b) analyzes the impact of the interSmell relations in the maintainability of four medium-sized industrial systems written in Java. The authors detect significant relationships between *Feature Envy*, *God Class* and *Long Method* and conclude that Inter-Smell relationships are associated with problems during maintenance activities .

Synthesis: We observe from the existing literature that most studies focus basically on desktop applications while little knowledge is available for mobile apps. Existing studies are merely limited to some particular code smell types. In our study, we focus basically on Android apps while considering the analysis of both Android-specific and traditional OO code smells.

1.2.3 Studies about the impact of refactoring on code smells

Bavota et al. Bavota *et al.* (2015a) analyzed various versions of three Java open source projects to investigate whether refactorings applied to classes having a low level of maintainability and examined their ability to remove traditional OO code smells. The study shows that 42% of refactoring operations are performed on smelly-classes, only 7% of them removed code smells from the affected classes. Refactorings are detected at the release level using Ref-Finder tool Prete, Rachatasumrit, Sudan & Kim (2010). Besides the low accuracy of Ref-Finder achieving an overall precision of 35% and an overall recall of 24% as shown in prior studies Hegedűs, Kádár, Ferenc & Gyimóthy (2018); Kádár, Hegedűs, Ferenc & Gyimóthy (2016); Soares, Gheyi, Murphy-Hill & Johnson (2013), our experimental setup is quite different. We study Android apps while Bavota et al. Bavota *et al.* (2015a) focus on desktop apps as well as distinguishing between both Android specific and traditional OO smells. We also follow a fine grained analysis of refactoring changes at the commit level using RefactoringMiner Tsantalis, Mansouri, Eshkevari, Mazinanian & Dig (2018b) instead of the release level to achieve fine grained analysis and higher level of precision and recall in the detected refactorings. To follow up with Bavota et al. Bavota *et al.* (2015a), Yoshida et al. Yoshida, Saika, Choi, Ouni & Inoue (2016) revisited the relationship between code smells and refactoring in desktop applications. The study identified different smell-refactoring patterns in practice and found some patterns that match with Fowler's catalog Fowler *et al.* (1999), while highlighting other new patterns refactorings instances on their studied system.

Recently, Muse et al. Muse *et al.* (2020) studied a new category of SQL code smells in data-intensive systems finding that SQL code smells have a weak co-occurrence with traditional OO code smells a weaker association with bugs than that of traditional code smells.

Bavota et al. Bavota *et al.* (2015a) analyzed various versions of three Java open source projects to investigate whether refactorings applied to classes having a low level of maintainability and examined their ability to remove traditional OO code smells. The study shows that 42% of refactoring operations are performed on smelly-classes, only 7% of them removed code smells from the affected classes. Refactorings are detected at the release level using Ref-Finder tool Prete *et al.* (2010). Besides the low accuracy of Ref-Finder achieving an overall precision of 35% and an overall recall of 24% as shown in prior studies Hegedűs *et al.* (2018); Kádár *et al.* (2016); Soares *et al.* (2013), our experimental setup is quite different. We study Android apps while Bavota et al. Bavota *et al.* (2015a) focus on desktop apps as well as distinguishing between both Android specific and traditional OO smells. We also follow a fine grained analysis of refactoring changes at the commit level using RefactoringMiner Tsantalis *et al.* (2018b) instead of the release level to achieve fine grained analysis and higher level of precision and recall in the detected refactorings. Also, Saika et al. investigated the impact of the severity of code smells on refactoring Saika, Choi, Yoshida, Haruna & Inoue (2016). Their results show that refactoring did not decrease the severity of code smells significantly. In later work, Yoshida et al. Yoshida *et al.* (2016) revisited the relationship between code smells and refactoring in desktop applications. Their study identified various smell-refactoring patterns in practice and found some patterns that match with those in Fowler's catalog Fowler *et al.* (1999), while highlighting other new patterns of refactorings in the systems they studied. They found similar results indicating that the refactorings applied rarely correspond to the type of the code smells, as would be predicted by the patterns in Fowler's catalog.

Similarly, Cedrim and Garcia et al. Cedrim *et al.* (2017) analyzed how 16,566 refactorings of 10 different types affect the density of 13 types of code smell in the version histories of 23 projects. Their results reveal that 79.4% of refactorings touched smelly elements, 57% did not reduce smell occurrence, 9.7% of refactorings removed smells, and 33.3% introduced new ones. Fontana and Spinelli Fontana & Spinelli (2011) analyzed the impact of refactoring on quality metrics when applied to remove code smells. They selected four code smells and six quality metrics selected to evaluate the code and design quality of a system after refactoring. They found that the only metric, whose value increased is the Tight Class Cohesion (TCC) for all the refactoring applied for smell removal and thus suggest that following the advice of correcting smells by refactoring does not predictably improve other quality metrics. Tavares et al. Tavares, Bigonha & Figueiredo (2020) conducted an empirical study analyzing the impact of refactoring on code smells. They selected seven open-source Java systems, applied the Move Method, Replace Type Code and Replace Conditional refactorings and measured their impact on ten code smells detected by five different tools. They observed that refactorings may decrease, increase, or have neutral impact on the number of code smells. Table 1.5 summarizes the findings for the studied code smells according to previous works and to Fowler's book, highlighting for each smell the refactoring types that would be expected to be used to resolve the smell.

Table 1.5 Code smells investigated in this study and the refactorings that would be anticipated to be employed to resolve them
Adapted from Bavota et al. (2015, p.12), Fowler et al. (1999, p.63), Yoshida et al. (2016, p.3) and Ouni et al. (2016, p.7)

Code smells	Refactorings
Blob Class	Move Method, Extract Method, Inline Method, Extract And Move Method, Push Down Method Pull Up Method, Move Attribute, Push Down Attribute, Pull Up Attribute, Extract Super Class
Complex Class	Move Method, Extract Method, Extract And Move Method, Push Down Method, Pull Up Method, Move Attribute, Push Down Attribute, Pull Up Attribute
Feature Envy	Move Method, Extract Method, Extract And Move Method, Push Down Method Pull Up Method, Push Down Attribute, Pull Up Attribute
Lazy Class	Move Method, Extract And Move Method, Pull Up Method, Move Attribute, Pull Up Attribute
Long Method	Extract Method, Inline Method, Extract And Move Method
Long Parameter List	Extract Method, Extract And Move Method,
Message Chain	Move Method, Extract Method, Extract And Move Method
Refused Bequest	Move Method, Extract And Move Method, Push Down Method, Move Attribute, Push Down Attribute
Spaghetti Code	Extract Class, Extract And Move Method
Speculative Generality	Move Method, Inline Method, Extract And Move, Push Down Method, Pull Up Method, Move Attribute, Push Down Attribute, Pull Up Attribute

Other research efforts have been focusing on refactoring and code smells prioritization. Zhang et al. Zhang, Baddoo, Wernick & Hall (2011) provided a prioritization schema among six analyzed smells according to their association with software faults. Sharma et al. Sharma, Suryanarayana & Samarthyam (2015) studied refactoring adoption challenges in industry through a survey with developers and advocated the need to move software development toward more effective refactoring adoption. This study aligns reasonably well with our findings in the context of mobile software development. Bois and Mens Du Bois & Mens (2003) measured the relationship between five quality metrics and three refactoring types, Extract Method, Encapsulate Field, and Pull up Method. The study proposed a formalism based on the Abstract Syntax Tree (AST) representation of the source code, extended with cross-references to describe the impact of refactoring on only five metrics. The results of their work showed both positive and negative impacts on the studied metrics. Ouni et al. Ouni, Kessentini, Bechikh & Sahraoui (2015a) introduced a refactoring prioritization approach based on the severity and importance of smells to better manage the developers efforts.

While our study constitutes, to the best of our knowledge, the first empirical study that investigates the impact of refactoring on code smells in Android apps using association rule mining, we found a number of similarities between our results for OO code smells and prior works on OO smells:

- Similar to Bavota et al. Bavota *et al.* (2015a), Cedrim and Garcia et al. Cedrim *et al.* (2017) and Yoshida et al. Yoshida *et al.* (2016), we found that refactoring rarely removes code smells: While an average of 25% of refactorings are applied to OO code smells, only 5% remove these smells.
- Similar to Bavota et al Bavota *et al.* (2015a), code elements affected by *Feature Envy* and *Long Method* code smells are more likely to be refactored.
- Similar to Bavota et al. Bavota *et al.* (2015a) and Yoshida et al. Yoshida *et al.* (2016) (Table 1.5), we found that some pairs of code smells and refactorings co-occur together such as Long Method → Extract Method and Feature Envy → Move Method.

Synthesis: We observe from the existing literature that most studies focus on desktop applications while little knowledge is available for mobile apps. Existing studies are merely limited to some particular code smell types, few specific quality metrics, or/and few refactoring types. In our study, we focus on Android apps while considering the analysis of both Android-specific and traditional OO code smells. While current studies collect refactorings based on mining developers documentation, or release-based static analysis tools, we use a fine-grained detection of refactorings based on RefactoringMiner to reduce any any bias towards imprecise collection of refactorings.

1.2.4 Studies about the impact of refactoring on quality metrics

Several research efforts have focused on studying when and how to apply refactoring. Fowler defined the first refactoring catalog that contains 72 refactoring operations and specified a guide containing information on when and how to apply them Fowler *et al.* (1999). Later, Simon et al. Simon, Steinbruckner & Lewerentz (2001) presented a generic approach to generate visualizations that supports developers to identify bad smells and propose adequate refactorings. They focus on use relations to propose move method/attribute and extract/inline class refactorings. They define a distance-based cohesion metric, which measures the cohesion between attributes and methods with the aim of identifying methods that use or are used by more features of another class than the class that they belong to, and attributes that are used by more methods of another class than the class that they belong to. The calculated distances are visualized in a three-dimensional perspective supporting the developer to manually identify refactoring opportunities.

Various research works attempted to quantitatively evaluate whether refactoring indeed improves quality in traditional software systems. Alshayeb et al. Alshayeb (2009) investigated the impact of refactoring operations on five quality attributes, namely adaptability, maintainability, understandability, reusability, and testability. Their results highlight that benefits brought by refactoring operations on some code classes are often counterbalanced by a decrease of quality in some other classes. Pantiuchina et al. Pantiuchina, Lanza & Bavota (2018) explored the correlation between code metrics and the quality improvement explicitly reported by developers in commit messages. The study shows that quality metrics sometimes do not capture the quality improvement documented by developers. Similarly, AlOmar et al. AlOmar *et al.* (2019) conducted a large scale empirical study on open-source java projects to investigate the extent to which refactorings impact on quality metrics match with the developers perception. The study results indicate that quality metrics related to cohesion, coupling and complexity capture more developer intentions of quality improvement than metrics related to encapsulation, abstraction, polymorphism and design size

Tahvildari & Kontogiannis Tahvildari & Kontogiannis (2003) analyzed the association of refactorings with a possible effect on maintainability enhancements through refactorings. They use a catalogue of object-oriented metrics as an indicator for the transformations to be applied to improve the quality of a legacy system. The indicator is achieved by analysing the impact of each refactoring on these object-oriented metrics. Ó Cinnéide et al. Ó Cinnéide *et al.* (2012) investigated the impact of refactoring on five popular cohesion metrics using eight Java desktop systems. Their results demonstrate that cohesion metrics disagree with each other in 55% of cases. Furthermore, Geppert et al. Geppert, Mockus & Robler (2005) empirically investigated the impact of refactoring on changeability. They considered three factors for changeability: customer reported defect rates, effort, and scope of changes. Szoke et al. Szóke, Antal, Nagy, Ferenc & Gyimóthy (2014) performed a study on five software systems to investigate the relationship between refactoring and code quality. They show that small refactoring operations performed in isolation rarely impact software quality. On the other side, a high number of refactoring operations performed in block helps in substantially improving code quality.

Strogglos and Spinellis Stroggylos & Spinellis (2007) investigated the impact of refactoring on eight OO quality metrics. Their results indicate that refactoring caused a non-trivial increase in some specific metrics such as LCOM, Ca, RFC leading to less coherent classes or assigning more responsibilities to other classes. Kataoka et al. Kataoka, Imai, Andou & Fukaya (2002) studied the refactoring effect on various coupling metrics, comparing the metrics before and and after the refactorings Extract Method and Extract Class, which were performed by a single developer in desktop C++ programs. More recently, Cedrim et al. Cedrim *et al.* (2016) conducted a longitudinal study of 25 desktop projects to examine the impact of refactoring on software quality. The results indicate that only 2.24% of refactorings removed code smells while 2.66% of the refactorings introduced new ones.

Synthesis: We observe from the existing literature that most studies focus basically on desktop applications while little knowledge is available for mobile apps. Furthermore, existing studies are merely limited to some particular quality metrics, or/and few refactoring types. In our study, we focus on Android apps while considering the analysis of more quality metrics. While current studies collect refactorings based on mining developers documentation, or release-based static analysis tools, we use a fine grained detection of refactoring based on RefactoringMiner to reduce any bias towards imprecise collection of refactorings. Furthermore, one of the limitations in the state-of-the-art studies is that they do not consider that refactoring operations are typically accompanied with other code changes in either the commit Stroggylos & Spinellis (2007); AlOmar *et al.* (2019); Cedrim *et al.* (2016) or release levels Bavota *et al.* (2015a); Alshayeb (2009); Chávez, Ferreira, Fernandes, Cedrim & Garcia (2017). Such code changes can add more noise to the analyzed quality metrics values, and impact the final outcome the metrics analysis. In our study, we adopt a causal inference based on the DiD model Angrist & Pischke (2008) to better assess the impact of refactoring on quality metrics and assure that the metrics variations are due to refactoring.

1.3 Conclusion

Based on our literature review, we observe that we still lack knowledge about various aspects (i.e., refactoring, code smells and quality metrics) in the context of mobile apps. Most of the reviewed research works only studied the object-oriented systems. Thus, in our work, we explore mobile apps to address these knowledge lacks and provide necessary elements to understand these aspects . We start by addressing the lack of studies by presenting in the upcoming chapter an empirical study that explores the impact of refactoring on code smells in Android apps. We focus basically on Android apps while considering the analysis of both Android-specific and traditional OO code smells.

CHAPTER 2

A LONGITUDINAL STUDY OF THE IMPACT OF REFACTORING IN ANDROID APPLICATIONS

Oumayma Hamdi[a] , Ali Ouni[a] , Mel Ó Cinnéide[b] , Mohamed Wiem Mkaouer[c]

[a] Department of Software and IT Engineering, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

[b] Department of Software Engineering, University College Dublin,
Dublin, Ireland,

[c] Department of Software Engineering, Rochester Institute of Technology, Rochester,
NY, USA,

Paper published in *the Information and Software Technology journal (IST)*, July 2021

2.1 Abstract

Context: Mobile applications have to continuously evolve in order to meet new user requirements and technological changes. Addressing these constraints may lead to poor implementation and design choices, known *code smells*. Code refactoring is a key practice that is employed to ensure that the intent of a code change is properly achieved without compromising internal software quality. While previous studies have investigated the impact of refactoring on traditional code smells in desktop applications, little attention has been paid to the impact of refactoring activities in mobile application development.

Objective: We aim to develop a broader understanding of the impact of refactoring activities on Android and traditional code smells in Android apps.

Method: We conduct a longitudinal empirical study by analyzing the evolution history of five open-source Android apps comprising 652 releases and exhibiting a total of 9,600 refactoring operations. We consider 15 common Android smell types and 10 common traditional Object-

Oriented (OO) code smell types to provide a broad overview of the relationship between refactoring and code smells.

Results: We find that code smells are widespread across Android applications, but smelly classes are not particularly targeted by refactoring activities and, when they are, it is rare for refactoring to actually remove a smell.

Conclusions: These somewhat surprising results indicate that it is critical to understand better the real quality issues that Android developers face, and to develop a model of code smells and refactoring that can better address their needs in practice.

2.2 Introduction

Android is the dominant operating system for mobile devices Research (2020) and currently runs on approximately 86% of smart phones worldwide, with over 2.5 billion monthly active users [3]. As a consequence of this popularity, the Android app marketplace is extremely congested and strong competition makes it necessary to build mobile apps rapidly and evolve them continuously to meet the needs of users. These rapid changes may lead to poor design and implementation choices, which are manifested in *code smells* Cedrim *et al.* (2017); Bavota *et al.* (2015a); Palomba, Oliveto & De Lucia (2017b); Ouni, Kessentini, Sahraoui & Boukadoum (2013). To make matters more challenging, a new category of smells, known as *Android smells*, has emerged in Android apps Reimann *et al.* (2014); Palomba *et al.* (2017a, 2019); Hecht, Moha & Rouvoy (2016); Carette, Younes, Hecht, Moha & Rouvoy (2017). The presence of traditional and Android code smells can degrade the quality of Android apps, reduce their performance and hinder their evolution Palomba *et al.* (2019); Hecht *et al.* (2016); Morales, Saborido, Khomh, Chicano & Antoniol (2017).

Refactoring is seen as a possible solution to this problem. Supporters claim that it helps to improve software quality and remove code smells by reorganizing and/or cleaning code fragments Fowler *et al.* (1999); AlOmar *et al.* (2019); Ouni, Kessentini, Sahraoui, Inoue & Deb (2016);

[3] https://venturebeat.com/2019/05/07/android-passes-2-5-billion-monthly-active-devices

Mkaouer *et al.* (2015). However, many studies dispute this claim Chatzigeorgiou & Manakos (2010); Bavota *et al.* (2015a), finding that refactoring practices may present different challenges in the context of Android apps due to the rapid evolution of Android apps, their short release deadlines, small code base, and heavy reuse of external libraries and classes Minelli (2012); Ruiz, Nagappan, Adams & Hassan (2012); Xu, Wu & Chen (2013). Indeed, the characteristics of mobile platforms, the constraints on resources (memory, CPU, varying screen sizes, etc.) as well as the highly dynamic nature of the mobile app market and its volatile user needs can play an important role in mobile app development and evolution. Unlike object-oriented (OO) software systems AlOmar *et al.* (2019); Ó Cinnéide *et al.* (2012); Alshayeb (2009); Bavota *et al.* (2015a); Cedrim *et al.* (2016); Stroggylos & Spinellis (2007); Tahir *et al.* (2020), the impact of refactoring on code smells in mobile apps has received little attention. Hence, much uncertainly exists about the relationship between refactoring and code smells/quality aspects in mobile apps.

To develop practical and reliable refactoring tools for mobile apps, practitioners and researchers need to understand current refactoring practices and their impact on code smells. However, to the best of our knowledge, little has been reported about the effects of refactoring on code smells in mobile apps.

In order to address this gap and to improve current knowledge about the impact of refactoring on code smells in Android apps, we performed a study of refactoring application and smell removal in five open-source Android applications. We analyze the impact of commonly used refactorings on 10 common OO code smells and 15 Android-specific smells.

In particular, we investigate whether developers apply refactoring to smelly code fragments and how effective refactoring is at removing code smells. To collect our datasets, we extracted a total of 9,600 applied refactoring operations along with commit history, as well as all OO and Android smell instances.

We provide our replication package containing the collected data, the obtained results as well as the used scripts publicly available for future replications and extensions dataset (2020). In particular, our study addresses the following research questions.

RQ1: *How prevalent are code smells in Android applications?*

RQ2: *To what extent do developers apply refactorings to smelly classes?*

RQ3: *To what extent do refactoring operations applied to smelly classes removecode smells?*

2.3 Empirical Study Design

The *aim* of this study is to investigate the potential relationship that refactoring, occurring in the development of an Android app, can have with code smells. Software refactoring might interfere in the presence of smelly or low quality code elements. We first study the prevalence of code smells in Android apps, then we analyze if the presence of code smells or low quality code elements represent a trigger for refactoring. Thereafter, as a consequence, we assess the effectiveness of refactoring at removing code smells and improving quality.

2.3.1 Goals and Research Questions

Our study aims at addressing the following research questions.

- **RQ1**. **How prevalent are code smells in Android applications?** This preliminary research question aims to determine the prevalence of smells in Android apps and to identify the most frequently-occurring Android and OO smells.
- **RQ2**. **To what extent do developers apply refactorings to smelly classes?** This research question investigates how Android developers practice refactoring and which particular smells are more or less likely to undergo refactoring.
- **RQ3**. **To what extent do refactoring operations applied to smelly classes remove code smells?** The goal of RQ3 is to investigate the efficiency of refactorings in fixing code smells in Android apps. It enables a better understanding of the relationship between refactoring and Android and traditional OO smells.

2.3.2 Study steps and data collection

This section presents all phases of the study designed to answer our research questions. The process is represented in Figure 2.1 and comprises a sequence of five main steps, (1) Android apps selection, (2) refactoring extraction, (3) commit extraction and (4) Android and traditional code smells detection.

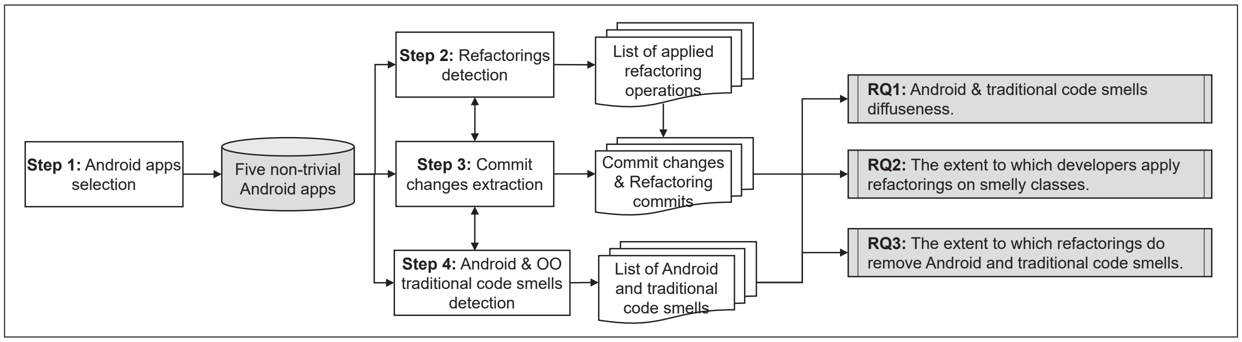
Step 1: Android apps selection
Five non-trivial Android apps
Step 2: Refactorings detection
Step 3: Commit changes extraction
Step 4: Android & OO traditional code smells detection
List of applied refactoring operations
Commit changes & Refactoring commits
List of Android and traditional code smells
RQ1: Android & traditional code smells diffuseness.
RQ2: The extent to which developers apply refactorings on smelly classes.
RQ3: The extent to which refactorings do remove Android and traditional code smells.

Figure 2.1 Overview of our study

2.3.2.1 Step 1: Selection of subject applications

The first step in our study involves selecting experimental Android apps from GitHub to obtain a study sample. The selected Android app projects, written in Java, cover different application domains, are of different sizes and have a varying number of contributors. Moreover, all the selected apps have (1) over 1,000 commits to ensure that they are actively maintained (i.e., no single commit or toy project), (2) over 100,000 user installations from Google Play Store, and (3) over 100 stars on GitHub to ensure that the studied apps are non-trivial ones Munaiah, Kroh, Cabrey & Nagappan (2017). We then randomly selected five apps from this set to be the subjects of our study. This sample size is larger than the size of samples used in related studies on refactoring and code smells Bavota *et al.* (2015a); Chatzigeorgiou & Manakos (2010); Tsantalis, Guana, Stroulia & Hindle (2013); Stroggylos & Spinellis (2007).

Table 2.1 presents a summary of the selected apps. The apps consist of a total of 20,132 commits and 652 releases from the open source Android apps *Apg*, *NetGuard*, *Omni-Notes*, *Congress* and *Notepad*. *Apg*[4] is the Android Privacy Guard application that originally brought email encryption to the Android platform. *NetGuard*[5] provides simple and advanced ways to block access to the Internet through WiFi and mobile connection. *Omni-Notes*[6] is a note taking app that provides a simple interface while keeping smart behavior. *Congress*[7] is an app for tracking U.S. Congress news and updates. Finally, *Notepad*[8] is an outliner application for taking notes and managing to-do lists.

Table 2.1 Characteristics of the analyzed Apps

App	#Commits	#Releases	#Classes	LOC	#Refactorings	#OO smells	#Android smells
Apg	4,367	3	135−272	18,735−37,471	2,921	286−533	212−377
NetGuard	3,566	365	6−41	528−11,099	2,333	5−132	6-72
Omni-Notes	2,988	127	164−190	14,494−15,875	1,983	217−482	125−142
Congress	1,689	59	69−88	8,892−10,358	1,523	54−85	74−111
Notepad	1,522	98	73−161	22,433−29,294	840	436−605	119−227

2.3.2.2 Step 2: Refactoring collection

In this step, we collect all the refactoring operations applied to the studied apps. We use RefactoringMiner Tsantalis *et al.* (2018b) to support the detection of the applied refactoring instances. RefactoringMiner is a command-line based open source tool [9] that is built on top of the UMLDiff Xing & Stroulia (2005) algorithm for differencing object-oriented models. RefactoringMiner has been reported to achieve a precision of 98% and recall of 87% Tsantalis *et al.* (2018b); Silva, Tsantalis & Valente (2016). The tool walks through the commit history of a project's Git repository to extract refactorings between consecutive commits. RefactoringMiner supports the detection of various common refactoring types from Fowler's catalog. From the supported refactorings, our extraction process identifies 12 different common refactoring types as presented earlier in Table 1.1 (See section 1.1.2).

[4] https://github.com/thialfihar/apg

[5] https://github.com/M66B/NetGuard

[6] https://github.com/federicoiosue/Omni-Notes

[7] https://github.com/konklone/congress-android

[8] https://github.com/spacecowboy/NotePad

[9] https://github.com/tsantalis/RefactoringMiner

2.3.2.3 Step 3: Commit changes extraction

After the extraction of all refactoring operations, we collect the IDs of all refactoring commits, i.e., commits in which a refactoring operation was applied, as well as the IDs of the commits that immediately precede the refactoring commit. The GitHub API facilitates this; in particular, we use the `git clone` command to download the source code of each refactoring commit as well as its immediately preceding commit. These commits enable the identification of the code smells before and after the application of refactoring in Steps 4 and 5.

2.3.2.4 Step 4: Detection of Android and traditional OO smell instances

In this step we identify smells in the source code of the apps. For each refactoring commit, we identify the presence of any instance of OO or Android smells at the class level.

Android-specific smells: Table 1.3 lists the Android code smells employed in this study (See section 1.1.3). We use aDoctor [10], a command-line based tool that implements rules provided by Palomba et al. Palomba *et al.* (2017a) to identify common Android smells. The tool achieves a high detection precision of 98%, and a high recall of 98%, as reported in Palomba et al. Palomba *et al.* (2017a).

Traditional OO code smells: We consider a set of 10 common types of traditional code smells as defined by Fowler Fowler *et al.* (1999) and Brown et al. Brown *et al.* (1998a) that have been widely studied in prior works Palomba *et al.* (2018b, 2017b); Bavota *et al.* (2015a); Ouni *et al.* (2016); Ouni, Kessentini, Sahraoui, Inoue & Hamdi (2015b). These smells have varying characteristics, e.g., classes characterized by long/complex code as well as violation of accepted OO design and implementation principles, such as SOLID Joshi (2016) and GRASP Sons (1999). We used an existing code smell detection tool, Organic [11], that has been used in prior studies by Bavota et al. Bavota *et al.* (2015a); Palomba *et al.* (2017b, 2018b). The list of supported OO smells is provided in Table 1.2 (See section 1.1.3). We selected the Organic tool as it (1)

[10] https://github.com/fpalomba/aDoctor

[11] https://github.com/opus-research/organic

supports the detection of all these common code smells unlike other specialized tools that focus on the detection of limited subset of smells, (2) provides a Command Line Interface to collect code smells from the project's commits which is suitable in the context of our study, (3) is based on a set of simple detection rules Bavota *et al.* (2015a) for OO code smells yielding a high accuracy: 72% precision and 81% recall on average, and (5) has been widely used in recent work Palomba *et al.* (2018b,a, 2017b); Bavota *et al.* (2015a).

2.3.3 Experimental Data Analysis

Analysis method for RQ1. To answer *RQ1*, we use the collected Android and traditional OO smells for each application as described in *Step 1.* Then, for each type of Android and traditional OO smell we calculate its prevalence, i.e., the ratio of the number of classes exhibiting the smell to the total number of classes. Furthermore, we analyze the distribution of each smell type with respect to the total number of smell instances in the studied apps.

Analysis Method for RQ2. To answer *RQ2*, we study the relationship between the refactoring operations and smelly classes from both Android and traditional code smells. In particular, for each refactoring operation, we analyze whether the refactoring has been applied to a smelly or non-smelly class. Our aim is to discover the relationship between the refactoring operations and the smell types. For this we employ the Apriori association rule mining algorithm Agrawal, Srikant *et al.* (1994), which has been successfully used to mine association between items in many problems Kamei *et al.* (2012); AlZu'bi, Hawashin, EIBes & Al-Ayyoub (2018); Palomba *et al.* (2017b); Kaur & Kang (2016); Agarwal (2017); Muse *et al.* (2020).

An association rule is defined as an implication in the form X $\Rightarrow$ Y, where X, Y $\subseteq$ I and X $\cap$ Y = $\emptyset$. Let $I = \{i_1, i_2, ..., i_n\}$ be a set of n items, and $T = \{t_1, t_2, ..., t_m\}$ a set of m the transactions. In our study, T is the set of classes present in each refactoring commit, and each item in the set I indicates the presence of a specific smell or a specific refactoring in that commit. Hence, an association rule translates a co-occurrence between a refactoring operation r_i and a specific smell instance s_i on the same class. Specifically, the association rule is written as follows:

$$\text{Smell}(s_i) \Rightarrow \text{Refactoring}(r_j)$$

To measure the strength of an association between two items, i.e., smell and refactoring, we use the support Agrawal, Imielinski & Swami (1993), confidence Agrawal *et al.* (1993), Lift Piatetsky-Shapiro (1991), Leverage Piatetsky-Shapiro (1991), and conviction Piatetsky-Shapiro (1991) metrics. The *support* measures the frequency in which a refactoring and a smell occur in the dataset, and has a range of [0..1]. The *confidence* implies how often (i.e., probability) a smell exhibits a refactoring operation with values varying from 0 to 1. To ensure that both smell and refactoring are not associated by chance, the *lift* is used to measure the dependence ratio. The range of values for the lift is between 0 and $+\infty$. When the lift value is greater than 1, it implies that the pair of smell-refactoring is highly correlated. Furthermore, the *conviction* measures the probability of a smell occurring without being refactored, returning a value within a range of 0 to $+\infty$. When the conviction score is equal to 1, this implies that the smells are independent. Finally, the *leverage* tests the difference between two smell-refactoring pairs support score with values ranging from -1 to 1. A leverage of 0 indicates total independence between both considered smell and refactoring.

Moreover, to get statistical evidence, we used Cramer's V test Cramir (1946), which measures how strongly two categorical variables are associated. A value of 0 indicates complete independence, and a value of 1 indicates complete association. Cramer's V test takes into account sample size when comparing two variables and is used as post-test to determine strengths of association based on the Pearson's chi-square coefficient that checks for its significance. The formula is given in Equation 3.4 :

$$V = \sqrt{\frac{\chi^2}{n \times min(row - 1, col - 1)}} \tag{2.1}$$

where χ^2 is the Pearson's Chi-square coefficient, n is the total number of samples and row and col represent the number of distinct values of the categorical variables whose association is to be computed.

Analysis method for RQ3. To answer *RQ3*, we analyze the co-occurrences between refactoring operations and smelly classes that are fixed after the application of a refactoring operation. We use the same analysis metrics used for RQ2 in terms of support, confidence, lift, leverage and conviction as well as the Cramer's V test. As we are interested for this research question in smelly classes that are fixed after refactoring, we consider only smelly classes where the smell is corrected after the application of a refactoring, instead of considering all smelly classes as analyzed in RQ2.

2.3.4 Replication package

We provide our collected data and results publicly available for future replications and extensions dataset (2020). We provide (*i*) the list of refactorings, traditional OO smells, Android smells, before and after the application of refactoring, (*ii*) data collection and analysis scripts.

2.4 Empirical Study Results

This section describes and discusses the results of our investigations.

2.4.1 RQ1. How prevalent are code smells in Android applications?

We provide the prevalence results for both Android and traditional OO code smells. Figures 2.2 and 2.3 report the distribution of Android and traditional OO smells, respectively, in the analyzed applications.

2.4.1.1 Android smells

Table 2.2 reports the prevalence of the Android smells in terms of the number and the percentage of classes affected by each smell type in each individual Android app as well as in the combined dataset, i.e., all apps. We can see from the table that 68% of all classes are affected by Android smells. We also observe a variability in terms of prevalence of the different types of Android smells (As highlighted with blue shades in Table 2.2). For instance, the most prevalent smell across all apps is the *Member Ignoring Method* (MIM) which affects, on average, 55% of all classes in the studied apps. On the other side, we observe that some smells are moderately prevalent such as *No Low Memory Resolver* (NLMR) while others are slightly prevalent such as *Leaking Thread* (LT). It is worth noting that some smells do not affect any class such as *Debuggable Release*. Overall, we notice that the highly and moderately prevalent Android smells in our benchmark are in line with the findings of Palomba et al. Palomba *et al.* (2019).

Furthermore, Figure 2.2 reports the smells occurrences, i.e., frequency. The total number of Android smell instances in our analyzed apps is 929 instances, with the three most frequent Android smells being *Member Ignoring Method* (MIM), *No Low Memory Resolver* (NLMR), and *Slow Loop* (SL). Overall, these three highly frequent smells account for 74% of the total number of Android smells. We can also observe a number of moderately frequent smells such as the *Leaking Inner Class* (LIC) and a set of infrequent smells such as the *Internal Getter and Setter* (IGS). It is worth noting that four smells do not occur in any of the apps, such as *Public Data* (PD). Unsurprisingly, the more frequent a smells is, the more prevalent it is.

Table 2.2 The prevalence of Android smells in the studied applications

Application	Code smell	MIM	NLMR	SL	LIC	UC	DTWC	LT	IGS	IDS	DW	RAM	DR	IDFP	ISQLQ	All smells
Apg	Classes affected	190	53	45	29	29	14	15	0	2	0	0	0	0	0	217
	percentage(%)	70	19	17	11	11	5	6	0	1	0	0	0	0	0	80
NetGuard	Classes affected	27	11	5	6	7	7	5	1	1	2	0	0	0	0	31
	percentage(%)	66	26	12	15	17	17	12	2	2	5	0	0	0	0	76
Omni-Notes	Classes affected	73	19	14	10	6	15	1	4	0	0	0	0	0	0	96
	percentage(%)	38	10	7	5	3	8	1	2	0	0	0	0	0	0	51
Congress	Classes affected	37	36	18	14	2	0	2	1	0	0	1	0	0	0	61
	percentage(%)	42	41	20	16	2	0	2	1	0	0	1	0	0	0	69
Notepad	Classes affected	86	35	28	27	23	13	8	2	3	2	0	0	0	0	106
	percentage(%)	53	22	17	17	14	8	5	1	2	1	0	0	0	0	66
Combined	**Classes affected**	**413**	**154**	**110**	**86**	**67**	**49**	**31**	**8**	**6**	**4**	**1**	**0**	**0**	**0**	**511**
	percentage(%)	**55**	**21**	**15**	**12**	**9**	**7**	**4**	**1**	**1**	**0.5**	**0.1**	**0**	**0**	**0**	**68**

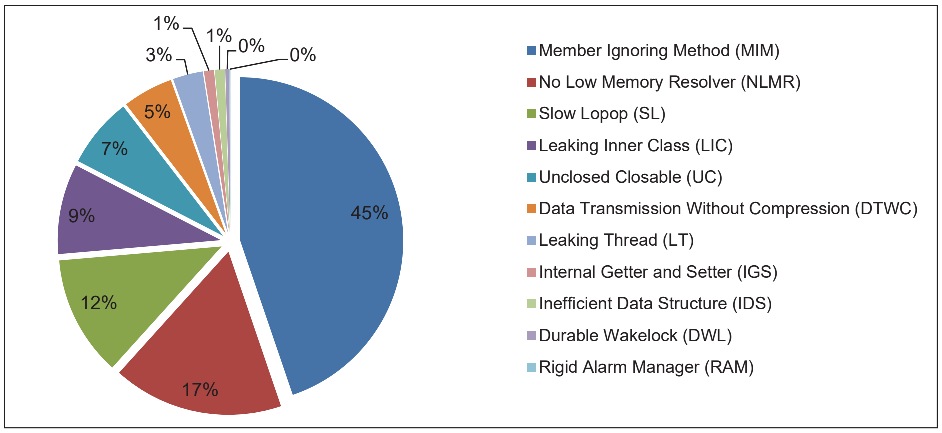

Figure 2.2 Distribution of Android smells.

2.4.1.2 Traditional OO smells

The results of the prevalence of the traditional smells are reported in Table 2.3. We found that 63% of classes are affected by traditional smells, with the three most prevalent smells being *Long Method (LM)*, *Message Chain (MC)*, and *Long Parameter List (LPL)*. Moreover, we observe some moderately prevalent smells like the *Complex Class (CC)* and some slightly prevalent smells such as the *Speculative Generality*.

Regarding smell frequency, the total number of traditional OO code smell instances detected in our benchmark is 1,871. As shown in Figure 2.3, the most frequent smells are *Message Chain* (MC), *Long Method* (LM) and *Long Parameter List* (LPL). On the other hand, we observe that some smells are moderately diffused such as the *Feature Envy* while others are less frequent such as *Spaghetti Code*.

In general, our results differ from the prevalence of OO smells in desktop applications as reported in recent studies Palomba *et al.* (2018a). In contrast to Android apps, *Feature Envy*, *Message Chain* and *Lazy Class* are not prevalent in desktop applications while *Spaghetti Code*, *Speculative Generality* and *Refused Bequest* are quite prevalent Palomba *et al.* (2018a). It is worth noting that our findings in Android apps share some similarities with desktop apps for the *Long Method* and *Complex Class* which are quite prevalent in both Android and desktop applications.

Table 2.3 The prevalence of each traditional OO code smell across the studied applications

Application	Code smell	LM	MC	LPL	CC	FE	LC	SC	BC	SG	RB	All smells
Apg	Classes affected	93	47	55	54	58	21	17	12	11	6	177
	percentage(%)	34	17	20	20	21	8	6	4	4	2	65
NetGuard	Classes affected	17	10	6	17	6	7	10	6	0	0	26
	percentage(%)	41	24	15	41	15	17	24	15	0	0	63
Omni-Notes	Classes affected	15	48	24	13	23	35	3	5	2	0	112
	percentage(%)	8	25	13	7	12	18	2	3	1	0	59
Congress	Classes affected	9	8	20	9	9	10	1	4	1	10	55
	percentage(%)	10	9	23	10	10	11	1	5	1	11	63
Notepad	Classes affected	33	39	31	24	18	21	11	7	10	7	100
	percentage(%)	21	24	19	15	11	13	7	4	6	4	62
Combined	**Classes**	**167**	**152**	**136**	**117**	**114**	**94**	**42**	**34**	**24**	**23**	**470**
	Percentage (%)	**22**	**20**	**18**	**16**	**15**	**13**	**6**	**5**	**3**	**3**	**63**

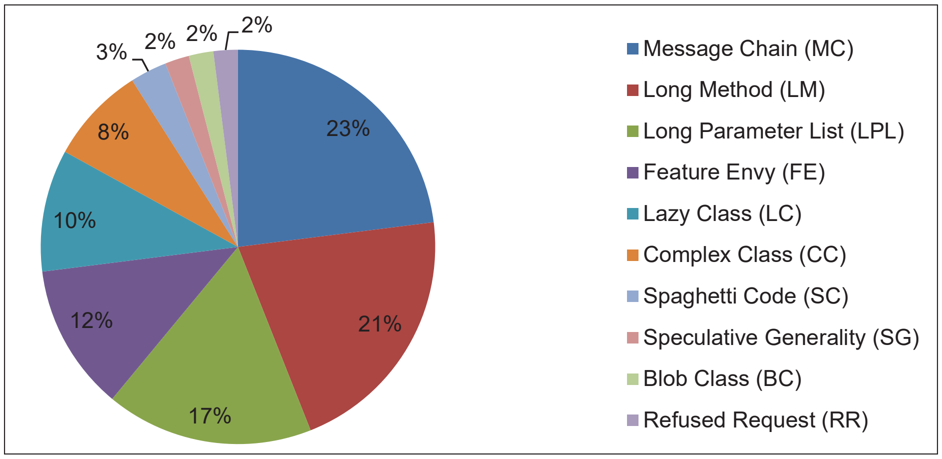

Figure 2.3 Distribution of Distribution of traditional OO smells

Summary for RQ1. Code smells are remarkably prevalent in Android apps. Android smells affect an average of 68% of classes with Member Ignoring Method (MIM), No Low Memory Resolver (NMR) and Slow Loop (SL) being the most prevalent smells. Traditional OO code smell are also remarkably prevalent in Android apps affecting an average of 63% of classes with Long Method (LM), Message chain (MC) and Long Parameter List (LPL) being the most prevalent smell types.

Results show that both Android and traditional OO smells are remarkably prevalent in Android applications. This widespread prevalence suggests either developer unawareness about these code smells, or that developers simply do not regard these code smells to be important. To explore this further, we now explore in RQ2 to what extent developers apply refactorings to smelly classes.

2.4.2 RQ2. Which code smells co-occur together?

As described earlier Section 2.3.2.2 and Table 2.1, the total number of applied refactorings is 9,600. To study the extent to which developers apply refactorings to Android and traditional OO smells, we first use the Apriori algorithm to determine the associations between the different applied refactoring operations and smell types. To generate frequent itemsets, we selected a minimum confidence of 0.5 and a minimum lift threshold of 1 to generate the relevant association between smells and refactorings. We also restrict the maximum number of items in each itemset to two since we are interested in the association between one smell type and one refactoring type. In the following sections, we discuss the results for both traditional and Android code smells.

2.4.2.1 Traditional OO smells

Table 2.4 reports the number of refactorings applied to each traditional OO smell type as well as the total number of refactorings applied to both smelly and non-smelly classes. Overall, we found that the total number of refactorings applied to classes containing traditional OO smells is 2,436 out of a total of 9,600 refactorings which accounts for 25% of the total number of refactorings, while the remaining 7,164 refactorings (75%) are applied to classes not impacted by traditional OO smells.

We also observe that *Extract Method*, *Move Method*, and *Rename Method* are the most applied refactorings accounting, respectively, for 963, 500, and 477 refactorings. The *Long Method* and *Feature Envy* code smells are the most refactored smells with 629 (26.47%) and 616 (25.92%) refactoring operations, respectively. It is worth noting that the relatively high number of *Extract*

Method refactorings may result in a high number of *Rename Method* refactorings. Indeed, methods that undergo an *Extract Method* refactoring may also undergo a *Rename Method* to reflect its new purpose. We also found that some smells are very rarely refactored including *Speculative Generality* (33 refactorings), *Lazy Class* (32 refactorings) and *Refused Bequest* (13 refactorings).

Looking at the smell-refactoring pairs (i.e., mined associations), Table 2.5 shows the frequent itemsets for the OO smells and refactorings for each studied app, where each itemset comprises one OO smell and one refactoring. Note that only the pairs that appear in bold in the table are statistically significant based on their Chi-square p-value, while other pairs (not in bold) are considered to be due to chance. We observe that some associations are consistent across various apps, while others differ from one app to another.

Table 2.4 The type and number of refactorings performed on traditional OO code smells in the studied applications

Refactoring	App	LC	CC	LPR	FE	LM	BC	MC	RB	SC	SG	Total Smelly	Total Non-Smelly
Extract Method	Apg	2	50	37	72	106	12	20	1	29	8	337	523
	congress	4	33	12	16	75	1	5	0	0	0	146	405
	Netguard	2	13	0	100	87	3	28	0	8	0	241	622
	Notepad	0	46	4	43	6	5	15	0	0	0	119	196
	Omni-Notes	2	3	0	2	24	3	36	0	50	0	120	554
Move Method	Apg	0	38	9	46	28	6	6	0	11	0	144	183
	congress	7	5	15	14	4	1	12	0	0	0	58	255
	Netguard	1	26	0	102	1	0	2	0	0	0	132	243
	Notepad	0	12	0	40	2	2	3	0	0	0	59	96
	Omni-Notes	0	24	6	4	28	2	26	3	14	0	107	210
Extract and Move Method	Apg	0	13	4	11	13	3	4	0	6	0	53	180
	congress	2	1	5	1	1	0	1	0	0	0	11	155
	Netguard	2	1	1	0	3	0	1	0	1	0	9	190
	Notepad	0	0	0	0	0	0	2	0	0	0	2	118
	Omni-Notes	4	19	10	18	23	0	12	0	11	0	97	300
Inline Method	Apg	0	4	0	4	4	3	4	0	2	0	21	95
	congress	0	0	6	12	13	0	0	0	6	0	37	87
	Netguard	0	0	0	0	3	0	0	0	1	0	4	82
	Notepad	1	0	0	0	0	0	0	0	0	0	1	22
	Omni-Notes	0	4	7	4	4	1	4	0	2	0	26	86
Rename Method	Apg	1	39	33	37	76	4	15	0	13	7	225	354
	congress	1	17	6	4	16	4	6	0	0	0	54	261
	Netguard	1	4	0	5	33	0	10	0	8	0	61	291
	Notepad	0	20	10	20	24	1	20	1	16	0	112	252
	Omni-Notes	0	1	0	1	11	0	12	0	0	0	25	267
Move Attribute	Apg	0	10	7	6	8	4	2	0	5	0	42	170
	congress	0	4	0	36	10	2	16	0	0	0	68	476
	Netguard	0	0	0	0	3	0	1	0	1	0	5	93
	Notepad	0	0	0	0	0	1	1	0	0	0	2	46
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	13
Pull Up Method	Apg	0	1	0	3	1	0	3	0	0	1	9	131
	congress	0	0	0	0	0	0	0	0	0	0	0	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	0
Pull Up Attribute	Apg	0	6	2	8	6	0	0	0	4	0	26	72
	congress	0	0	0	0	0	0	0	0	0	0	0	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	1	0	0	0	0	0	0	0	0	1	49
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	0
Push Down Method	Apg	0	4	2	2	3	3	1	1	0	1	17	20
	congress	0	0	0	0	0	0	0	0	0	0	0	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	0
Rename Class	Apg	0	0	0	0	0	0	0	0	0	0	0	1
	congress	0	0	2	0	0	0	0	6	0	0	8	5
	Netguard	0	1	0	0	4	0	0	0	1	0	6	7
	Notepad	0	0	0	0	0	0	0	0	0	0	0	2
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	5
Extract Super Class	Apg	2	6	4	5	9	0	3	1	1	12	43	39
	congress	0	0	0	0	0	0	0	0	0	0	0	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0
	Omni-Notes	0	0	2	0	0	0	2	0	0	0	4	8
Move Class	Apg	0	0	0	0	0	0	0	0	0	0	0	0
	congress	0	3	0	0	0	0	0	0	1	0	4	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0
	Omni-notes	0	0	0	0	0	0	0	0	0	0	0	0
Total		**32**	**409**	**184**	**616**	**629**	**61**	**273**	**13**	**191**	**33**	**2,436**	**7,164**

Table 2.5 The top traditional OO code smells and refactoring pairs in each application based on confidence and lift values

App	Item pairs	Support	Confidence	Lift	Leverage	Conviction	Chi-square P-value	Cramer's V
Congress	Lazy Class → Extract Method	0.06	**0.54**	**4.682**	0.060	3.437	0.785	0.004
	Lazy Class → Rename Method	0.08	**0.8**	**4.712**	0.062	3.706	0.663	0.004
	Lazy Class → Move Method	0.08	**0.75**	**4.714**	0.061	3.506	0.121	0.014
	Complex Class → Extract Method	0.35	**0.62**	**4.426**	0.028	1.811	**<0.05**	**0.017**
	Long Method → Extract Method	0.34	**0.63**	**6.137**	0.025	1.792	**<0.05**	**0.013**
Netguard	Lazy Class → Extract and Move Method	0.11	**0.51**	**1.149**	0.014	1.402	1	0
	Complex Class → Rename Method	0.1	**0.77**	**4.711**	0.082	3.633	1	0
	Feature Envy → Extract Method	0.16	**0.97**	**1.366**	0.022	1.175	0.076	0.046
	Spaghetti Code → Extract Method	0.02	**0.56**	**3.887**	0.023	2.171	**<0.05**	**0.094**
Apg	**Complex Class → Extract Method**	0.09	**0.53**	**1.790**	0.047	1.654	**<0.05**	**0.029**
	Complex Class → Move Method	0.45	**0.64**	**2.084**	0.007	0.903	**<0.05**	**0.017**
	Complex Class → Pull Up Attribute	0.16	**0.5**	**1.148**	0.033	0.828	0.107	0.007
	Complex Class → Push Down Method	0.06	**0.5**	**1.185**	0.033	0.832	**<0.05**	**0.011**
	Complex Class → Extract and Move Method	0.26	**0.62**	**1.140**	0.006	0.888	**<0.05**	**0.019**
	Feature Envy → Move Method	0.45	**0.64**	**1.548**	0.024	0.897	**<0.05**	**0.013**
	Feature Envy → Pull Up Attribute	0.22	**0.69**	**1.234**	0.034	0.832	**<0.05**	**0.011**
	Feature Envy → Extract and Move Method	0.29	**0.71**	**1.405**	0.030	0.862	**<0.05**	**0.013**
	Long Method → Extract Method	0.34	**0.63**	**7.268**	0.392	0.357	**<0.05**	**0.021**
	Long Method → Move Method	0.59	**0.83**	**1.248**	0.149	2.287	**<0.05**	**0.024**
	Long Method → Pull Up Attribute	0.24	**0.75**	**1.948**	0.297	0.952	0.587	0.002
	Long Method → Extract and Move Method	0.28	**0.67**	**1.567**	0.033	1.398	**<0.05**	**0.012**
Notepad	Complex Class → Rename Method	0.01	**0.64**	**1.450**	0.082	1.684	1	0.002
	Feature Envy → Move Method	0.01	**0.54**	**1.214**	0.026	0.932	0.08	0.028
Omni-Notes	**Complex Class → Extract Method**	0.02	**0.56**	**4.103**	0.037	0.264	**<0.05**	**0.071**

For example, *Complex Class* is frequently refactored in all the studied apps. We observe that developers applied various refactoring operations to classes exhibiting this code smell, including *Extract Method*, *Move Method*, *Extract and Move Method*, *Rename Method*, and *Push Down Method*. This result aligns with Fowler's refactoring catalog Fowler *et al.* (1999), which states that complex classes are typically composed of several complex and/or long methods that could be fixed generally through any of these refactorings.

We also see that *Long Method* is involved in various refactorings, mainly *Extract Method* and *Move Method* in both the Congress and Apg apps. *Lazy Class* is often involved in various refactoring operations such as *Extract Method*, *Rename Method*, *Move Method* and *Extract and Move Method* in both Congress and Netguard apps. However, it is rarely refactored in the other apps. As for other code smells such as *Message Chain*, *Speculative Generality*, and *Long Parameter List*, we did not observe any particular result. These code smells could be fixed by various types of refactorings depending on the context Fowler *et al.* (1999); AlOmar *et al.* (2019); Ouni *et al.* (2016, 2015b) and may attract less attention from developers.

An interesting observation is that some code smells attract various refactorings in the same app depending on the context. On the other hand, we observe that the same refactoring is sometimes applied to several code smells. This makes the relationship between code smells and refactorings a *many-to-many* relationship. For example, Lazy Class is involved in three refactoring types in the congress app, while Complex Class is involved in five refactoring types as can be seen in Table 2.5. We also see that the refactoring *Extract Method* is applied to several code smell types including the *Lazy Class*, *Complex Class*, *Long Method*, and *Feature Envy*. This result also suggests that refactoring strategies depend on the context where they are applied and there is no specific refactoring that can be universally employed to fix a specific code smell.

We also further investigate statistically the association between smells and refactorings. Employing a significance level of ≤ 0.05 in Table 2.5, we observe a significant association between smells and refactorings. Furthermore, we can determine the degree of associations using Cramer's V test. We can see from the table that not all smell-refactoring association are statistically significant. The top association is *Spaghetti Code* → *Extract Method* having a highest Cramer's V score of 0.094. This finding is interesting as developers tend to apply *Extract Method* to reduce spaghetti code code fragments. We also observe that the associations with Complex Class are statistically significant most of the time. Indeed, complex classes are hard to maintain and are challenging to refactor. Typical refactorings include *Extract Method* and *Move Method* which can reduce the total number of methods having high complexity.

We also observe that not all these associations are statically significant and are due to chance (i.e., their chi-square p-value > 0.05). Among these associations, we find Lazy Class → Extract Method and *Lazy Class → Move Method* in the Congress app, *Lazy Class → Extract And Move Method* in the Netguard app which are aligned with the suggestions by Fowler. Indeed, this aligns with our overall perspective that the refactorings applied are not usually targeting the smells.

2.4.2.2 Android smells

Table 2.6 reports the number of refactorings applied to each Android smell type as well as the total number of refactorings applied non-smelly classes.

In general, we found that the total number of refactorings applied to classes containing Android smells is 2,189 out of 9,600 which accounts for 23% of the total number of refactorings. We also found that the *Extract Method*, *Move Attribute*, *Rename Method* and *Move Method* are the most applied refactorings accounting, respectively, for 694, 429, 414 and 365 refactorings. Furthermore, results indicate that some particular Android code smells tend to attract more refactorings than others. For instance, *Member Ignoring Method*, *Inefficient Data Structure* and *No Low Memory Resolver* are the most refactored smells with 659 and 529 refactorings, respectively. On the other hand, we found that the poorly refactored smells are the *Unclosed Closable (85)*, *Leaking Thread* (69), *Durable Wakelock* (58), *Inefficient Data Structure* (43) and *Internal Getter and Setter* (11).

To better analyze the associations between Android smells and refactorings, we assess the support, confidence, lift and conviction scores for each frequent itemset. Table 2.7 shows the frequent itemsets for the Android smells and refactorings for each studied app, where each itemset comprises one Android smell and one refactoring operation. The last two columns report the Chi-squared and Cramer's V tests to check whether the associations between code smells and refactorings are statistically significant or not.

We found that from the 15 studied Android smells, only five Android smells appear in the frequent itemsets:

- *Leaking Inner Class*: has a high chance being involved in *Move Attribute* or *Extract and Move Method* refactorings in the Congress app. However, this smell is rarely refactored in other apps.
- *Inefficient Data Structure*: is often involved in *Extract Method* refactorings in the Congress app, while it is rarely refactored in other apps.

Table 2.6 Refactorings performed on each type of Android-specific code smell

Refactoring	Application	DTWC	DR	DW	IDFP	IDS	ISQLQ	IGS	LIC	LT	MIM	NLMR	PD	RAM	SL	UC	Total Smelly	Total Non-Smelly
Extract Method	Apg	7	0	0	0	1	0	0	9	3	49	19	0	0	26	13	114	553
	congress	1	28	0	0	29	0	0	18	0	15	30	0	0	0	0	121	386
	Netguard	30	0	33	0	0	0	0	30	35	47	40	0	0	0	0	215	635
	Notepad	21	0	0	0	0	0	0	28	0	40	32	0	0	0	0	121	281
	Omni-Notes	0	0	0	0	0	0	0	0	0	61	26	0	0	36	12	123	584
Move Method	Apg	1	0	0	0	2	0	0	4	0	43	4	0	0	8	14	62	248
	congress	0	36	0	0	0	0	0	46	0	23	44	0	0	0	0	149	308
	Netguard	8	0	0	0	0	0	0	8	7	11	9	0	0	0	0	43	279
	Notepad	0	0	0	0	0	0	0	0	0	13	14	0	0	0	0	27	156
	Omni-Notes	8	0	0	0	0	0	4	6	0	32	20	0	0	14	8	84	268
Extract and Move Method	Apg	2	0	0	0	0	0	1	3	2	11	4	0	0	5	3	28	123
	congress	0	13	0	0	1	0	0	11	0	13	13	0	0	0	0	51	111
	Netguard	10	0	9	0	9	0	0	10	9	15	13	0	0	0	0	75	152
	Notepad	1	0	1	0	0	0	0	0	0	2	2	0	0	0	0	6	99
	Omni-Notes	0	0	0	0	0	0	0	2	0	6	4	0	0	4	0	16	260
Inline Method	Apg	3	0	0	0	0	0	0	3	1	3	1	0	0	7	7	18	84
	congress	0	11	0	0	1	0	0	9	0	12	10	0	0	0	0	43	74
	Netguard	3	0	4	0	0	0	0	0	5	6	6	0	0	0	0	24	75
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	30
	Omni-Notes	0	0	0	0	0	0	0	0	0	4	2	0	0	4	0	10	51
Rename Method	Apg	5	0	0	0	0	0	0	13	3	46	18	0	0	27	17	112	324
	congress	0	38	2	0	0	0	0	15	0	17	20	0	0	0	0	92	247
	Netguard	0	0	3	0	0	0	0	0	0	35	44	0	0	0	0	82	286
	Notepad	8	0	6	0	0	0	0	0	0	13	12	0	0	0	0	39	249
	Omni-Notes	12	0	0	0	0	0	6	18	0	30	10	0	0	13	10	89	267
Move Attribute	Apg	0	0	0	0	0	0	0	1	1	7	3	0	0	1	1	13	134
	congress	0	99	0	0	0	0	0	90	3	80	117	0	0	0	0	389	467
	Netguard	6	0	0	0	0	0	0	0	0	10	9	0	0	0	0	25	155
	Notepad	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	2	79
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	42
Pull Up Method	Apg	0	0	0	0	0	0		0	0	10	1	0	0	0	0	11	127
	congress	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Pull Up Attribute	Apg	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	5	74
	congress	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	22
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
Push Down Method	Apg	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	70
	congress	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0
Rename Class	Apg	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	congress	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	18
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
Extract Super Class	Apg	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	70
	congress	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Omni-Notes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Move Class	Apg	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	congress	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Netguard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Notepad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Omni-notes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total		**126**	**225**	**58**	**0**	**43**	**0**	**11**	**324**	**69**	**659**	**529**	**0**	**0**	**145**	**85**	**2,189**	**7,411**

Table 2.7 Top Android-specific code smells and refactoring associations in each application based on confidence and lift values

App	Item pairs	Support	Confidence	Lift	leverage	Conviction	Chi-square P-value	Cramer's V
Congress	**Leaking Inner Class → Move Attribute**	0.11	**0.63**	**2.437**	0.072	2.871	**<0.05**	**0.030**
	Leaking Inner Class → Extract and Move Method	0.09	**0.52**	**1.524**	0.03	1.587	**<0.05**	**0.051**
	Inefficient Data Structure → Extract Method	0.01	**0.98**	**4.48**	0.018	1.033	0.070	0.021
	No Low Memory Resolver → Move Attribute	0.16	**0.53**	**1.124**	0.02	1.56	0.282	0.010
Netguard	Leaking Thread → Extract And Move Method	0.1	**0.58**	**1.503**	0.034	1.75	0.643	0.014
	No Low Memory Resolver → Extract Method	0.07	**0.72**	**2.831**	0.511	0.385	**<0.05**	**0.102**
Apg	Member Ignoring Method → Extract Method	0.24	**0.62**	**1.118**	0.087	0.584	0.221	0.005
Omni-Notes	**Member Ignoring Method → Extract Method**	0.44	**0.67**	**4.475**	0.412	1.688	**<0.05**	**0.03**

- *No Low Memory Resolver*: is often involved in *Move Attribute* refactorings in the Congress app, and *Extract Method* in the Netguard app, but rarely refactored in other apps.
- *Member Ignoring Method*: is often involved in *Extract Method* refactorings in both Apg and Omni-Notes apps, while being rarely involved with any refactoring type in the other studied apps.
- *Leaking Thread*: tends to be frequently involved with the *Extract and Move Method* refactorings in the Congress app, while being rarely refactored in the other apps.

We further assess the statistically significant associations from these Android smells and refactorings within our subject apps, and determine the degree of associations using Cramer's V tests. We can see from Table 2.7 that not all these associations are statistically significant. We see that *No Low Memory Resolver → Extract Method* had the highest Cramer's V, reaching 0.102. However, *Inefficient Data Structure* and *Leaking Thread* do not have statistically significant associations with any particular refactoring types. For the remaining Android smells, we did not observe any particular associations with any refactoring operation.

Android smells have been discussed in previous works involving code snippet examples Hecht *et al.* (2016); Palomba *et al.* (2017a); Habchi, Moha & Rouvoy (2020). To better understand this phenomenon, we present a real-world example from our dataset showing the association between the *Member Ignoring Method* smell and the *Move Method* refactoring, taken from commit #7b24ee7 of the Apg app [12]. In this commit, the applied refactorings in-

[12] https://github.com/thialfihar/apg/commit/7b24ee7b55db99467dd63e631ba55a27d08587d5

volve moving some methods, namely `getKeyIdPassphraseNeeded()`, `getNfcHash()`, `getNfcAlgo()`, `isPending()`, `getNfcTimestamp()` and `getDetachedSignature()` from the class `SignEncryptResult` to the class `PgpSignEncryptResult`. All these methods are affected by the *Member Ignoring Method* smell since they are non-static methods that do not access any non-static attributes. Generally, to remove this smell developers either make the affected method static or introduce code to the affected method that accesses non-static attributes, or simply remove the affected method entirely. In this example, developers opted to move these smelly methods from the smelly class to another class. Note that although this action removes the smell from the original class, the smell still persists in the target class.

The obtained results indicate that developers are unlikely to apply refactorings to Android-specific code smells. This is unsurprising in that the refactoring operations employed in this study were defined to remove OO code smells Fowler *et al.* (1999); Brown *et al.* (1998a). However, it is notable that when refactoring a class, Android developers pay little attention to Android smells. In any case, it may be the case that such new emerging types of smells require specialized refactoring tools to consider the characteristics of the Android platform, as pointed out in prior research Palomba *et al.* (2019); Kessentini & Ouni (2017).

Summary for RQ2. Android developers do not seem to focus their refactoring efforts on code smells. A total of 23% of refactorings were applied to Android smells while 25% of refactoring were applied to traditional OO smells. The most applied refactoring operations include Extract Method, Move Method, Rename Method and Move Attribute, comprising 62% of the total number of refactorings. The association between code smells and refactorings varies depending on the application under consideration.

2.4.3 RQ3. To what extent do refactoring operations applied to smelly classes remove code smells?

To study the extent to which refactoring operations performed on smelly classes are able to remove code smells, we use the same analysis metrics used for RQ2, i.e., support, confidence, lift and conviction as well as the Chi-squared and Cramer's V tests. As this research question is concerned with smells that are fixed after refactoring, we only consider smelly classes where the smell is corrected after the application of a refactoring, instead of considering all smelly classes in was was done for RQ2.

Traditional OO smells: Table 2.8 reports the percentage of smells removed by refactoring for each app. We observe that on average only 5% of OO smells are removed by refactoring in the studied apps. We are interested in exploring further the co-occurrence between refactoring operations and smelly classes that are fixed by refactoring. To this end, Table 2.5 shows the frequent itemsets for the fixed OO smells and refactorings for each studied app, where each itemset comprises one OO smell and one refactoring, as well as the Chi-squared and Cramer's V tests. We found that Extract Method and Move Method are able to fix some code smells such as Complex Class, Long Method, Feature Envy, Message chain and Spaghetti Code. However, we can see from the table that not all these associations are statistically significant. The top three associations are (1) Long Method → Extract Method, (2) Message Chain → Extract Method and (3) Complex Class → Extract Method. On the other hand, some associations do not seem to be statistically such as Feature Envy → Move Method, Long Method → Move Method and Message chain → Move Method.

To gain some insight into the reasons why refactoring rarely removes the OO smells in our study, we refer to an example that shows the impact of the Extract Method refactoring on the Complex Class code smell. This example is taken from commit #f8c9248 of the Omni-Notes app Impact of Extract Method refactoring (2021), and involves the extraction of the methods `initViewFooter()`, `initViewReminder()`, `initViewLocation()`, `initViewAttachments()` and `initViewTitle()` from the class `DetailFragment`. In spite of the application of these Extract Method refactorings, we found that the `DetailFragment` class remains complex, even though it is generally accepted that the Extract Method refactoring should remove this smell Fowler *et al.* (1999); Bavota *et al.* (2015a); Saika *et al.* (2016); Yoshida *et al.* (2016). However, this class contains 2,364 lines of code, involving several overly complex/long methods and contains many method invocations. Refactoring such a class with

Table 2.8 Percentage of removed Android and traditional OO code smells

App	OO smells (%)	Android smells (%)
Apg	7	1
NetGuard	4.7	3
Omni-notes	5	2
Congress	4	0.5
Notepad	3	1
Average	**4.74**	**1.5**

Table 2.9 Top removed traditional OO code smells and refactoring pairs based on confidence and lift values

App	Item pairs	Support	Confidence	Lift	Leverage	Conviction	Chi-square P-value	Cramer's V
Apg	**ComplexClass → Extract Method**	0.11	**0.54**	**4.7085**	0.0189	1.3569	**<0.05**	**0.021**
	Feature Envy → Move Method	0.12	**0.5**	**3.5569**	0.0280	1.5208	0.710	0.004
	Long Method → Extract Method	0.05	**0.63**	**6.2597**	0.0454	2.7301	**<0.05**	**0.024**
	Message chain → Move Method	0.14	**0.69**	**5.3887**	0.0151	1.5293	0.160	0.003
Congress	Complex Class → Extract Method	0.39	**0.65**	**1.7292**	0.0071	4.8261	0.450	0.005
	Long Method → Extract Method	0.11	**0.66**	**3.6162**	0.0262	1.0613	**<0.05**	**0.032**
NetGuard	**Feature Envy → Extract Method**	0.01	**0.64**	**2.6881**	0.0365	1.2060	**<0.05**	**0.015**
	Spaghetti Code → Extract Method	0.02	**0.56**	**1.2804**	0.0146	1.1522	**<0.05**	**0.110**
	Long Method → Extract Method	0.01	**0.77**	**3.6162**	0.0262	1.0613	0.070	0.070
	Message chain → Move Method	0.11	**0.61**	**5.0833**	0.0189	1.4731	**<0.05**	**0.039**
Omni-Notes	Long Method → Move Method	0.44	**0.59**	**1.1116**	0.0821	4.3337	0.661	0.011
Notepad	**Message chain → Extract Method**	0.14	**0.69**	**5.0833**	0.0189	1.4731	**<0.05**	**0.067**

only Extract Method will not resolve the problem. Rather, this class needs to be reworked intensively by the application of several combined refactorings in order to remove this smell.

Android smells: From Table 2.8 we observe that on average, only 1.5% of Android smells are removed through refactoring operations. Looking at the co-occurrences between refactoring operations and smelly classes that are fixed when they exhibit a refactoring operation, results show that refactorings applied fix only the Member Ignoring Method smell. However, we observe from Table 2.8 that only three associations are statistically significant: Member Ignoring Method and Extract Method, Member Ignoring Method and Push Down Method, and Member Ignoring Method and Move Method.

To further investigate why refactoring rarely removes the Android smells in our study, we refer to an example that shows the associations between Leaking Inner Class (LIC) and the Extract

Table 2.10 Top removed Android-specific code smells and refactoring pairs based on confidence and lift values

App	Item pairs	Support	Confidence	Lift	Leverage	Conviction	Chi-square P-value	Cramer's V
Apg	**Member Ignoring Method → Extract Method**	0.24	**0.51**	**4.605**	0.012	1.2842	**<0.05**	**0.019**
	Member Ignoring Method → Move Method	0.47	**0.67**	**1.333**	0.010	1.0974	**<0.05**	**0.034**
	Member Ignoring Method → Pull Up Method	0.06	**0.6**	**1.286**	0.010	1.0110	0.760	0.001
	Member Ignoring Method → Push Down Method	0.08	**0.67**	**1.059**	0.010	1.0272	**<0.05**	**0.028**
	Member Ignoring Method → Extract And Move Method	0.27	**0.67**	**2.7241**	0.0098	1.0472	0.404	0.003
Notepad	Member Ignoring Method → Move Method	0.47	**0.67**	**1.0658**	0.0045	1.6645	0.430	0.002
	Member Ignoring Method → Pull Up Attribute	0.06	**0.6**	**0.339**	0.003	1.0087	1	0.001

And Move Method refactorings. We select this pair for consideration since in the results of RQ2 shown in Table 2.7, we see that developers often apply this refactoring when a class is affected by the Leaking Inner Class smell. However, the result of RQ3 (Table 2.10) shows that refactoring fails to remove this smell. The LIC smell occurs when an anonymous inner class holds a reference to its containing class instance. There exists two possibilities to remove this smell: either by making the affected inner class static, or by removing it entirely. Thus, the applied the Extract And Move Method refactorings cannot remove this smell. Further investigation is needed to understand the intention of the developers behind the applications of certain refactorings, but it seems clear in this case that the presence of the LIC smell is not what prompted the developers to refactor the class.

Summary for RQ3. The effectiveness of refactoring in removing code smells is generally low in all analyzed apps, with only 5% of traditional code smells and 1.5% of Android-specific code smells being removed.

2.4.4 Discussion

In this section, we discuss the implications that our results have on the prevalence of code smells, the impact code smells have on refactoring activity and the extent to which code smells are removed by refactoring.

2.4.4.1 Implications for smell prevalence (RQ1)

The result of RQ1 shows that both Android and traditional OO smells are remarkably prevalent in Android applications. This widespread prevalence suggests either developer unawareness of these code smells, or that developers simply do not regard these code smells to be important Peters & Zaidman (2012); Silva *et al.* (2016); Habchi *et al.* (2020). Not all code smells are equally prevalent in the studied apps, nor should they be regarded as being equally serious.

For example, *Member Ignoring Method* and *Message Chain* are both prevalent smells in the studied applications. However *Member Ignoring Method* has only a minor impact on performance and is easily repaired by making the offending method static. On the other hand, *Message Chain* is a smell that is likely to represent a violation of the Law of Demeter Lieberherr & Holland (1989) which causes insidious coupling between classes. Developers need to focus their attention on smells like this that may lead to unforeseen and long-lasting quality issues in the software.

2.4.4.2 Implications for smell refactoring (RQ2)

The results for RQ2 show that a total of 23% of refactorings were applied to Android smells while 25% of refactoring were applied to traditional OO smells. It is striking here that most refactorings are applied to non-smelly code, which is additionally surprising given the widespread prevalence of code smells. This indicates that even when refactoring code, developers seem not focus on smelly code. This resonates with the results for RQ1 — that developers seem not to be too concerned by smells in their code.

We also observe that refactorings have a higher number of associations with OO smells than they do with Android code smells. This is to be expected as the employed refactoring detection tool, RefactoringMiner, was developed to identify refactorings associated with OO smells. Also OO smells are very well established and appear in many programming text books, whereas Android code smells have only recently been cataloged and they are not yet as generally accepted as their OO counterparts.

2.4.4.3 Implications for smell removal (RQ3)

The result of RQ3 reveals that the effectiveness of refactoring in removing code smells is generally low in all analyzed apps, with only 5% of traditional OO code smells and 1.5% of Android-specific code smells being removed. This result emphasises more powerfully what we have observed for the other RQs: that although smells are prevalent and refactoring is a prevalent practice, very few smells are actually removed by refactoring.

Assuming that developers are generally aware of code smells and could remove them by refactoring if desired, it is apparent that they are not bothered by the code smells evident in the code. One possible explanation is that refactoring activity is mainly driven by changes in the requirements rather than by the necessity to fix code smells, i.e. developers refactor to make the code more amenable to the new functionality they are implementing rather than with the aim of removing code smells. In order to investigate this further, a follow up study is necessary to uncover the reasons for which developers do not invest refactoring efforts in removing smells, similar to related studies carried out on desktop applications Bavota *et al.* (2015a).

2.5 Threats to validity

This section discusses potential threats to the validity of our study.

1. *Threats to internal validity* concern factors that could influence our observations. In particular, when code smells disappear in a given refactoring commit, this may or may not be related to the refactoring itself that occurred in that commit. However, while our investigation allowed us to claim correlation and not causation, we manually checked a sample of commits in which specific types of refactorings helped to remove some code smells. To further investigate this phenomenon, we plan to conduct a qualitative analysis with developers to better understand their intuitions AlOmar, Mkaouer & Ouni (2020a). Moreover, although we have double checked our experiments and the datasets collected, there could still be errors that we did not notice.

2. *Threats to construct validity* concerns imprecisions/errors in the various measurements we performed in the course of our experiments. We consider these further in the paragraphs below.

   Detecting code smell instances is a vital part of our work and to perform this we relied on two tools. For Android smells, we used the aDoctor tool Palomba *et al.* (2017a), while for traditional code smells, we used a rule-based, state of the art tool, Organic, developed by Bavota et al. Bavota *et al.* (2015a). We are well aware that the validity of our results could be affected by the presence of false positives and false negatives in the results of these tools. The aDoctor tool has been shown to have a precision and recall of 98% Palomba *et al.* (2017a, 2019), while the Organic tool has been shown to have a precision and recall of 72% and 81% respectively Bavota *et al.* (2015a). These results form a promising basis for our investigations, however we cannot exclude the possibility that some code smells were missed by our analysis or that false positives occurred.

   Similarly, our analysis is threatened by the accuracy of the employed refactoring detection tool, RefactoringMiner Tsantalis, Mansouri, Eshkevari, Mazinanian & Dig (2018a). However, previous studies reported that RefactoringMiner has high precision and recall scores compared to other state-of-the-art refactoring detection tools Silva *et al.* (2016); Tsantalis *et al.* (2018a), which gives us confidence in using the tool. It is also worth noting that some smells we consider in our study, especially the Android ones, simply cannot be fixed by refactorings identified by RefactoringMiner. However, our analysis allows us to detect that a smell has been removed in a given commit (i.e., using the smell detection tool, aDoctor, we can identify if a given class still contain a smell or not), but just we wouldn't attribute its removal to a refactoring. This allows to mitigate this concern and makes us more confident about our analysis

   Another threat to validity is related to the metric we employ for code smell prevalence (i.e., the ratio of the total number of classes affected by a given smell to the total number of classes). This measure was used in recent empirical studies in code smells Palomba *et al.* (2019); Grano, Palomba, Di Nucci, De Lucia & Gall (2019), which gives us confidence that it is a suitable metric, however it may be beneficial to measure prevalence in another way.

For example, one smell may occur multiple times in a single class and this ratio would then under-represent its prevalence. It is thus interesting to consider more fine-grained measures such as the density, i.e., the ratio of smells per method or per KLOC.

3. *Threats to conclusion validity* concerns the relationship between the treatment and the outcome, which in this context refers to the analysis methods employed in our study. Unlike other papers on code smell and refactoring co-occurrence, we exploited association rule mining Agrawal *et al.* (1994), rather than using a logistic regression model. To avoid unreliable results and to assess the significance of our findings, we performed Cramer's V test to determine the strengths of association and Pearson's chi-square coefficient to test for significance. Another aspect is related to the details of the measurements conducted, which might have influenced our observations. We worked at the class level as our goal was to assess the relationship between code smells and refactorings. However, this does not exclude the possibility that other code smells at method-level might still have an impact. Furthermore, while we found 75% of refactorings were applied to classes that do not contain traditional code smells, this might depend upon the number of traditional smells that are detected by the employed tool. Hence, considering other available tools to detect a wider range of traditional smells would increase the confidence in our findings.
4. *Threats to external validity* concern the generalization of our findings. Our analysis was based on five open-source Java Android apps of varying size and from different application domains. Since we opted for a detailed analysis for each app, we preferred to observe fewer apps over a longer period of evolution history, rather than many projects for short periods. Although this is a relatively small number of applications, they represent variety in terms of size and application domain, and in total we examined 652 separate releases compromising 9,600 refactoring operations. This is a large enough sample to have some confidence that similar results could be expected from other, open-source, Java Android applications. Of course further analysis of other open source and commercial apps from both Android and iOS marketplaces would further test the generalizability of our conclusions.

5. *Threats to reliability validity* To minimize potential threats to reliability, we analyzed open-source projects available on GitHub and provide a replication package that contains our complete dataset dataset (2020).

2.6 Conclusion and future work

In this paper we presented the results of a study of the impact of refactoring activities in five Android applications. We started with a preliminary study of the prevalence and co-occurrences of a number of Android-specific and traditional code smells, and then investigated the impact of different refactoring operations on these code smells.

Our study delivers several important findings. Firstly, results indicate that code smells are very prevalent in Android apps, with 68% of classes being affected by Android specific smells and 63% of classes affected by traditional OO smells. Secondly, developers are more likely to apply refactoring operations to *non-smelly* code elements. A total of 25% of refactorings were applied to traditional OO smells, while 23% of refactorings were applied to Android smells. Thirdly, these refactoring activities removed only 5% of traditional OO smells and 1.5% of Android smells.

In summary, we observe that code smells are widespread across Android applications, but smelly classes are not particularly targeted by refactoring activities and, when they are, it is rare for refactoring to actually remove a smell. These results are remarkable, and run contrary to the "traditional" wisdom that refactoring is used to remove code smells.

Two extreme interpretations of these results are: (1) Android developers are poorly versed in code smells and refactoring. They introduce them liberally while developing and are unable to direct their refactoring activities to remove them subsequently and (2) Android developers do not in general regard code smells as signs of any real quality problem. The introduce them while developing and do not care to remove them, preferring to focus their refactoring activities on real quality issues.

While the true interpretation lies between these two extremes, we expect that (2) is closer to the truth. So while it may be of value to promote awareness of both Android and traditional code smells, and how to refactor them, among Android developers, it is more critical to understand better what are the real quality issues Android developers face and to develop a model of code smells and refactoring that better addresses their needs.

We foresee several possible directions for future work. Firstly, the use of non-smell models of software quality, in particular software quality metrics, to determine if they can better explain the refactoring activities that take place during Android app development. Secondly, we consider extending our study to (1) more open source and commercial Android apps to better generalize our results and to develop Android specific refactoring tools to better support developers during maintenance and evolution and (2) to more refactoring tools, different code smells, and refactorings. Thirdly, we may evaluate the impact of code smells on some non-functional aspects such as energy smells Palomba *et al.* (2019); Carette *et al.* (2017). Finally, we plan to conduct a qualitative investigation through a survey with Android developers to better understand their intuition behind refactoring activities in the context of mobile apps.

CHAPTER 3

AN EMPIRICAL STUDY ON CODE SMELLS CO-OCCURRENCES IN ANDROID APPLICATIONS

Oumayma Hamdi[a] , Ali Ouni[a] , Eman Abdullah AlOmar[b] , Mohamed Wiem Mkaouer[b]

[a] Department of Software and IT Engineering, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

[b] Department of Software Engineering, Rochester Institute of Technology, Rochester,
NY, USA,

Paper published in the *International Workshop of Refactoring (IWoR)* , October 2021

3.1 Abstract

Android applications (apps) evolve quickly to meet users requirements, fix bugs or adapt to technological changes. Such changes can lead to the presence of code smells – symptoms of poor design and/or implementation choices that may hinder the project maintenance and evolution. Most of previous research focused on studying the characteristics of traditional object-oriented (OO) code smells affecting source code files in desktop software systems, and advocated that the interaction and co-presence of code smells reduce the ability of developers to understand and maintain source code. However, little knowledge is available on emerging categories of Android-specific code smells and their interactions, i.e., co-occurrences, with traditional OO smells, in the context of Android apps. To provide a broader understanding of this phenomenon, we conduct an empirical study on 1,923 open source Android apps taking into account 15 types of Android-specific and 10 types of traditional OO code smells to explore (*i*) the extent to which code smells co-occur together, and (*ii*) which code smells tend to co-occur together. Our results show that (*i*) the co-occurrence phenomenon is indeed prevalent in Android apps, where 51% of classes are affected by more than one smell instance (from either OO and Android smells), while 34% of classes are affected by more than one Android smell, and 26% are affected by more than one OO smells, and (*ii*) there exist 14 smell pairs that have strong associations. Developers need

to be aware of this phenomenon and consider detecting and eliminating both traditional and Android smells, using dedicated tools.

3.2 Introduction

Android apps have to evolve quickly to meet the continuous user needs, technological changes and stay ahead of the mobile apps store competition. However, throughout their evolution, Android apps undergo changes that often lead to poor implementation and design practices that are manifested in the form of *code smells* Brown *et al.* (1998a); Fowler (2018); Reimann *et al.* (2014). The presence of code smells often hinders the maintenance and evolution of any software system Habchi *et al.* (2019b); Cedrim *et al.* (2017); Bavota *et al.* (2015a); Palomba *et al.* (2017b); Ouni *et al.* (2013). Like any software system, Android apps can be affected by traditional Object-Oriented (OO) code smells Brown *et al.* (1998a); Fowler (2018), but also with new categories of emerging Android-specific smells, known as *Android smells* Reimann *et al.* (2014); Kessentini & Ouni (2017); Palomba *et al.* (2017a, 2019); Hecht *et al.* (2016). The presence of these smells can lead to resource leaks (e.g., CPU, memory, battery, etc.) causing, therefore, several performance and usability problems Palomba *et al.* (2019); Habchi, Moha & Rouvoy (2019a); Hecht *et al.* (2016); Morales *et al.* (2017).

Most of the existing studies focused on traditional OO smells Palomba *et al.* (2017b, 2018b); Fontana *et al.* (2015); Garg *et al.* (2016); Yamashita *et al.* (2015); Muse *et al.* (2020); Martins, Bezerra, Uchôa & Garcia (2020); Abbes *et al.* (2011). In particular, they focused on various aspects of OO code smells including code smells prevalence Palomba *et al.* (2019); Delchev & Harun (2015); Peruma *et al.* (2019); Mannan *et al.* (2016), co-occurrences Palomba *et al.* (2018b); Garg *et al.* (2016); Fontana *et al.* (2015); Abbes *et al.* (2011); Yamashita *et al.* (2015); Yamashita & Moonen (2013b) as well as the effects of code smells on software quality and maintainability D'Ambros, Bacchelli & Lanza (2010); Abbes *et al.* (2011); Khomh *et al.* (2012). It has been also demonstrated that the co-existence and the interactions between OO code smells consistently reduce the ability of developers to understand source code, and thus, it complicates maintenance tasks Palomba *et al.* (2017b, 2018b); Fontana *et al.* (2015); Garg *et al.*

(2016); Yamashita *et al.* (2015); Muse *et al.* (2020); Martins *et al.* (2020); Abbes *et al.* (2011). Furthermore, few studies have recently examined individual instances of code smells in Android apps Habchi *et al.* (2019b); Palomba *et al.* (2019); Hecht *et al.* (2016); Morales *et al.* (2017); Mannan *et al.* (2016).

Although several important research steps have been made and despite the ever-increasing number of empirical studies aimed at understanding traditional OO code smells, little knowledge is available about the phenomenon of code smell co-occurrences in Android apps. While knowledge about such individual smell types is established in recent years Habchi *et al.* (2019b); Palomba *et al.* (2019); Reimann *et al.* (2014); Palomba *et al.* (2017a), important relationships are missing between traditional OO smells and Android smells.

This knowledge is particularly important for developers researchers and tool creators. For Android developers, discovering such relationships will help them to save time and effort by focusing their attention by getting a high priority in refactoring the smells that frequently co-occur together which may lead to better monitoring the quality of their apps. As for researchers, it can be a starting point for a deep investigation of the relation between Android smells and traditional code smells. Also, such knowledge can help researchers designing Android-specific refactoring techniques and prototypes that take into consideration the hidden dependencies between such smells. For tool creators, such knowledge can be helpful to develop practical and reliable refactoring tools for mobile apps based on the detection of the occurrence of Android code smells given some traditional code smells or vice versa.

This paper aims at improving the current knowledge about code smells in Android apps. We conduct an empirical study on the prevalence of code smell co-occurrences and determine which code smell types tend to co-exist more frequently. Our study is conducted on a large dataset composed of 1,923 open-source Android apps that are freely distributed in Google Play Store. We considered 10 common types of OO code smells, and 15 common Android smells having different characteristics and different granularity levels. To discover such relationships between smells, we use association rule learning based on the Apriori algorithm Agrawal *et al.* (1993) which is commonly used to find patterns in data.

Overall, our investigation delivers several actionable findings indicating that:

- The phenomenon of code smells co-occurrences is quite prevalent in Android apps. Particularly, 51% of classes are affected by more than one smell instance (from either OO or Android smells), while 34% of classes are affected by more than one Android smell, and 26% are affected by more than one OO smells.
- There exist 14 smell pairs that frequently co-occur together: three pairs for Android smells (e.g., *Leaking Inner Class* and *Member Ignoring Method*), seven pairs for OO code smells (e.g., *Long Method* and *Long Parameter List*) and four pair combining both Android and OO (e.g., *Complex Class* and *Member Ignoring Method*). For OO smells, our results are inline with prior findings in the literature on code smells co-occurrences in desktop applications Palomba *et al.* (2017b, 2018b, 2019). However, for Android smells, our findings reveal various interesting relationships in the context of Android apps development.

3.3 Study Design

The *goal* of this study is to investigate various types code smell co-occurrences in the context of Android apps for the purpose of assessing the prevalence of this phenomenon and determining the pairs of smells that tend to co-occur together frequently.

3.3.1 Goals and Research Questions

Our study aims at addressing the following research questions.

- **RQ1**. **To what extent code smells co-occur in Android apps?**
  This research question aims at assessing the extent to which Android apps contain classes affected by one or more code smell types. By answering *RQ1*, we can reveal the prevalence of this phenomenon.
- **RQ2 Which code smells co-occur together?**

With this research question, we aim at identifying which code smells tend to co-occur together, and thus reporting on the existence of different patterns of code smell co-occurrences that can exist in Android apps.

3.3.2 Context and Dataset

The context of our study consists of a set of 1,923 open source Android apps, and two categories of code smell types that can exist in Android apps (1) traditional OO smells and (2) Android specific smells. In particular, we analyzed 15 common Android smells extracted from the catalog defined by Reimann et al. Reimann *et al.* (2014). This catalog reports a set of poor design/implementation choices applied by Android developers that can impact non-functional attributes of Android apps, and have been used by prior studies on Android smells Palomba *et al.* (2019); Habchi *et al.* (2019b); Carette *et al.* (2017); Kessentini & Ouni (2017). We also considered 10 common traditional OO smells defined by Fowler Fowler (2018) and Brown et al. Brown *et al.* (1998a) that have been widely studied in prior works Bavota *et al.* (2015a); Palomba *et al.* (2017b, 2018b). These smells have (1) different granularity, e.g., class, method, statement, etc., and (2) varying characteristics, e.g., classes characterized by long/complex code as well as violation of accepted OO design and implementation principles. Tables 1.2 and 1.3 report the set of OO and Android smells, respectively, that are investigated in our study (See section 1.1.3).

3.3.3 Data Extraction

Figure 4.1 describes the overall process used to collect our dataset. We targeted real world apps that have been designed and developed as open source projects and that are freely distributed on Google Play Store and hosted on GitHub.

First, we performed a custom search on GitHub by targeting all Java repositories in which the `readme.md` file contains a link to a Google Play Store page (Step A). In total, we obtained 19,212 apps. Thereafter, we filtered our dataset with the following criteria inspired by Das, Di Penta & Malavolta (2016); Malavolta, Verdecchia, Filipovic, Bruntink & Lago (2018):

- We consider only the repositories that contain the `AndroidManifest.xml` file, as the apps whose GitHub repository does not contain an Android manifest file clearly do not refer to real Android apps. The result of this filter was a collection of 5,766 apps.
- We excluded all unpublished apps, i.e., those apps for which the corresponding Google Play page is not existing anymore (i.e., removed from the store). Our filter returned 3,160 apps.
- We excluded repositories that contain forks of other repositories. This filtering step leads to a final set of 1,923 Android apps.

Our final dataset resulting from the filtering process contains 1,923 real Android apps, each of them is represented by its GitHub and Google Play identifiers. Then, we download the source code of the last release from each app using `git clone` command. The latter will serve for the next step: collecting the code smells.

Thereafter (Step B), for each app we identify the presence of any instance of OO and Android smell at the class level. As for Android-specific smells, we used aDoctor [13], a command-line based tool that implements rules provided by Palomba et al. Palomba *et al.* (2017a) to identify common Android smells. We selected this tool as it achieves a high detection precision of 98%, and recall of 98%, as reported in Palomba et al. Palomba *et al.* (2017a). As for the traditional OO smells, we used an existing tool [14], that has been widely used in recent studies Palomba *et al.* (2018b,a, 2017b); Bavota *et al.* (2015a). The tool detects 10 common types of OO smells and implements simple detection rules published by Bavota et al. Bavota *et al.* (2015a) to ensure a high recall and precision. The detection process resulted in identifying 29,550 instances of traditional OO smells, and 23,267 instances of Android smells. Table 4.1 summarizes the statistics about the collected dataset.

[13] https://github.com/fpalomba/aDoctor

[14] https://github.com/opus-research/organic

Table 3.1 Dataset statistics

Statistic	Count
Number of Android apps	1,923
Total number of classes	19,212
Total number of methods	134,400
Number of traditonal OO smell instances	29,550
Number of Android smell instances	23,267
Total number of all smell instances	52,817

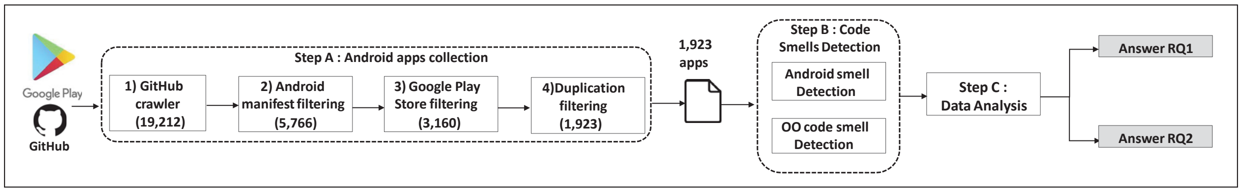
Google Play
GitHub
Step A : Android apps collection
1) GitHub crawler (19,212)
2) Android manifest filtering (5,766)
3) Google Play Store filtering (3,160)
4)Duplication filtering (1,923)
1,923 apps
Step B : Code Smells Detection
Android smell Detection
OO code smell Detection
Step C : Data Analysis
Answer RQ1
Answer RQ2

Figure 3.1 Overall process to conduct our empirical study

3.3.4 Data Analysis

After collecting all necessary data for our study, we use specific analysis methods to answer each RQ (Step C).

3.3.4.1 Analysis method for RQ1

To answer *RQ1*, we compute the number of smells affecting each class in the dataset. Then, we report the percentage of classes affected by one or multiple types of code smells.

3.3.4.2 Analysis method for RQ2

To answer *RQ2*, we employ association rule mining (also known as market basket analysis) using the *Apriori* algorithm Agrawal *et al.* (1993). The algorithm parses the dataset, i.e., transactions, and generates frequent itemsets based on filtering criteria set. Association rules are generated during searching for frequent itemsets. An association rule is defined as an implication of the form X ⇒ Y, where X, Y ⊆ I and X ∩ Y = ∅ . Let $I = \{i_1, i_2, ..., i_n\}$ be a set of n items, and

$T = \{t_1, t_2, ..., t_m\}$ a set of m the transactions. In our study, T is the set of classes present in version, and each item in the set I indicates the presence of two specific smell types. Therefore, an association rule translates a co-occurrence between a smell S_i and other smell S_j on the same class. Specifically, the association rule is written as follows: $Smell(S_i) \Rightarrow Smell(S_j)$.

We use the support Agrawal *et al.* (1993), confidence Agrawal *et al.* (1993) and lift Brin, Motwani, Ullman & Tsur (1997) scores to quantify the degree of association between each pair of smells.

1. *Support*: is an indication of how frequently an itemset appears in the dataset and consists of the proportion of transactions in the dataset that contain both S_i and S_j.

$$Support(S_i \Rightarrow S_j) = P(S_i \cup S_j) \tag{3.1}$$

2. *Confidence*: is the proportion of transactions in the dataset containing S_i, that also contain S_j.

$$Confidence(S_i \Rightarrow S_j) = P(S_i \cup S_j)/P(S_i) \tag{3.2}$$

3. *Lift*: is the ratio of the observed support to that expected if S_i and S_j were independent.

$$Lift(S_i \Rightarrow S_j) = P(S_i \cup S_j)/(P(S_i) \times P(B)) \tag{3.3}$$

The range of values for support and confidence is between 0 and 1, whereas lift can take any value between 0 and $+\infty$. When the lift value is greater than 1, it implies that the smell pair is highly correlated.

Moreover, we use the Pearson's Chi-square coefficient and Cramer's V Cramér (1999) tests to determine if there were significant associations between the smells. Specifically, for any Chi-square test that was found to be significant (p-value < 0.001), Cramer's V test is calculated and it has a value between 0 and 1. A value of 0 indicates complete independence, and a value of 1 indicates complete association. The formula is given in Equation 3.4:

$$V = \sqrt{\frac{\chi^2}{n \times min(row - 1, col - 1)}} \tag{3.4}$$

3.3.5 Replication package

Our dataset is available in our replication package for future replications and extensions Dataset (2021a).

3.4 Empirical Study results

3.4.1 RQ1: To what extent code smells co-occur in Android apps?

Table 3.2 reports the obtained results for RQ1, for each of the Android and OO code smells individually and also when both categories are combined together (OO + Android smells). Overall, we observe that the phenomenon of smells co-occurrences is prevalent. For the Android smells, 30% of classes are affected by a single Android smell instance, while 34% of classes are affected by two or more Android smell instances (i.e., the sum of rows from Two smells to Twelve smells). On the other side, for the traditional OO smells, we observe that almost 30% of classes are affected by one OO smell instance, while 26% of classes are affected by two or more OO smells. It is worth noting that we did not find any class affected by more than nine Android or OO code smell types at the same time, as shown in Table 3.2.

When combining both Android and OO smells, we observe from Table 3.2 that 30% of classes are affected by only one smell, while a majority of 52% of classes are affected by two or more smells. Specifically, 18% of classes are affected by exactly two smells, while co-occurrences of three and four smells was observed in 12% and 7% of classes, respectively. Interestingly, we also found that the percentage of classes affected by five or more Android and OO smells is less than 5%.

To better understand the phenomenon of smells co-occurrence, we refer to an illustrative example from the Nextcloud[15] app, version dev-20201223[16]. In particular, the `FileContentProvider` class contains 1,902 line of code and 27 methods. This class is detected at the same time as *Blob Class* and a *Complex Class* code smell as it contains several complex methods. For instance, the method `onUpgrade()` is a complex method having an extremely high cyclomatic complexity of 136 [17] making it difficult to comprehend, maintain, test and evolve. Furthermore, this method is detected at the same time as a *Long Method* and a *Message chain* code smell as it contains 966 lines of code and make 21 calls to other methods.In addition to these traditional OO smells, this method holds also an Android smell, namely the *Slow Loop* as it uses the standard version of the `for` loop which is slow instead of using the `for-each` loop and this that may affect the efficiency of the app Palomba *et al.* (2017a). From this example, one clearly see how some code artefacts can be impacted by several types of code smells. The presence of such smell co-occurences severely impact the understandability, maintainability and extensibility of any software application Habchi *et al.* (2019a); Palomba *et al.* (2018a); Fowler (2018); Hecht, Benomar, Rouvoy, Moha & Duchien (2015); Moha, Gueheneuc, Duchien & Le Meur (2009).

Overall, the obtained results for RQ1 show that the co-occurrence of code smells is indeed prevalent in Android applications. Such prevalence advocates that there might be a lack of awareness about this phenomenon from developers. We thus assess which specific code smells tend to frequently co-occur together.

[15] https://github.com/nextcloud/android

[16] https://github.com/nextcloud/android/releases/tag/dev-20201223

[17] https://scitools.com/

Table 3.2 Prevalence of the code smells co-occurrences in the studied apps

	OO smells			**Android smells**			**OO ∨ Android smells**		
Category	Classes affected	Percentage		Classes affected	Percentage		Classes affected	Percentage	
One smell	5,772	30%	0.6	5,825	30%	0.6	5,696	30%	0.6
Two smells	1,644	9%	0.18	3,663	19%	0.38	3,526	18%	0.36
Three smells	1,477	8%	0.16	1,893	10%	0.2	2,278	12%	0.24
Four smells	788	4%	0.08	787	4%	0.08	1,429	7%	0.14
Five smells	629	3%	0.06	202	1%	0.02	1,049	5%	0.1
Six smells	290	2%	0.04	37	<1%	0.01	716	4%	0.08
Seven Smells	51	<1%	0.01	7	<1%	0.01	525	3%	0.06
Eight smells	2	<1%	0.01	1	<1%	0.01	334	2%	0.04
Nine smells	0	0%		0	0%		209	1%	0.02
Ten smells	0	0%		0	0%		112	1%	0.02
Eleven smells	0	0%		0	0%		43	<1%	0.01
Twelve smells	0	0%		0	0%		18	<1%	0.01

Summary for RQ1. The phenomenon of code smell co-occurrences is quite prevalent. In a dataset containing 52,817 instances of Android and OO code smells, we observe that 34%, 26% and 51% of classes are affected respectively by more than one Android, OO and both (i.e., OO and Android) smell types. These results advocate for the need of awareness mechanisms to support Android developers discovering and removing code smells from their apps.

3.4.2 RQ2: Which code smells co-occur together?

We address RQ2 by identifying the most frequent co-occurrences of code smells in the studied apps. The procedure that we used to identify the code smells co-occurrences are described in Section 3.3.4. To generate frequent itemsets, we selected a minimum confidence of 0.5. We also restrict the maximum number of items in every itemset to 2 since we were interested in the association between one pair of smells.

Table 3.4 presents the frequent itemsets where each itemset comprises two smell types. We also conduct Chi-squared and Cramer's V tests to check whether the associations between code smells are statistically significant or not. It is worth noting that we found some reciprocal associations which are due to variation in the confidence value. To better explain this aspect, we illustrate in Table 3.3 a simplified example of code smells co-occurrences at the class level where transactions are the 6 classes and the items are the S_i and S_j smells. We observe that $\mathit{confidence}(S_i \Rightarrow S_j) = 0.5$, while $\mathit{confidence}(S_j \Rightarrow S_i) = 1$. Thus, the two smells frequently co-occur together in both ways. As for our analysis. overall, we found that there are 14 pairs of code smells that frequently co-occur together and 9 types of code smells that tend to compose such co-occurrences.

Table 3.3 A simplified example of code smells co-occurrences

Class	**Smell** (S_i)	**Smell** (S_j)
Class 1	×	
Class 2	×	
Class 3	×	
Class 4	×	×
Class 5	×	×
Class 6	×	×

The *Complex Class* code smell often co-occurs with other code smell types, and in particular with *Message Chain*, *Feature Envy* and *Member Ignoring Method*. This result is likely to be expected for *Message Chain* and *Feature Envy* smells since complex classes are typically composed of several complex and/or long methods that could be responsible for provoking the long chain of method calls resulting in a *Message Chain* code smell and including dependencies toward other classes since they are composed of several code statements resulting in a *Feature Envy* code smell. However the strong association with the *Member Ignoring Method* was ambiguous and thus, we perform some manual analysis to understand reasons. we found that complex classes contain empty methods (i.e., without instructions) created for prototyping purposes and since they are empty they do not access any non-static attributes or methods. Moreover, as shown in Table 3.4, all these associations are statistically significant.

For the co-occurrence between *Complex Class* and *Message chain*, a clear example was found in WiFi Analyzer application, in version V3.0.3-F-DROID[18]. The class `TitleLineGraphSeries` is affected by the *Complex Class* smell, and indeed its McCabe's cyclomatic complexity reaches 45. At the same time, the method `draw()` is affected by the *Message Chain* and *Feature Envy* smells as it recursively invoke 14 different methods such us `hasNext()` and `isNaN()`

[18] https://github.com/VREMSoftwareDevelopment/WiFiAnalyzer/releases/tag/V3.0.3-F-DROID

and make extensive use of the `width` and `height` attributes belonging to the `View` class to perform some computation in order to draw the background. Hence, this method implementation resulted into a poor cohesion with a lack of cohesion that reaches 87%. As for the co-occurrence between *Complex Class* and *Member Ignoring Method*, we found that the class contains some empty methods created for prototyping purposes such as `getTitle()` and `drawPoint()`. and since they are empty they do not access any attributes or methods.

The *Feature Envy* code smell often co-exists with two code smell types namely *Long Method* and *Long Parameter List*. It is worth noting that this association is reciprocal as shown in Table 3.4. This association is an intended outcome since *Long Methods* are composed of several code statements, accessing of course the data of other classes, they are more prone to also be affected by the *Feature Envy* code smell. Furthermore, the association with *Feature Envy* and *Long Method* smells frequently co-occur with a *Long Parameter List*. This could be an expected consequence since long methods implement several class responsibilities, and thus they require a higher number of parameters, increasing the chances of also being affected by a *Long Parameter List* smell. Furthermore, as shown in Table 3.4, the *Feature Envy:Long Method* smell pair has the highest degree of association with a Cramer's V test value of 0.6

The co-occurrences between the *Message Chain* and the *Member Ignoring Method* code smells are less obvious and not expected. By conducting a qualitative investigation on various samples of some co-occurences, we simply found that the *Complex Class* often co-occurs with both *Message Chain* and *Member Ignoring Method*. Thus the *Message Chain* has a higher chance to be associated with *Member Ignoring Method*.

The *Member Ignoring Method* co-occur with other smell types such as *No Low Memory Resolver*, *Slow Loop* and *Leaking Inner Class*. These smells are associated since they are related to the app's performance and energy consumption, i.e., the CPU time of a method or the memory usage of one variable. This is a reason why the correction of such smells can contribute to improve performance and user experience without impacting the apps quality Hecht *et al.* (2016);

Palomba *et al.* (2019); Carette *et al.* (2017). Moreover, as shown in Table 3.4 these smell pairs are significantly associated.

For OO smells, our results are inline with prior findings in the literature on code smells co-occurrences in desktop applications Palomba *et al.* (2017b, 2018b, 2019). However, for Android smells, our findings reveal various interesting relationships that have not been yet explored previously in the context of Android apps development.

Table 3.4 Association rule mining results: the identified frequent itemsets of code smells co-occurrences

Code smell item set #1	Code smell item set #2	Support	Confidence	Lift	Chi-square p-values	Cramer's V
Feature Envy	Long Method	0.120	0.572	3.725	<0.0001	0.601
Long Method	Feature Envy	0.120	0.781	3.725	<0.0001	0.601
Long Method	Long Parameter List	0.122	0.793	3.412	<0.0001	0.568
Complex Class	Message chain	0.087	0.765	3.997	<0.0001	0.524
Complex Class	Feature Envy	0.073	0.637	3.037	<0.0001	0.382
Long Parameter List	Feature Envy	0.135	0.579	2.761	<0.0001	0.374
Feature Envy	Long Parameter List	0.135	0.642	2.761	<0.0001	0.374
Member Ignoring Method	No Low Memory Resolver	0.094	0.719	1.716	<0.0001	0.235
Leaking Inner Class	Member Ignoring Method	0.078	0.712	1.699	<0.0001	0.208
Message Chain	Member Ignoring Method	0.120	0.628	1.499	<0.0001	0.207
Member Ignoring Method	Slow Loop	0.086	0.681	1.625	<0.0001	0.202
Complex Class	Member Ignoring Method	0.074	0.651	1.554	<0.0001	0.169
Long Method	Member Ignoring Method	0.092	0.598	1.426	<0.0001	0.154
Long Parameter List	Member Ignoring Method	0.126	0.544	1.298	<0.0001	0.139

Summary for RQ2. Several pairs of code smells (14) tend to co-occur very often, including three pairs from Android-specific smells (e.g., *Leaking Inner Class* and *Member Ignoring Method*), seven pairs from OO code smells (e.g., *Long Method* and *Long Parameter List*) and four from both Android and OO smells (e.g., *Complex Class* and *Member Ignoring Method*).

3.5 Threats to validity

This section discusses threats to validity of the study.

1. *Threat to construct validity* could be related to the smells detection. Both Android and OO code smells were automatically detected using two widely used state-of-the-art tools. We are aware that our results can be affected by the presence of false positives and false negatives. While the performance of both tools has been evaluated in previous research. For Android smells, we used the aDoctor tool that has a precision of 98% and a recall of 98%

Palomba *et al.* (2017a, 2019); Bavota *et al.* (2015a). For OO smells, we used organic which is an implementations of rules published by Bavota et al. Bavota *et al.* (2015a). However, we cannot exclude that some code smells were missed by our analysis or false positives were considered.

2. *Threat to conclusion validity* could be related to the analysis methods used in our study. While we exploited association rule mining based on the Apriori algorithm, other methods such logistic regression could be used. A part of our future work, we plan to investigate the performance of other techniques.

   Another threat in this category might be related to the high diffusion of a certain code smell types. Indeed, frequent co-occurrences between the considered smells might simply be the results of the high distribution of one smell type, possibly indicating no causation in the observed relationships. Figure 3.2 reports the diffusion of code smells in terms of the percentage of classes affected by each smell type in the analyzed apps. We observe that the frequent associations between the considered smells indicated in Table 3.2 are not just the results of code smells disparity. Indeed, results indicate that some smells that were not involved in co-occurrences (e.g., *Lazy Class*, *Leaking Thread* and *Internal Getter And Setter*) are frequent comparing to other smells that were involved (e.g., *Complex Class*, *No Low Memory Resolver*). Therefore, the observed associations are not be just the result of the high diffuseness of single code smell types.
3. *Threat to external validity* are related to generalizability of our results. While we used a large sample of 1,923 open source Android apps written in Java, we cannot generalize our results to other open source or commercial mobile apps or to other technologies.

3.6 Implications

In this section, we discuss the implications that one can derive from our results.

- **Identifying refactoring opportunities to remove the co-occurrences of code smells.** Our study have shown that the phenomenon of code smell co-occurrences is highly spread in Android apps. It is widely accepted refactoring techniques can be used to remove code

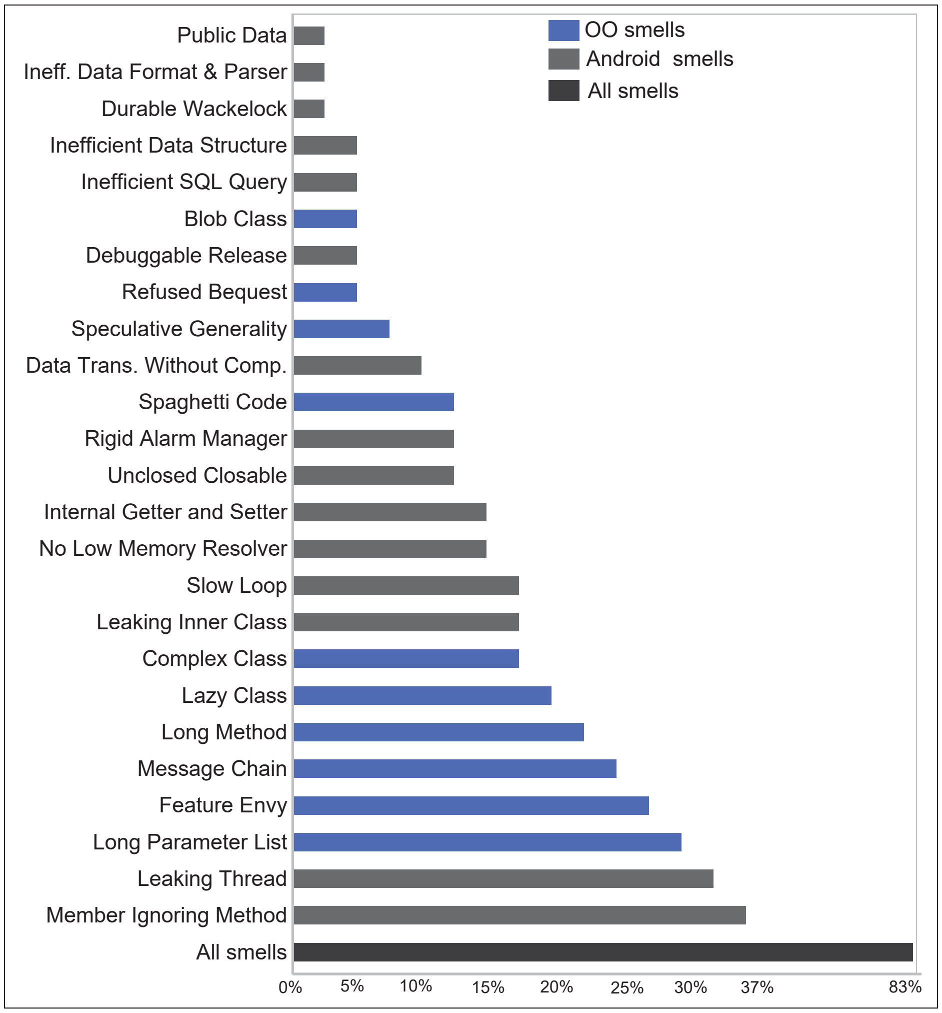

Figure 3.2 The diffusion of each code smell type across the studied applications

smells, hence, the use of such relationships from code smells co-occurrence (i.e., code smell pairs that frequently co-occur together) can provide a valuable knowledge to help identifying which refactoring strategies (e.g., primitive or composite refactorings) should be applied, and which are the most difficult co-occurrences to be refactored. Since this is one of the major research challenges, this study shed light on the importance of developing practical refactoring tools based on the information about co-occurrences of code smells.

- **Building code smell detection tools that have to regard the co-occurrence phenomenon**
  The frequent co-occurrence between some smells might represent an important source of

information code smell detection in mobile apps (e.g., observing together the traits marking a frequent code smell pair). Indeed, our results might be exploited to build more reliable code smell detection tools able to identify the location and the gravity of design problems affecting a class.

- **Understanding the impact of code smells co-occurrences on software quality.** Investigating the effects of the co-occurrences of code smells on software quality is crucial as it can bring unforeseen maintenance efforts and costs. Various studies have explored the effects of individual occurrences of code smells Santos *et al.* (2018); Martins *et al.* (2020); Palomba *et al.* (2018b); Yamashita & Moonen (2013b) in traditional software systems. On the other side, other works have shown that classes affected by more than one instance of code smells have a higher change-proneness and fault-proneness as compared to classes affected by a single instance Palomba, Di Nucci, Panichella, Oliveto & De Lucia (2016a). As our study indicates that developers can be often confronted with the phenomenon of code smell co-occurrences in their code base, therefore it is crucial to eliminate these anomalies in early stages of the development process to avoid the deterioration of their code. Hence, the research community can further perform an in-depth analysis on the impact of the co-occurrences of code smells on various structural quality aspects such internal and external quality attributes, and other performance quality aspects such as memory and energy consumption in the context of mobile apps. Such analysis can provide practical guidelines for mobile apps refactoring.

3.7 Conclusion and Future Work

In this study, we investigated the co-occurrence of code smells in Android apps on a large dataset of 1,923 open-source apps, 15 types of Android smells and 10 types of OO code smells. We jointly analyzed (1) the prevalence of the co-occurrence phenomenon, and (2) code smell pairs that most tend to co-occur. The key findings of our study indicate that *(i)* the co-occurrence phenomenon is quite prevalent in Android apps with 34%, 26% and 51% of classes are affected respectively by more than one Android, OO and both (i.e., OO and Android) smell types, and *(ii)* there exist 14 smell pairs frequently co-occur together. As future work, we plan to analyze other types of code smells and investigate the impact of code smell co-occurrences on internal and external quality attributes as well as other performance aspects. We also plan to develop customized Android app refactoring tools based on the information about co-occurrences of code smells.

CHAPTER 4

AN EMPIRICAL STUDY ON THE IMPACT OF REFACTORING ON QUALITY METRICS IN ANDROID APPLICATIONS

Oumayma Hamdi[a] , Ali Ouni[a] , Eman Abdullah AlOmar[b] , Mel Ó Cinnéide[c] , Mohamed Wiem Mkaouer[b]

[a] Department of Software and IT Engineering, École de Technologie Supérieure, 1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

[b] Department of Software Engineering, Rochester Institute of Technology, Rochester, NY, USA,

[c] Department of Software Engineering, University College Dublin, Dublin, Ireland,

Paper published in the *International Conference on Mobile Software Engineering and Systems*, March 2021

4.1 Abstract

Mobile applications must continuously evolve, sometimes under such time pressure that poor design or implementation choices are made, which inevitably result in structural software quality problems. Refactoring is the widely-accepted approach to ameliorating such quality problems. While the impact of refactoring on software quality has been widely studied in object-oriented software, its impact is still unclear in the context of mobile apps. This paper reports on the first empirical study that aims to address this gap. We conduct a large empirical study that analyses the evolution history of 300 open-source Android apps exhibiting a total of 42,181 refactoring operations. We analyze the impact of these refactoring operations on 10 common quality metrics using a causal inference method based on the Difference-in-Differences (DiD) model. Our results indicate that when refactoring affects the metrics it generally improves them. In many cases refactoring has no significant impact on the metrics, whereas one metric (LCOM) deteriorates overall as a result of refactoring. These findings provide practical insights into the current practice of refactoring in the context of Android app development.

4.2 Introduction

Android applications undergo modifications, improvements and enhancements to cope with rapid and evolving user requirements. Such maintenance activities can cause quality to decrease if improperly conducted Palomba *et al.* (2019); Hecht *et al.* (2016); Morales *et al.* (2017); Uchôa *et al.* (2020). In order to facilitate software evolution, developers need to improve software structure on a regular basis. Refactoring is the most common approach to improve the internal structure of software systems without affecting their external behavior Fowler *et al.* (1999); AlOmar *et al.* (2019); Ouni *et al.* (2016); Mkaouer *et al.* (2015); Murphy-Hill, Parnin & Black (2011).

Mobile apps differ significantly from traditional software systems Minelli & Lanza (2013); Mannan *et al.* (2016); Kessentini & Ouni (2017) in having to deal with limitations on specific hardware resources like memory, CPU, display size, etc., as well as the highly dynamic nature of the mobile app market and the ever-increasing user requirements. These differences can play an important role in mobile app development and evolution. Indeed, unlike object-oriented software systems AlOmar *et al.* (2019); Ó Cinnéide *et al.* (2012); Alshayeb (2009); Bavota *et al.* (2015a); Cedrim *et al.* (2016); Stroggylos & Spinellis (2007), the impact of refactoring on quality metrics in mobile apps has received little attention. Hence, much uncertainty exists about the relationship between refactoring and quality aspects in mobile apps. Yet, refactoring practices may exhibit different challenges in the context of Android apps. Even though refactoring aims at improving code structure, this expectation might not be always met in real settings as refactoring changes are often performed quickly to meet users requirements, fix defects or adapt to environment changes in the highly volatile mobile market Cedrim *et al.* (2016). To develop efficient and reliable refactoring support tools for mobile apps, there is a need to better understand the current refactoring practice and its impact on structural quality.

To fill this gap and improve the current knowledge about the impact of refactoring on structural quality, we conduct an empirical study on a dataset composed of 300 open-source Android apps that are freely distributed in the Google Play Store. We analyze the impact of 10 commonly used refactoring operations on 10 well-known quality metrics in Android apps. We identified a total of 42,181 applied refactoring operations and measured quality metrics values before and after each refactoring operation. Then we analyzed the impact of each refactoring on the considered quality metrics using a causal inference method based on the Difference-in-Differences (DiD) model, one of the widely-used analytical techniques for causal inference Angrist & Pischke (2008).

Overall, our study findings can be summarized as follows.

- Some refactoring types correlate with a broad improvement in software metric values. For example, the Move Method refactoring brings about a significant improvement in terms of coupling (CBO and RFC), cohesion (LCOM, TCC and LCC), complexity (WMC) and design size (LOC).
- The cohesion metric LCOM proved to be least consistent metric, improving for some refactoring types while deteriorating for others, and exhibiting an overall deterioration in response to refactoring. This resonates with earlier work showing LCOM to be very volatile under refactoring Ó Cinnéide *et al.* (2012), and inclined to deteriorate when coupling improves Paixao, Harman, Zhang & Yu (2017).
- Non-refactoring code changes tend to have a negligible impact on the majority of quality metrics, except for the design size-related metrics which tend to increase in most of the commits. It is to be expected that design size related metrics increase over time as a project evolves, following Lehman's law on software evolution Lehman (1996).

4.3 Study Design

This section describes the design of our empirical study. We first setup our research question. Then, we explain our experimental setup including data collection, and analysis methods employed.

Our main goal in this study is to investigate how refactoring affects structural quality metrics in Android apps. In particular, we address the following research question:

RQ. *Do refactorings applied by Android developers improve quality metrics?*

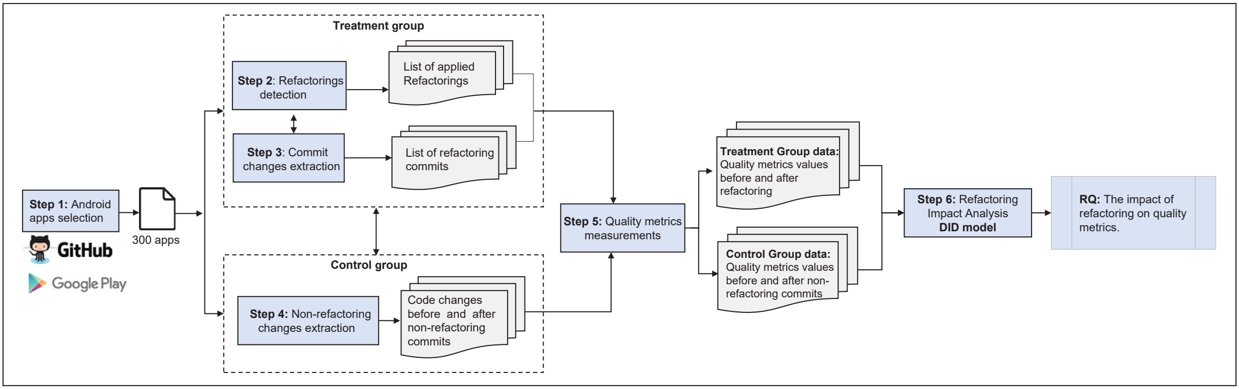
Treatment group
Step 2: Refactorings detection
List of applied Refactorings
Step 3: Commit changes extraction
List of refactoring commits
Step 1: Android apps selection
300 apps
GitHub
Google Play
Control group
Step 4: Non-refactoring changes extraction
Code changes before and after non-refactoring commits
Step 5: Quality metrics measurements
Treatment Group data: Quality metrics values before and after refactoring
Control Group data: Quality metrics values before and after non-refactoring commits
Step 6: Refactoring Impact Analysis DID model
RQ: The impact of refactoring on quality metrics.

Figure 4.1 The overall process of our empirical study

4.3.1 Empirical Study Setup

To address our research question, we design a controlled experiment where we select two groups of code changes, a first group that consists of refactoring-related change changes (i.e., treatment group), and a second that consists of a non-refactoring code changes (i.e., control group). Thereafter, we investigate the impact of both groups on quality metrics to allow statistical analysis. Figure 4.1 describes the overall process of our study which consists of six main steps: Android apps selection, refactoring extraction, commit extraction, non-refactoring changes extraction, Quality metrics measurement, and refactoring impact analysis.

4.3.1.1 Step 1: Android apps selection

We target open-source Android apps that are freely distributed in the Google Play store and have their versioning history hosted on GitHub. For this purpose, we performed a custom search on GitHub by targeting all Java repositories in which the `readme.md` file contains a link to a Google Play Store page. Overall, we obtained 19,212 apps. Thereafter, inspired by previous works Das *et al.* (2016); Malavolta *et al.* (2018), we applied the following filters to exclude:

- Apps whose GitHub repository does not contain an `AndroidManifest.xml` file as they clearly do not refer to real Android apps. The result of this filter was a collection of 5,766 apps.
- Apps for which the corresponding Google Play page is not existing anymore. This filter returned 3,160 apps.
- Repositories that contain forks of other repositories. This filtering step leads to a final set of 1,923 Android apps

Thereafter, we randomly selected a representative set of 300 apps which represents 15% of the final set, exhibiting a total of 42,181 refactoring operations. We focused our study to this set of apps for computational reasons. It is worth noting that the sample size of 300 apps and 42,181 refactoring operations is larger than related studies on the impact of refactoring on software quality Bavota *et al.* (2015a); Cedrim *et al.* (2016); Alshayeb (2009), and than typical samples in software engineering research Wohlin *et al.* (2012). Table 4.1 summarizes the statistics about the collected dataset.

Table 4.1 Dataset statistics

Statistic	**Count**
# of Android apps	300
# of commits with refactorings	13,500
# of refactoring operations	42,181
Total number of commits	271,263

4.3.1.2 Step 2: Refactorings detection

In this step, we collect all the refactoring operations applied to the studied apps. We utilize RefactoringMiner Tsantalis *et al.* (2018b) to detect applied refactoring instances on the commit level. RefactoringMiner is a command-line based open source tool that is built on top of the UMLDiff Xing & Stroulia (2005) algorithm for differencing object-oriented models. RefactoringMiner has been shown to achieve a precision of 98% and recall of 87% Tsantalis *et al.* (2018b); Silva *et al.* (2016). The tool walks through the commit history of a project's Git repository to extract refactorings between consecutive commits. RefactoringMiner supports the detection of various common refactoring types from Fowler's catalog. Among the supported refactorings, all refactoring types detected by Refactoring Miner were considered in this study, except the Rename Method and Rename Class refactorings as they are not directly related to one of the structural metrics studied in our study. Overall, our extraction process identifies a list of 10 common refactoring types which are amongst the most common refactoring types Murphy-Hill *et al.* (2011); Cedrim *et al.* (2017, 2016); Ouni *et al.* (2016); AlOmar *et al.* (2019). Table 4.2 reports the list and the number of refactorings, respectively, that are investigated in our study.

Table 4.2 The list of refactoring applied to the analyzed apps

Refactoring type	Number
Extract Method	11,736
Move Attribute	8,321
Move Method	5,847
Extract And Move Method	5,121
Inline Method	3,952
Push Down Method	2,541
Pull Up Attribute	1,371
Pull Up Method	1,170
Extract SuperClass	1,140
Move Class	982
Total	42,181

4.3.1.3 Step 3: Commit changes extraction

After the extraction of all refactoring operations, we collect the IDs of all refactoring commits, i.e., commits in which a refactoring operation was applied, as well as the IDs of the commits that immediately precede the refactoring commit. The GitHub API facilitates this process; in particular, we use the `git clone` command to download the source code of each refactoring commit as well as its immediately preceding commit. These commits enable the identification of quality metrics values before and after the application of refactoring.

4.3.1.4 Step 4: Non-refactoring changes extraction

In this step, we extract a set of commits that contain non-refactoring changes for our controlled experiment. To do this, based on the treatment group, we randomly selected a set of non-refactoring commits representing our control group. For each commit, we collected its ID as well as the commits that precede it. Thereafter, we performed the same procedure adopted in Step 3 to collect their source code.

4.3.1.5 Step 5: Quality metrics measurement

To assess the impact of refactoring on software quality, we need to measure a set of quality metrics. In particular, we measure for each applied refactoring change as well as non-refactoring changes, the class level metrics before and after the change has been applied in the commit level. Specifically, since we already have the list of refactoring operations applied in each commit, we compute for each class the quality metric values before and after each commit in both treatment and control groups. To calculate the values of these metrics we utilized a widely-used open source CK Metrics Suite tool, namely, CK-metrics,which is a command-line based tool provided by Aniche Aniche (2016) that allows automating our dataset collection process.

4.3.1.6 Step 6: Refactoring Impact Analysis

In this step, we investigate whether or not each metric is improved by refactoring. In order to do this, we set up two hypotheses, the null hypothesis H_0 assumes that a refactoring operation r_i does not improve a quality metric m_j, and the alternative hypothesis H indicates that the refactoring r_i improves m_j.

After collecting the metric values before and after each commit in both treatment and control groups, we calculate the differences between their quality metric values before and after the refactoring change, at the class level. Thereafter, we use two statistical methods statistical significance, and causal inference.

1. **Statistical Significance Analysis.** To capture the overall trends of the variation in the metric values we use statistical significance analysis. To do so, for each refactoring operation r_i, and each metrics m_j, we use the Wilcoxon rank-sum test Wilcoxon, Katti & Wilcox (1970), a non-parametric test, to assess the statistical differences between the distribution of m_j before and after the application of r_i. In addition to the Wilcoxon test, we used the non-parametric effect Cliff's delta (δ) Cliff (1993) to compute the effect size, i.e., the magnitude of the difference between the distributions. The value of effect size is statistically interpreted as:
   - $Negligible:$ if $\mid \delta \mid < 0.147$,
   - $Small:$ if $0.147 \leq\mid \delta \mid < 0.33$,
   - $Medium:$ if $0.33 \leq\mid \delta \mid < 0.474$, or
   - $High:$ if $\mid \delta \mid \geq 0.474$.

   Furthermore, to better assess the impact of a specific refactoring operation on quality metrics, we performed a causal inference experiment to assess whether the metrics variations are due to the refactoring changes or to other code changes.
2. **Causal Inference Analysis.** Causal inference stems from the social sciences and explores cause and effects as its main concern Angrist & Pischke (2008). In econometrics, difference-in-differences (DiD) methods are one of the key analytical elements for causal inference. DiD is used to statistically analyze actual and counterfactual scenarios, thereby enabling a causality analysis. To investigate the effects of a treatment in statistics, one cannot see the results with and without an intervention based on one individual only. As shown in Figure 4.2, the DiD model addresses this problem by comparing two groups, a group with the intervention, called *treatment* group (i.e., a set of code changes with refactoring) and a group without it, called a *control* group (i.e., a set of code changes without refactoring). The underlying assumption of DiD design is that the trend of the control group provides an adequate proxy for the trend that would have been observed in the treatment group in the absence of treatment. Let, T and C, the treatment and the control group, respectively. The refactoring impact RI of a given refactoring operation R on a given quality metric M_i is calculated as follows:

$$RI(R, M_j) = Y^R_{M_i} - Y^C_{M_i} \tag{4.1}$$

where $Y^R_{M_i}$ is the median perceived impact after the application of the set of refactorings R on the treatment group T on the metric M_i; and $Y^C_{M_i}$ is the median perceived change in the control group C on the metric M_i.

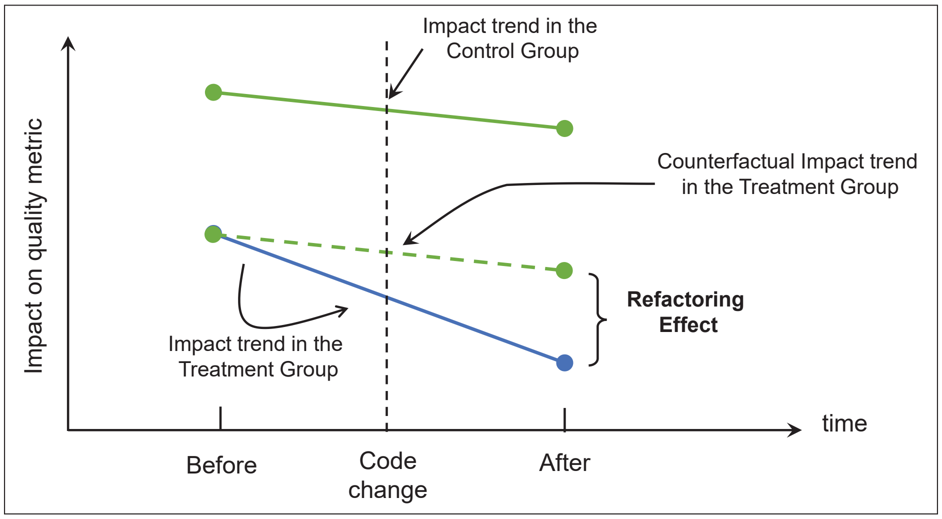
Impact trend in the Control Group
Counterfactual Impact trend in the Treatment Group
Refactoring Effect
Impact trend in the Treatment Group
Impact on quality metric
time
Before
Code change
After

Figure 4.2 An example of the causal inference method using a DiD model showing the refactoring impact on a quality metric before versus after the application of refactoring

4.4 Empirical study results

This section reports and discusses our experimental results to address our research question: *do refactorings applied by Android developers improve quality metrics?* To answer this question, for each commit change of both groups, described in 4.3, we compute its corresponding metric values before and after each commit in both treatment and control groups. Figures 4.3 and 4.4 show the general distribution of the metrics values before and after commit changes in the treatment, and control groups, respectively. We also provide a detailed analysis in Table 4.3 where each column reports:

1. The impact of the respective refactoring type based on the DiD technique using Equation 4.1.

2. The predominant behavior indicating whether the refactoring impact is positive or negative.
3. The p-value as well as the Cliff's delta (δ).

In the following, we report and discuss the obtained results for each quality metric along with real world examples from our experiments.

Table 4.3 The impact of refactoring (treatment group) and non-refactoring (control group) changes on quality metrics

Data	Change	Measure	Coupling			**Cohesion**			**Complexity**	**Design Size**		**Inheritance**
			CBO	RFC	NOSI	LCOM	TCC	LCC	WMC	LOC	VQTY	DIT
Treatment Group	Extract Method	Refactoring impact	0	-4	0	3	0	0	-2	-4	0	0
		Behavior	-	↓	-	↑	-	-	↓	↓	-	-
		P-value (δ)	0.06 (S)	<0.05 (S)	0.10 (S)	0.07 (N)	0.18 (S)	0.18 (S)	<0.05 (S)	<0.05 (S)	0.06 (N)	0.06 (N)
	Move Attribute	Refactoring impact	0	-9	5	1	0.1	0.1	0	0	-12	0
		Behavior	-	↓	↑	↑	↑	↑	-	-	↓	-
		P-value (δ)	<0.05 (N)	<0.05 (N)	0.17 (N)	<0.05 (N)	<0.05 (N)	<0.05(N)	0.40 (N)	0.13 (N)	<0.05 (S)	0.1 (N)
	Move Method	Refactoring impact	-8	-4	0	-2	0.3	0.3	-3	-13	0	0
		Behavior	↓	↓	-	↓	↑	↑	↓	↓	-	-
		P-value (δ)	<0.05 (S)	<0.05 (N)	0.70 (N)	<0.05 (S)	<0.05 (N)	<0.05 (N)	<0.05 (S)	<0.05 (M)	0.11 (N) (M)	0.15 (S)
	Extract And Move Method	Refactoring impact	-8	-11	0	-2	0.7	0.7	-3	-12	0	0
		Behavior	↓	↓	-	↓	↑	↑	↓	↓	-	-
		P-value (δ)	<0.05 (S)	<0.05 (N)	0.33 (N)	<0.05 (S)	<0.05 (N)	<0.05 (N)	<0.05 (S)	<0.05 (M)	0.08 (N)	0.07 (N)
	Inline Method	Refactoring impact	-3	0	0	-4	0	0	-1	0	-5	0
		Behavior	↓	-	-	↓	-	-	↓	-	↓	-
		P-value (δ)	<0.05 (N)	0.82 (N)	1 (N)	0.31 (N)	0.10 (N)	0.10 (N)	0.17 (S)	1(N)	0.59 (N)	1 (N)
	Push Down Method	Refactoring impact	-10	0	0	3	0	0	1	-15	-3	-7
		Behavior	↓	-	-	↑	-	-	↑	↓	↓	↓
		P-value (δ)	<0.05 (M)	1 (N)	1 (S)	1 (N)	1 (N)	1 (N)	0.10 (N)	<0.05 (M)	<0.05 (N)	<0.05 (S)
	Pull Up Attribute	Refactoring impact	-4	0	0	0	0	0	-2	-9	0	-10
		Behavior	↓	-	-	-	-	-	↓	↓	-	↓
		P-value (δ)	<0.05 (S)	1 (N)	1 (N)	0.08 (N)	<0.05 (N)	<0.05 (N)	<0.05(N)	<0.05 (S)	0.08 (N)	<0.05 (M)
	Pull Up Method	Refactoring impact	-11	2	1	2	0.01	0.01	-3	-16	-7	-5
		Behavior	↓	↑	↑	↑	↑	↑	↓	↓	↓	↓
		P-value (δ)	<0.05 (M)	0.51 (N)	1 (N)	<0.05 (N)	1 (N)	1 (N)	0.03 (N)	<0.05 (M)	<0.05 (S)	<0.05 (S)
	Extract Super Class	Refactoring impact	-9	-8	0	0	0	0	-10	-24	-1	-8
		Behavior	↓	↓	-	-	-	-	↓	↓	↓	↓
		P-value (δ)	<0.05(M)	<0.05 (S)	0.88 (N)	0.09 (N)	1 (N)	1 (N)	<0.05 (M)	<0.05 (M)	1 (N)	<0.05 (S)
	Move Class	Refactoring impact	0	0	-1	2	0	0	-1	-6	-1	-3
		Behavior	-	-	↓	↑	-	-	↓	↓	↓	↓
		P-value (δ)	<0.05 (S)	1 (N)	0.17 (N)	<0.05 (N)	1 (N)	1 (N)	<0.05 (N)	<0.05 (N)	0.10 (N)	<0.05 (S)
Control Group	Commit change	Change commit	-1	0	0	-3	0	0	0	10	1	0
		Behavior	↓	-	-	↓	-	-	-	↑	↑	-
		P-value (δ)	0.08 (N)	0.11 (N)	0.73 (N)	0.10 (N)	0.97 (N)	0.97 (N)	0.07 (N)	<0.05 (S)	<0.05 (N)	0.34 (N)

Legend:
Metric improvement: Low 🟦 High
Metric disprovement: Low 🟥 High

Effect size: L: Large, M: Medium, S: Small, N: Negligible
Behavior: ↑ : indicates that the metrics increased; ↓ : indicates that the metric decreased; - : indicates that the metric remains unaffected.

4.4.1 Replication package

Our dataset is available in our replication package for future replications and extensions Dataset (2021b).

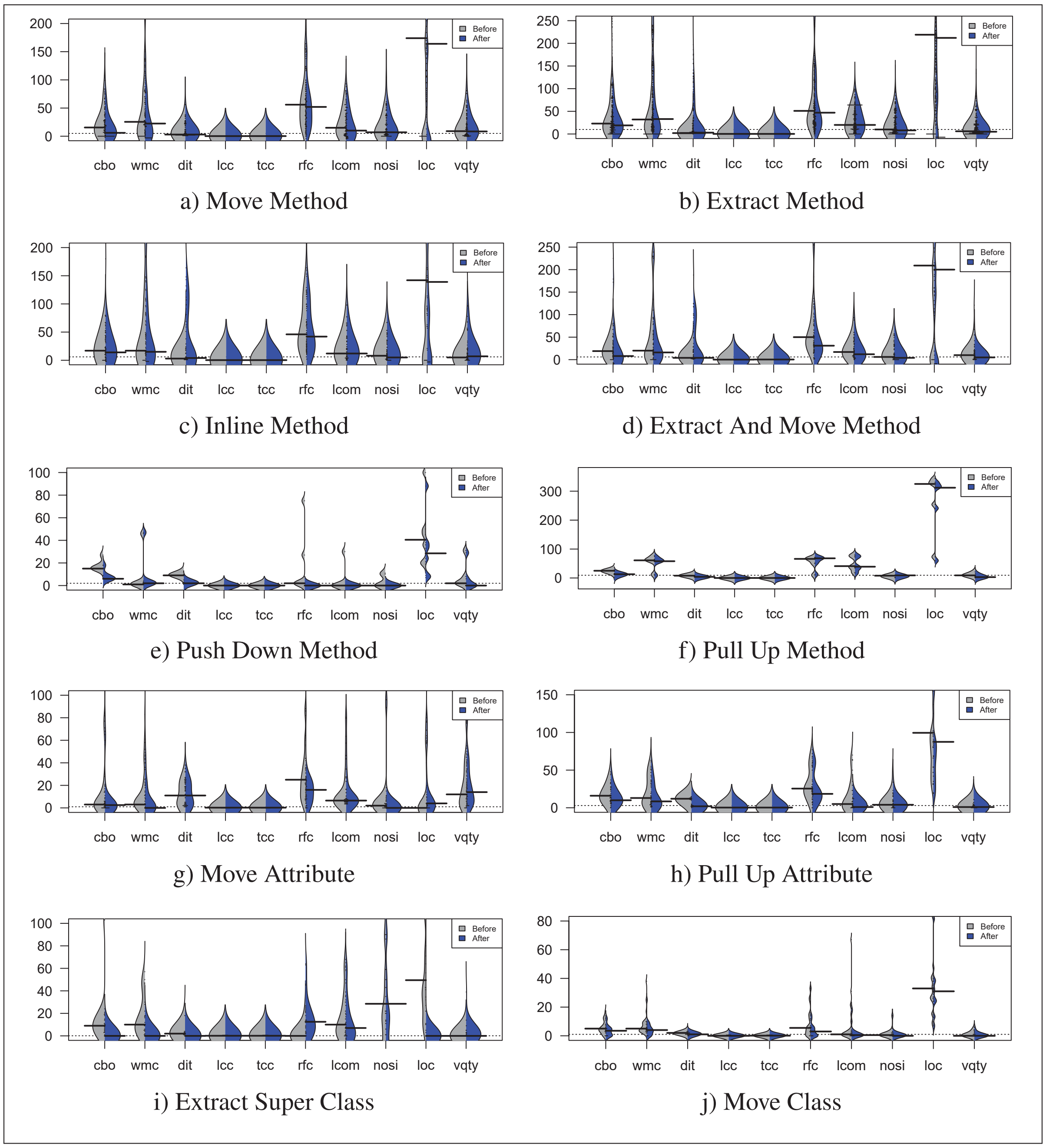

Figure 4.3 Treatment group results: Beanplots of metric values before and after each refactoring operation

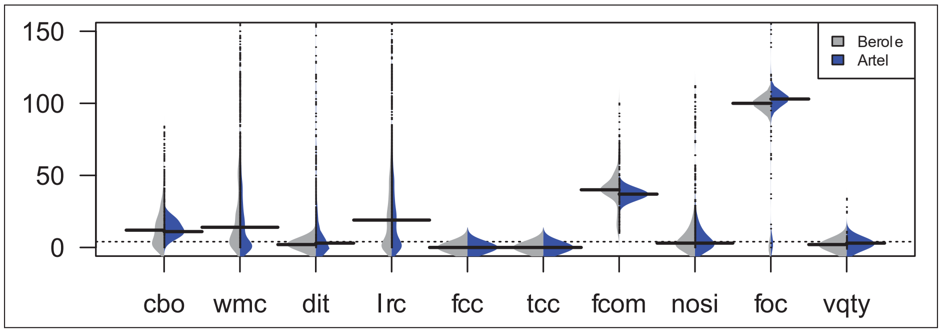

Figure 4.4 Control group results: The impact of non-refactoring code changes on quality metrics

4.4.2 Results for Coupling Metrics

Coupling is defined as the strength of the dependencies that exist between classes Chidamber & Kemerer (1994); Stevens, Myers & Constantine (1974). Low coupling is desirable since it helps in isolating responsibilities and changes. As shown in Table 1.4, we assess three coupling metrics. The first is the Coupling Between Object (CBO), counting the number of dependencies a class has (i.e., the number of other classes it depends on). The second metric is the Response for a Class (RFC), calculated as the number of distinct methods and constructors invoked by a class. The third is Number Of Static Invocations (NOSI) which counts the number of invocations of static methods. The higher the CBO, RFC and NOSI the worse is the class coupling.

4.4.2.1 CBO

From the beanplots in Figure 4.3 and Table 4.3, we observe that several applied refactorings improve the CBO metric (i.e., decrease its value). The most influential refactorings are *Pull Up Method*, *Push Down Method*, and *Extract Super Class* significantly reducing CBO from 9 to 11 with a medium effect size. These refactorings are typically applied when classes or subclasses grow and develop independently of one another, causing identical (or similar) methods each having its own dependencies. They often help reducing duplicate code, replacing inheritance with delegation and vice versa as well as reducing dependencies through polymorphism. Furthermore, *Move Method* or *Extract And Move Method* tend to significantly improve CBO by a median value of 8, each with a small effect size. Typically, these method-level move refactorings help organizing functionalities across classes and thus reduce dependencies between them. Overall, as shown in Table 4.3, the CBO variation for all refactorings is significant and accompanied with a medium or small effect size depending on the refactoring type. Particularly, the effect size is negligible for the *Inline Method*, as it could be applied either in methods from the same class or from different classes. Only the latter can help reducing coupling.

4.4.2.2 RFC

As it can be seen from the beanplots in Figure 4.3 and Table 1.4, *Extract And Move Method*, and *Move Attribute* are the most influential refactoring that improve RFC by 11 and 9, respectively. Moreover, *Extract Super Class* have shown to improve RFC by a median score of 8, while less impact is observed by both *Extract Method* and *Move Method* refactorings with a median of 4, each.

4.4.2.3 NOSI

From Figure 4.3 and Table 4.3, we observe that the NOSI metric has not been impacted by any of the applied refactorings since when comparing the distributions of values before and after refactoring, no statistically significant difference is observed. This is not very surprising, as most of refactorings do not have a direct impact on static methods.

It is worth noting that our findings in Android apps share some similarities with desktop apps for the coupling which is positively impacted after applying the refactoring Pantiuchina *et al.* (2018); Moser, Abrahamsson, Pedrycz, Sillitti & Succi (2007). Moreover, the the *Extract Method* and *Move Method* refactorings are applied in both Android and desktop applications to improve the coupling Bavota *et al.* (2015a); Fernandes *et al.* (2020).

To observe the salient impacts of refactoring on coupling, we refer to a real world example from our dataset showing the impact of a *Move Method* refactoring on coupling metrics, from the WordPress-Android app, in the commit[19]. The commit's refactoring consists of moving of the method `showJetpackSettings()` from the class `EditPostActivity` to the class `ActivityLauncher`. Interestingly, this Move Method refactoring resulted in a coupling reduction for the `ActivityLauncher` class, with a drop of its CBO from 18 to 13 and its RFC from 33 to 31. Looking deeper into the the source code to understand the reason behind these improvements, we find that the *Start Jetpack security settings* activity was initially launched from the `ActivityLauncher` class via the `startActivity()` method. This method use the `intent` object to start this activity. However, `intent` was intitially implemented in the `showJetpackSettings()` method in the *EditPostActivity*. As consequence, each time this activity is launched, the class `ActivityLauncher` calls the `showJetpackSettings()` method. Thus, this refactoring helped moving the method to the class that uses it most which decreased the number of dependencies between both classes, resulting in an improvement in both CBO and RFC.

Finding 1. Refactoring has a significant positive impact on coupling in terms of both the CBO and RFC metrics, while no significant impact was found on the NOSI metric. The most influential refactorings that promote low coupling are Move Method, and Extract And Move Method.

4.4.3 Results for Cohesion Metrics

Cohesion assesses the degree to which the responsibilities implemented in a class belong together Stevens *et al.* (1974). High cohesion is desirable since it promotes encapsulation and adherence to the Single Responsibility Principle, one of the SOLID design principles Joshi (2016). In this study, we consider three cohesion metrics, the first metric is the normalized Okike (2010), considering the shared instance variables between method pairs of a class. If the value of this metric is low, it indicates a strong cohesiveness of the class. The second cohesion metric is TCC, which considers the direct connection of public methods in a class. The third is LCC, which is similar to TCC but additionally considers the indirect connection of public methods in a class.

[19] https://github.com/wordpress-mobile/WordPress-Android/commit/ead8683e044a70fb3b288d562966c7ed442b8925

TCC and LCC provide another way of measuring the cohesiveness of a class. The higher the TCC and LCC a values are, the more cohesive is the class. It is anticipated that cohesion may be improved by moving-related refactoring operations. In general, moving a method that does not access local attributes or methods, or is called by few local methods improves cohesion.

4.4.3.1 LCOM

The beanplots from Figure 4.3 and Table 4.3 show that LCOM is improved when applying *Move Method* and *Extract And Move Method* refactorings. For both refactorings, the median values significantly decreased by 2, even though they are accompanied by a small effect size. However, we also observe from the results in Table 4.3 that the *Move Attribute*, *Pull Up Method* and *Move Class* refactorings caused LCOM metric to disprove by a median of 1 and 2 with a negligible effect size.

These results suggest that the LCOM metric can either improve or disprove after refactoring, and developers need to pay attention to cohesion when modifying their code and use appropriate refactoring operations. It is worth noting that our results match previous work observations for the cohesion, i.e., cohesion worsens rather than improves after the refactoring application Stroggylos & Spinellis (2007); Fernandes *et al.* (2020).

4.4.3.2 TCC

As can be seen in beanplots of Figure 4.3 and Table 4.3, *Extract And Move Method* is the most influential refactoring on TCC which improve it by 0.7. Whereas, *Move Method* and *Move Attribute* refactorings tend to have less impact on the metrics with a median improvement of 0.3, 0.1, respectively. It is worth noting that the differences are statistically significant even though with negligible effect size.

4.4.3.3 LCC

We found that LCC achieves similar results to the TCC metric. This was expected since both metrics reflect similar cohesion characteristic as mentioned earlier in Section 4.4.3, except that LCC further involves the number of indirect connections between visible classes. Thus, the constraint $LCC \geq TCC$ holds always. Upon a qualitative investigation of our dataset, we observe that moving methods from one class to another is a popular and effective refactoring to improve cohesion, as it often involves adding a parameter when resources of the original class are used, and removing that parameter which is an instance of the target class.

As an illustrative example, we refer to the WordPress-Android app from the commit [20], we observe that the method `onDraw()` is moved from `GraphView` class to the *GraphViewContentView* class which makes the class `GraphView` more cohesive with an improvement in each of TCC and LCC from 0.2 to 0.5 and an improvement of LCOM from 32 to 28.

Finding 2. Cohesion quality metrics, LCOM, TCC and LCC, tend to exhibit statistically significant variations with attribute and method-level moving-related refactoring operations. The refactorings that most influence cohesion are Move Attribute, Move Method and Extract And Move Method. However, LCOM tend to be more volatile under refactoring, which suggests Android developers to pay attention when dealing with cohesion.

4.4.4 Results for Complexity Metrics

Reducing code complexity is one of the main challenges in any software system and one of the prominent goals of refactoring. We use the Weighted Methods per Class (WMC) to assess class complexity Chidamber & Kemerer (1994). WMC for a given class is computed as the sum of the McCabe's cyclomatic complexity of its methods McCabe (1976). Being a direct metric, the higher WMC, the higher is the class complexity.

[20] https://github.com/wordpress-mobile/WordPress-Android/commit/71a2b5277623415a7657accefc57c6599455aa3c

The distributions of the WMC metric depicted in Figure 4.3 and Table 4.3 indicate a significant improvement after applying *Extract Super Class* refactoring by a median value of 10 with a medium effect size. Indeed, this refactoring operation is effective to remove code duplication and thus reducing complexity. Duplicate code often occurs when two classes perform similar tasks in the same way, or perform similar tasks in different ways. As a consequence, extracting a superclass can concentrate similar tasks and provide a built-in mechanism for simplifying such situations and removing duplicate code via inheritance. Moreover, various other refactorings tend to also improve WMC including *Move Method*, *Extract And Move Method*, *Pull Up Attribute*, *Extract Method*, and *Move Class* refactorings, but with less impact varying from 1 to 3 with negligible or small effect size. These improvements are expected since the applied refactoring operations deal with the simplification of methods inside a class. Particularly, the extraction of sub-methods that tend to break down long methods, or moving the methods to the appropriate class which decrease the complexity of the methods in the class.

An interesting example that shows the impact of Extract Method refactoring on complexity was found in the ownCloud app from commit[21] that involves the extraction of `readIsDeveloper()` method from `onCreate()` method in the `MainApp` class and the extraction of `showDeveloperItems()` method of `onCreate()` in the `Preferences` class. These refactorings reduce the complexity by decreasing the WMC metric from 7 to 3. Even though this commit is part of a pull request (# 13401), the developer tend to take care of the quality through refactoring commits.

Finding 3. Several refactorings types tend to improve complexity by decreasing the WMC metric. The most impactful refactorings are Extract Super Class, Extract Method, and Move Method which typically help simplifying methods structure and/or reducing duplicate code.

[21] https://github.com/owncloud/android/commit/7e460488f7fb2179c4476332dd5142a110450297

4.4.5 Results for Design Size Metrics

The design size is an indication of code density. We use two common metrics to estimate the size. The first is the Lines of Code (LOC) which counts the number of lines of code ignoring spaces and comments. The second metric is the Variables Quantity (VQTY) that counts the number of declared variables.

4.4.5.1 LOC

As shown in Figure 4.3 and Table 4.3, the refactorings *Extract Super Class*, *Pull Up Method*, *Push Down Method*, *Move Method*, and *Extract And Move Method* are the most influential refactorings that improve the LOC with a median ranging from 12 to 24 and a medium effect size. We notice that the size of the code elements significantly improve after the application of inheritance-related refactorings, as well as composing and moving method refactorings. For example, developers tend to apply *Extract Super Class* to reduce the size of its subclasses and reduce duplicated code, or *Extract And Move Method* to avoid code duplication and simplify the structure of the code.

4.4.5.2 VQTY

We observe from the results in Figure 4.3 and Table 4.3 that VQTY is less impacted by refactoring than LOC with only 3 refactorings including *Move Attribute*, *Pull Up Method*, and *Push Down Method* which reduce significantly the Quantity variables metric with either negligible or small effect size. Similar to LOC, we speculate that moving and inheritance-related refactorings help reducing code duplication which will in turn reduce the number of declared variables in code fragments and/or improve code reuse. It is worth noting that our results match also with desktop applications Fernandes *et al.* (2020); Bavota *et al.* (2015a). As an illustrative example, we refer to the WordPress-Android app, commit[22] which implements an *Extract Super Class*.

[22] https://github.com/wordpress-mobile/WordPress-Android/commit/3ff275dbf59b685666cc56ba8d094c1744955a5a

The refactoring extracted the class `TagsFragment` from `PostSettingsTagsFragment` which clearly resulted in a reduced size.

Finding 4. Most refactoring types tend to reduce the design size metrics LOC and VQTY. The most influential refactorings are *Extract Super class*, *Pull UP/Push Down Method* and *Move Method* which typically help reducing duplicate code, or moving code between classes, hence improving the design size.

4.4.6 Results for inheritance

The inheritance is a key concept in any object-oriented (OO) programming infrastructure such as Android. Designing and implementing the inheritance relations in an Android app is expected to improve the overall quality of the app such as software reuse and extensibility. The depth of inheritance tree (DIT) is the most used metric to assess the inheritance in OO software applications.

We notice from Figure 4.3 and Table 4.3 that five from all the applied refactorings do improve the DIT metric, including *Pull Up Attribute*, *Extract Super Class*, *Push Down Method*, *Pull Up Method*, and *Move Class*. The majority of these refactorings deal with changes applied to the class hierarchy. We expect that refactoring types that are mainly managing class inheritance do impact the DIT metric. A recent study showed that inheritance-related refactorings such as *Extract Super Class* and *Pull Up Method* tend to improve the depth of the inheritance to support software reusability and help in the elimination of code duplication AlOmar *et al.* (2020b). Our qualitative analysis has shown scenarios of moving method down, from a super class, to a child class, for the purpose of sharing its behavior which is relevant only for some of its subclasses. One of the examples that show the inheritance improvement was found in the Nextcloud app, in commit ID[23]. Specifically, the developer applied a *Push Down Method* refactoring operations involving the class `AbstractIT` and its subclass `AbstractOnServerIT`. This was realized through pushing down the `after()`, `deleteAllFiles()`, `createDummyFiles()` and

[23] https://github.com/nextcloud/android/commit/db6c1ba0554e5468ad568efe52c862c99ba7379c

`waitForServer()` methods from `AbstractIT` to its subclass. These changes resulted in inheritance improvement for the `AbstractIT` class, with a drop of its DIT from 5 to 3.

Finding 5. Hierarchy-level refactorings tend to improve the inheritance quality attribute (DIT). The most influential refactorings being Pull Up Attribute, Extract Super Class and Push Down Method. Improving the inheritance typically helps sharing common behavior across subclasses, reduces code duplication, and increases reusability which is a common practice in Android development.

Looking at the control group results from Figure 4.4 and Table 4.3, we noticed that the different quality metrics did not exhibit any significant change with non-refactoring changes (control group), except for the LOC and VQTY metrics that tend to increase after each commit. Indeed, it is normal that the design size related metrics increase over time as the project evolves. These results provide more evidence that the metrics changes observed in the experiment data are due to refactoring activities and not to chance.

4.5 Implications and Discussions

4.5.1 Implications for researchers

Further exploit quality metrics and refactoring in mobile software development. The existing literature discusses different automatic refactoring approaches that help practitioners in detecting anti-patterns or code smells. More recently, Baqais and Alshayeb Baqais & Alshayeb (2020) show that there is an increase in the number of studies on automatic refactoring approaches and researchers have begun exploring how machine learning can be used in identifying refactoring opportunities. Since the features play a vital role in the quality of the obtained machine learning models, this study can help determine which metrics can be used as effective features in machine learning algorithms to accurately predict refactoring opportunities at different levels of granularity (i.e., class, method, field), which can assist developers in automatically making their decisions. For example, using the most impactful metrics as a feature to predict whether a given piece of code should undergo a specific refactoring operation make developers more confident in accepting the recommended refactoring. Such knowledge is needed as, in practice, the built model should require as little data as possible.

4.5.2 Implications for practitioners

Android developers should be careful about their apps code quality. Our results indicate that developers can apply refactoring operations that do not improve their apps structural quality during refactoring, and particularly for the cohesion metric, LCOM. While LCOM tend to be very volatile under refactoring as also shown in prior works Ó Cinnéide *et al.* (2012); Paixao *et al.* (2017), these results indicate that there is a risk that developers degrade their apps structural quality while performing refactoring changes. Given that Android apps should evolve quickly to add new user requirements, fix bugs or adapt to new technological changes, such refactorings may increase technical debt and thus cause developers to invest additional maintenance effort in the future in order to fix quality issues in their apps. Hence, developers need to pay attention to their refactoring edits.

Need for Android-specific refactoring tools. Our findings on the impact of refactoring on quality attributes/metrics can help build practical and customized refactoring recommendation tool for Android developers. For example, given the relatively small size and rapid evolution and release cycles of mobile apps, it is relevant to recommend refactoring opportunities for classes suffering from specific quality aspects, e.g., coupling, complexity, etc.

4.5.3 Implications for educators

Learn refactoring best practices. Teaching the next generation of engineers best practices on refactoring and its impact on software quality in mobile apps and in software development, in general. Educators can use our study results and our dataset dataset (2020) to teach and motivate students to follow best refactoring practices while avoiding refactoring changes that may cause regression in their apps. In particular, our real world dataset of 42,181 refactorings from 300

Android apps, represents a valuable resource that could enable the introduction of refactoring to students using a “*learn by example*" methodology, illustrating best refactoring practices that should be followed and bad practices to be avoided.

4.6 Threats to validity

This section discusses threats to validity of the study.

1. *Threat to internal validity:* The accuracy of the refactoring detection tool, Refactoring Miner, can represent a threat to internal validity because it may miss the detection of some refactorings. However, previous studies report that Refactoring Miner has high precision and recall scores (98% and 87%, respectively) compared to other state-of-the-art refactoring detection tools Silva *et al.* (2016); Tsantalis *et al.* (2018a), which gives us confidence in using the tool. Furthermore, the CK-metrics tool could also have its own threats. While we conducted a manual inspection and double checked the values of the studied metrics with an alternative commercial tool, namely Scitools Understand, to make sure that the tool is reliable, still there could be errors that we did not notice. Another threat to internal validity could be related to the size of commit changes. In particular, the metric change in a given refactoring commit may or may not be related to the refactoring itself that occurred in that commit. To mitigate this problem, we adopted a widely-used causal inference method based on the Difference-in-Differences model that compare two groups, a treatment and a control group.
2. *Threats to construct validity:* A potential construct threat to validity could be related to the set of metrics being studied, as it may miss some properties of the selected internal quality attributes. To mitigate this threat, we select well-known metrics that cover various properties of each attribute, as reported in the literature Chidamber & Kemerer (1994); AlOmar *et al.* (2019).
3. *Threats to conclusion validity:* Unlike other works on the impact of refactoring on quality metrics Bavota *et al.* (2015a); AlOmar *et al.* (2019); Stroggylos & Spinellis (2007); Cedrim *et al.* (2016), we employed the DiD method to compare the changes in quality metrics between a treatment and control group. Moreover, we used the non-parametric Wilcoxon

rank-sum test and the Cliff's effect size, that do not make assumptions on the underlying data. As part of our future work, we plan to explore other quality aspects in mobile apps.

4. *Threat to external validity:* While we used a large sample of 300 open source Android apps written in Java, we cannot generalize our results to other open source or commercial mobile apps or to other technologies.

4.7 Conclusion and Future Work

We presented a study aimed at investigating the impact of refactoring on quality metrics in Android apps. We mined 300 open-source apps containing 42,181 refactoring operations in total. We determined the effect each refactoring had upon the 10 chosen software quality metrics, and employed the difference-in-differences (DiD) model to determine the extent to which the metric changes brought about by refactoring differ from the metric changes in non-refactoring commits.

In one sense, our anticipated results were that the benefits of refactoring would be clearly reflected in the changes brought about in the software metrics. The observed results were not that simple however. For most refactoring type and metric combinations, the refactoring produced no significant change in the metric. On the other hand, some refactoring types yielded a broad improvement in several metric values. LCOM stood out as the least consistent metric, improving for some refactoring types and disimproving for others. For the non-refactoring commits, the metrics exhibit no significant change, other than (unsurprisingly) the design size metrics.

As future work, we plan to analyze other refactoring types and investigate their impact on internal and external quality attributes. We also plan to extend our study to more open source and commercial Android apps to better generalize our results, and to develop Android specific refactoring tools to better support developers during maintenance and evolution.

CONCLUSION AND RECOMMENDATIONS

In this chapter, we summarize the contributions of our work and we discuss our perspectives.

Mobile applications are relatively new in the world of software engineering, they differ from traditional desktop applications by their composition and their development needs. In addition, they often rely on very short evolution cycles to satisfy demanding users and to adapt to their environments which are also constantly evolving. It is, therefore, necessary to provide developers of these applications with the appropriate knowledge and tools to enable them to provide high-quality applications. Thus, we presented in this work three contributions to the research in the field of mobile apps. These contributions provide actionable recommendations for researchers and tools makers who aim to propose solutions that limit code smells and improve the quality of mobile apps.

In chapter 2, we presented an empirical study that examines the impact of refactoring operations on code smells. This study covered the evolution history of five Android apps comprising 652 releases and exhibiting a total of 9,600 refactoring operation and 15 common Android smell types as well as 10 common traditional Object-Oriented (OO) code smell types. The results showed that code smells are very prevalent in Android apps, with 68% of classes being affected by Android specific smells and 63% of classes affected by traditional OO smells. Moreover, developers are more likely to apply refactoring operations to *non-smelly* code elements. A total of 25% of refactorings were applied to traditional OO smells, while 23% of refactorings were applied to Android smells. Finally, these refactoring activities removed only 5% of traditional OO smells and 1.5% of Android smells.

In Chapter 3, we presented an empirical study on a large dataset composed of 1,923 open source Android apps taking into account 15 types of Android-specific and 10 types of traditional OO code smells to explore the prevalence of code smell co-occurrences and determine which code smell types tend to co-exist more frequently. The findings of our study indicate that the co-occurrence phenomenon is quite prevalent in Android apps with 34%, 26% and 51% of classes are affected respectively by more than one Android, OO and both (i.e., OO and Android) smell types and there exist 14 smell pairs that frequently co-occur together.

In Chapter 4, we presented a large empirical study that analyses the evolution history of 300 open-source Android apps exhibiting a total of 42,181 refactoring operations. We investigate the impact of these refactoring operations on 10 common quality metrics using a causal inference method based on the Difference-in-Differences (DiD) model. Our results indicate that for most refactoring type and metric combinations, the refactoring produced no significant change in the metric. On the other hand, some refactoring types yielded a broad improvement in several metric values. The LCOM metric stood out as the least consistent metric, improving for some refactoring types and disimproving for others. For the non-refactoring commits, the metrics exhibit no significant change, other than (unsurprisingly) the design size metrics.

Perspectives:

In this section we present the perspectives of our work. We mentioned many perspectives throughout our presentation of the contributions and their implications. The objective of this section is to summarize these perspectives.

- We plan to extend our study to more open source and commercial Android apps to better generalize our results.
- We plan to analyze other types of code smells, quality metrics and refactorings.
- We plan to conduct a qualitative investigation through a survey with Android developers to better understand their intuition behind refactoring activities in the context of mobile apps.
- We plan to analyze other types of code smells and investigate the impact of code smell co-occurrences on internal and external quality attributes as well as other performance aspects.

- We plan to develop customized Android app refactoring tools based on the information about co-occurrences of code smells.

BIBLIOGRAPHY

Abbes, M., Khomh, F., Gueheneuc, Y.-G. & Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. *2011 15th European Conference on Software Maintenance and Reengineering*, pp. 181–190.

Agarwal, R. (2017). Decision making with association rule mining and clustering in supply chains. *International Journal of Data and Network Science*, 1(1), 11–18.

Agrawal, R., Imielinski, T. & Swami, A. (1993). Mining associations between sets of items in large databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 207–216.

Agrawal, R., Srikant, R. et al. (1994). Fast algorithms for mining association rules. *Proc. 20th int. conf. very large data bases, VLDB*, 1215, 487–499.

Al-Qutaish, R. E. (2010). Quality models in software engineering literature: an analytical and comparative study. *Journal of American Science*, 6(3), 166–175.

AlOmar, E. A., Mkaouer, M. W., Ouni, A. & Kessentini, M. (2019). On the impact of refactoring on the relationship between quality attributes and design metrics. *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–11.

AlOmar, E. A., Mkaouer, M. W. & Ouni, A. (2020a). Toward the Automatic Classification of Self-Affirmed Refactoring. *Journal of Systems and Software (JSS)*, 171.

AlOmar, E. A., Rodriguez, P. T., Bowman, J., Wang, T., Adepoju, B., Lopez, K., Newman, C., Ouni, A. & Mkaouer, M. W. (2020b). How Do Developers Refactor Code to Improve Code Reusability? *International Conference on Software and Software Reuse*, pp. 261–276.

Alshayeb, M. (2009). Empirical investigation of refactoring effect on software quality. *Information and software technology*, 51(9), 1319–1326.

AlZu'bi, S., Hawashin, B., EIBes, M. & Al-Ayyoub, M. (2018). A novel recommender system based on apriori algorithm for requirements engineering. *International conference on social networks analysis, management and security (snams)*, pp. 323–327.

Angrist, J. D. & Pischke, J.-S. (2008). *Mostly harmless econometrics: An empiricist's companion*. Princeton university press.

Aniche, M. (2016). CK-Metrics. Retrieved from: https://github.com/mauricioaniche/ck.

Arcoverde, R., Garcia, A. & Figueiredo, E. (2011). Understanding the longevity of code smells: preliminary results of an explanatory survey. *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 33–36.

Baqais, A. A. B. & Alshayeb, M. (2020). Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2), 459–502.

Basili, V. R., Briand, L. C. & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10), 751–761.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A. & Binkley, D. (2012). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. *IEEE International Conference on Software Maintenance (ICSM)*, pp. 56–65.

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R. & Palomba, F. (2015a). An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107, 1–14.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A. & Binkley, D. (2015b). Are test smells really harmful? An empirical study. *Empirical Software Engineering*, 20(4), 1052–1094.

Bessghaier, N., Ouni, A. & Mkaouer, M. W. (2020). On the Diffusion and Impact of Code Smells in Web Applications. *International Conference on Services Computing (SCC)*, pp. 1–15.

Brin, S., Motwani, R., Ullman, J. D. & Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pp. 255–264.

Brown, W. H., Malveau, R. C., McCormick, H. W. & Mowbray, T. J. (1998a). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Brown, W. J., Malveau, R. C., McCormick III, H. W. & Mowbray, T. J. (1998b). Refactoring Software, Architectures, and Projects in Crisis. Wiley.

Carette, A., Younes, M. A. A., Hecht, G., Moha, N. & Rouvoy, R. (2017). Investigating the energy impact of android smells. *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 115–126.

Cedrim, D., Sousa, L., Garcia, A. & Gheyi, R. (2016). Does refactoring improve software structural quality? A longitudinal study of 25 projects. *Proceedings of the 30th Brazilian Symposium on Software Engineering*, pp. 73–82.

Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M. & Chávez, A. (2017). Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. *Joint Meeting on Foundations of Software Engineering*, pp. 465–475.

Chatzigeorgiou, A. & Manakos, A. (2010). Investigating the evolution of bad smells in object-oriented code. *Seventh International Conference on the Quality of Information and Communications Technology*, pp. 106–115.

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D. & Garcia, A. (2017). How does refactoring affect internal quality attributes? A multi-project study. *31st Brazilian Symposium on Software Engineering*, pp. 74–83.

Chidamber, S. R. & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6), 476–493.

Cliff, N. (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3), 494.

Cramér, H. (1999). *Mathematical methods of statistics*. Princeton university press.

Cramir, H. (1946). Mathematical methods of statistics. *Princeton U. Press, Princeton*, 500.

D'Ambros, M., Bacchelli, A. & Lanza, M. (2010). On the impact of design flaws on software defects. *10th International Conference on Quality Software*, pp. 23–31.

Das, T., Di Penta, M. & Malavolta, I. (2016). A quantitative and qualitative investigation of performance-related commits in android apps. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 443–447.

dataset. (2020). Retrieved from: https://github.com/Oumaymahamdi/IST2020ReplicationPackage.

Dataset. (2021a). Retrieved from: https://github.com/stilab-ets/smell-co-occurences.

Dataset. (2021b). Retrieved from: https://github.com/stilab-ets/Android-refactoring.

Delchev, M. & Harun, M. F. (2015). Investigation of code smells in different software domains. *Full-scale Software Engineering*, 31.

Du Bois, B. & Mens, T. (2003). Describing the impact of refactoring on internal program quality. *International Workshop on Evolution of Large-scale Industrial Software Applications*, pp. 37–48.

Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L. & Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology*, 126, 106347.

Fontana, F. A. & Spinelli, S. (2011). Impact of refactoring on quality code evaluation. *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 37–40.

Fontana, F. A., Ferme, V. & Zanoni, M. (2015). Towards assessing software architecture quality by exploiting code smell relations. *International Workshop on Software Architecture and Metrics*, pp. 1–7.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, d. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Garg, A., Gupta, M., Bansal, G., Mishra, B. & Bajpai, V. (2016). Do bad smells follow some pattern? *International Congress on Information and Communication Technology*, pp. 39–46.

Geppert, B., Mockus, A. & Robler, F. (2005). Refactoring for Changeability: A way to go? *11th IEEE International Software Metrics Symposium*, pp. 10.

Gjoshevski, M. & Schweighofer, T. (2015). Small Scale Analysis of Source Code Quality with regard to Native Android Mobile Applications. *SQAMIA*, pp. 9–16.

Grano, G., Palomba, F., Di Nucci, D., De Lucia, A. & Gall, H. C. (2019). Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156, 312–327.

Gyimothy, T., Ferenc, R. & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10), 897–910.

Habchi, S., Moha, N. & Rouvoy, R. (2019a). The rise of android code smells: who is to blame? *International Conference on Mining Software Repositories (MSR)*, pp. 445–456.

Habchi, S., Rouvoy, R. & Moha, N. (2019b). On the survival of Android code smells in the wild. *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 87–98.

Habchi, S., Moha, N. & Rouvoy, R. (2020). Android Code Smells: From Introduction to Refactoring. *arXiv preprint arXiv:2010.07121*.

Hamdi, O., Ouni, A., AlOmar, E., Ó Cinnéide, M. & Mkaouer, M. W. (2021a). An Empirical Study on the Impact of Refactoring on Quality Metrics in Android Applications. *8th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2021)*, pp. 1–12.

Hamdi, O., Ouni, A., AlOmar, E. A., Cinnéide, M. O. & Mkaouer, M. W. (2021b). An Empirical Study on the Impact of Refactoring on Quality Metrics in Android Applications. *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pp. 28–39.

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

Hecht, G., Moha, N. & Rouvoy, R. (2016). An empirical study of the performance impacts of android code smells. *International conference on mobile software engineering and systems*, pp. 59–69.

Hegedűs, P., Kádár, I., Ferenc, R. & Gyimóthy, T. (2018). Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology*, 95, 313–327.

Impact of Extract Method refactoring, o.-N. (2021). Retrieved from: https://github.com/federicoiosue/Omni-Notes/commit/f8c9248023ce54885a7c06ad0df86bb3214e3269.

Joshi, B. (2016). Beginning SOLID Principles and Design Patterns for ASP.NET Developers. *Apress*.

Kádár, I., Hegedűs, P., Ferenc, R. & Gyimóthy, T. (2016). A manually validated code refactoring dataset and its assessment regarding software maintainability. *International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 1–4.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A. & Ubayashi, N. (2012). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6), 757–773.

Karimi-Majd, A.-M. & Mahootchi, M. (2015). A new data mining methodology for generating new service ideas. *Information Systems and e-Business Management*, 13(3), 421–443.

Karkhanis, S. P. & Dumbre, S. S. (2015). A study of application of data mining and analytics in education domain. *International Journal of Computer Applications*, 120(22).

Kataoka, Y., Imai, T., Andou, H. & Fukaya, T. (2002). A quantitative evaluation of maintainability enhancement by refactoring. *International Conference on Software Maintenance*, pp. 576–585.

Kaur, M. & Kang, S. (2016). Market Basket Analysis: Identify the changing trends of market data using association rule mining. *Procedia computer science*, 85, 78–85.

Kessentini, M. & Ouni, A. (2017). Detecting android smells using multi-objective genetic programming. *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 122–132.

Khomh, F., Di Penta, M. & Gueheneuc, Y.-G. (2009a). An exploratory study of the impact of code smells on software change-proneness. *2009 16th Working Conference on Reverse Engineering*, pp. 75–84.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G. & Sahraoui, H. (2009b). A bayesian approach for the detection of code and design smells. *2009 Ninth International Conference on Quality Software*, pp. 305–314.

Khomh, F., Di Penta, M., Guéhéneuc, Y.-G. & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3), 243–275.

Lanza, M. & Marinescu, R. (2007). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

Lehman, M. M. (1996). Laws of software evolution revisited. *European Workshop on Software Process Technology*, pp. 108–124.

Lieberherr, K. & Holland, I. (1989). Assuring good style for object-oriented programs. *IEEE Software*, 6(5), 38-48. doi: 10.1109/52.35588.

Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D. & Guéhéneuc, Y.-G. (2014). Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps. *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 232–243.

Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimóthy, T. & Chrisochoides, N. (2009). Modeling class cohesion as mixtures of latent topics. *IEEE International Conference on Software Maintenance*, pp. 233–242.

Lozano, A., Wermelinger, M. & Nuseibeh, B. (2007). Assessing the impact of bad smells using historical information. *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pp. 31–34.

Malavolta, I., Verdecchia, R., Filipovic, B., Bruntink, M. & Lago, P. (2018). How maintainability issues of android apps evolve. *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 334–344.

Mannan, U. A., Ahmed, I., Almurshed, R. A. M., Dig, D. & Jensen, C. (2016). Understanding code smells in android applications. *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 225–236.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 350–359.

Martin, W., Sarro, F., Jia, Y., Zhang, Y. & Harman, M. (2016). A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9), 817–847.

Martins, J., Bezerra, C., Uchôa, A. & Garcia, A. (2020). Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. *34th SBES*, 1–10.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4), 308–320.

Minelli, R. & Lanza, M. (2013). *Software analytics for mobile applications-insights and lessons learned*. 17th European Conference on Software Maintenance and Reengineering.

Minelli, R. (2012). *Software analytics for mobile applications*. Roberto Minelli.

Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K. & Ouni, A. (2015). Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3), 1–45.

Moha, N., Gueheneuc, Y.-G., Duchien, L. & Le Meur, A.-F. (2009). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1), 20–36.

Morales, R., Saborido, R., Khomh, F., Chicano, F. & Antoniol, G. (2016). Anti-patterns and the energy efficiency of Android applications. *arXiv preprint arXiv:1610.05711*.

Morales, R., Saborido, R., Khomh, F., Chicano, F. & Antoniol, G. (2017). Earmo: An energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*, 44(12), 1176–1206.

Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A. & Succi, G. (2007). A case study on the impact of refactoring on quality and productivity in an agile team. *IFIP Central and East European Conference on Software Engineering Techniques*, pp. 252–266.

Munaiah, N., Kroh, S., Cabrey, C. & Nagappan, M. (2017). Curating GitHub for engineered software projects. *Empirical Software Engineering*, 22(6), 3219–3253.

Munro, M. J. (2005). Product metrics for automatic identification of" bad smell" design problems in java source-code. *11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 15–15.

Murphy-Hill, E., Parnin, C. & Black, A. P. (2011). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1), 5–18.

Muse, B. A., Rahman, M. M., Nagy, C., Cleve, A., Khomh, F. & Antoniol, G. (2020). On the Prevalence, Impact, and Evolution of SQL code smells in Data-Intensive Systems. *International Conference on Mining Software Repositories (MSR)*.

Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S. & Hemati Moghadam, I. (2012). Experimental assessment of software metrics using automated refactoring. *ACM-IEEE International symposium on Empirical software engineering and measurement*, pp. 49–58.

Okike, E. (2010). A proposal for normalized lack of cohesion in method (LCOM) metric using field experiment. *IJCSI International Journal of Computer Science Issues*, 7(4), 19–26.

Olbrich, S., Cruzes, D. S., Basili, V. & Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. *3rd international symposium on empirical software engineering and measurement*, pp. 390–400.

Opdyke, W. F. (1992). *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. (Ph.D. thesis, University of Illinois at Urbana-Champaign).

Ouni, A., Kessentini, M., Sahraoui, H. & Boukadoum, M. (2013). Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1), 47–79.

Ouni, A., Kessentini, M., Bechikh, S. & Sahraoui, H. (2015a). Prioritizing code-smells correction tasks using chemical reaction optimization. *Software Quality Journal*, 23(2), 323–361.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K. & Hamdi, M. S. (2015b). Improving multi-objective code-smells correction using development history. *Journal of Systems and Software*, 105, 18–39.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K. & Deb, K. (2016). Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3), 1–53.

Paixao, M., Harman, M., Zhang, Y. & Yu, Y. (2017). An empirical study of cohesion and coupling: Balancing optimization and disruption. *IEEE Transactions on Evolutionary Computation*, 22(3), 394–414.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R. & De Lucia, A. (2014a). Do they really smell bad? a study on developers' perception of bad code smells. *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 101–110.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D. & De Lucia, A. (2014b). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5), 462–489.

Palomba, F., Di Nucci, D., Panichella, A., Oliveto, R. & De Lucia, A. (2016a). On the diffusion of test smells in automatically generated test code: An empirical study. *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, pp. 5–14.

Palomba, F., Panichella, A., De Lucia, A., Oliveto, R. & Zaidman, A. (2016b). A textual-based technique for smell detection. *2016 IEEE 24th international conference on program comprehension (ICPC)*, pp. 1–10.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A. & De Lucia, A. (2017a). Lightweight detection of Android-specific code smells: The aDoctor project. *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 487–491.

Palomba, F., Oliveto, R. & De Lucia, A. (2017b). Investigating code smell co-occurrences using association rule learning: A replicated study. *IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pp. 8–13.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R. & De Lucia, A. (2018a). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3), 1188–1221.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R. & De Lucia, A. (2018b). A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology*, 99, 1–10.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A. & De Lucia, A. (2019). On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, 105, 43–55.

Pantiuchina, J., Lanza, M. & Bavota, G. (2018). Improving Code: The (Mis) perception of Quality Metrics. *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 80–91.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A. & Palomba, F. (2019). On the distribution of test smells in open source Android applications: an exploratory study. *29th Annual International Conference on Computer Science and Software Engineering (CASCON)*, pp. 193–202.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A. & Palomba, F. (2020). tsDetect: an open source test smells detection tool. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1650–1654.

Peters, R. & Zaidman, A. (2012). Evaluating the lifespan of code smells using software repository mining. *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 411–416.

Piatetsky-Shapiro, G. (1991). Discovery, analysis, and presentation of strong rules. *Knowledge discovery in databases*, 229–238.

Prete, K., Rachatasumrit, N., Sudan, N. & Kim, M. (2010). Template-based reconstruction of complex refactorings. *IEEE International Conference on Software Maintenance*, pp. 1–10.

Rapu, D., Ducasse, S., Gîrba, T. & Marinescu, R. (2004). Using history information to improve design flaws detection. *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pp. 223–232.

Reimann, J., Brylski, M. & Aßmann, U. (2014). A tool-supported quality smell catalogue for android developers. *Conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*.

Research, I. D. C. I. (2020). Android dominating mobile market. Retrieved from: https://www.idc.com/promo/smartphone-market-share/os.

Rice, J. A. (2006). *Mathematical statistics and data analysis*. Cengage Learning.

Rubin, J., Henniche, A. N., Moha, N., Bouguessa, M. & Bousbia, N. (2019). Sniffing android code smells: an association rules mining-based approach. *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 123–127.

Ruiz, I. J. M., Nagappan, M., Adams, B. & Hassan, A. E. (2012). Understanding reuse in the android market. *IEEE International Conference on Program Comprehension (ICPC)*, pp. 113–122.

Saika, T., Choi, E., Yoshida, N., Haruna, S. & Inoue, K. (2016). Do developers focus on severe code smells? *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 4, 1–3.

Santos, J. A. M., Rocha-Junior, J. B., Prates, L. C. L., do Nascimento, R. S., Freitas, M. F. & de Mendonça, M. G. (2018). A systematic review on the code smell effect. *Journal of Systems and Software*, 144, 450–477.

Sharma, T. (2012). Quantifying quality of software design to measure the impact of refactoring. *IEEE 36th Annual Computer Software and Applications Conference Workshops*, pp. 266–271.

Sharma, T., Suryanarayana, G. & Samarthyam, G. (2015). Challenges to and solutions for refactoring adoption: An industrial perspective. *IEEE Software*, 32(6), 44–51.

Sharma, T., Singh, P. & Spinellis, D. (2020). An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering*, 25(5), 4020–4068.

Silva, D. & Valente, M. T. (2017). Refdiff: detecting refactorings in version histories. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 269–279.

Silva, D., Tsantalis, N. & Valente, M. T. (2016). Why we refactor? confessions of github contributors. *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 858–870.

Simon, F., Steinbruckner, F. & Lewerentz, C. (2001). Metrics based refactoring. *Proceedings fifth european conference on software maintenance and reengineering*, pp. 30–38.

Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A. & Dybå, T. (2012). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8), 1144–1156.

Soares, G., Gheyi, R., Murphy-Hill, E. & Johnson, B. (2013). Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86(4), 1006–1022.

Sons, J. W. . (1999). Patterns in Java. *Wiley*, 2. Chapter 4 - GRASP Patterns.

Stevens, W. P., Myers, G. J. & Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, 13(2), 115–139.

Stroggylos, K. & Spinellis, D. (2007). Refactoring–does it improve software quality? *International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, pp. 10–10.

Szóke, G., Antal, G., Nagy, C., Ferenc, R. & Gyimóthy, T. (2014). Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 95–104.

Tahir, A., Dietrich, J., Counsell, S., Licorish, S. & Yamashita, A. (2020). A large scale study on how developers discuss code smells and anti-pattern in Stack Exchange sites. *Information and Software Technology*, 106333.

Tahvildari, L. & Kontogiannis, K. (2003). A metric-based approach to enhance design quality through meta-pattern transformations. *European Conference on Software Maintenance and Reengineering*, pp. 183–192.

Tan, L. & Bockisch, C. (2019). A Survey of Refactoring Detection Tools. *Software Engineering (Workshops)*, pp. 100–105.

Tavares, C., Bigonha, M. & Figueiredo, E. (2020). Analyzing the Impact of Refactoring on Bad Smells. *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pp. 97–101.

Thongtanunam, P., Shang, W. & Hassan, A. E. (2019). Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones. *Empirical Software Engineering*, 24(2), 937–972.

Tsantalis, N. & Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3), 347–367.

Tsantalis, N., Guana, V., Stroulia, E. & Hindle, A. (2013). A multidimensional empirical study on refactoring activity. *CASCON*, pp. 132–146.

Tsantalis, N., Mansouri, M., Eshkevari, L., Mazinanian, D. & Dig, D. (2018a). Accurate and efficient refactoring detection in commit history. *International Conference on Software Engineering (ICSE)*, pp. 483–494.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D. & Dig, D. (2018b). Accurate and Efficient Refactoring Detection in Commit History. *International Conference on Software Engineering*, pp. 483–494.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A. & Poshyvanyk, D. (2015). When and why your code starts to smell bad. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 1, 403–414.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A. & Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. *IEEE/ACM International Conference on Automated Software Engineering*, pp. 4–15.

Uchôa, A., Barbosa, C., Oizumi, W., Blenilio, P., Lima, R., Garcia, A. & Bezerra, C. (2020). How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 511–522.

Vaucher, S., Khomh, F., Moha, N. & Guéhéneuc, Y.-G. (2009). Tracking design smells: Lessons from a study of god classes. *Working Conference on Reverse Engineering*, pp. 145–154.

Vidal, S. A., Marcos, C. & Díaz-Pace, J. A. (2016). An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23(3), 501–532.

Wang, J., Li, H., Huang, J. & Su, C. (2016). Association rules mining based analysis of consequential alarm sequences in chemical processes. *Journal of Loss Prevention in the Process Industries*, 41, 178–185.

Wilcoxon, F., Katti, S. & Wilcox, R. A. (1970). Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Selected tables in mathematical statistics*, 1, 171–259.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. & Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Xing, Z. & Stroulia, E. (2005). UMLDiff: an algorithm for object-oriented design differencing. *IEEE/ACM international Conference on Automated software engineering*, pp. 54–65.

Xu, L., Wu, S. F. & Chen, H. (2013). *Techniques and tools for analyzing and understanding android applications*. University of California, Davis.

Yamashita, A. & Moonen, L. (2013a). Do developers care about code smells? An exploratory survey. *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 242–251.

Yamashita, A. & Moonen, L. (2013b). Exploring the impact of inter-smell relations on software maintainability: An empirical study. *International Conference on Software Engineering*, pp. 682–691.

Yamashita, A., Zanoni, M., Fontana, F. A. & Walter, B. (2015). Inter-smell relations in industrial and open source systems: A replication and comparative analysis. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 121–130.

Yoshida, N., Saika, T., Choi, E., Ouni, A. & Inoue, K. (2016). Revisiting the relationship between code smells and refactoring. *International Conference on Program Comprehension (ICPC)*, pp. 1–4.

Zhang, M., Baddoo, N., Wernick, P. & Hall, T. (2011). Prioritising refactoring using code bad smells. *IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 458–464.

Zhang, Z. & Cai, H. (2019). A look into developer intentions for app compatibility in Android. *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 40–44.