# On the Co-occurrences of Code Smells in Android Applications

***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.**

***Index Terms*—Android, code smells, Android smells, Association rule mining, empirical study**

## I. 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* [1]–[3]. The presence of code smells often hinders the maintenance and evolution of any software system [4]–[8]. Like any software system, Android apps can be affected by traditional Object-Oriented (OO) code smells [1], [2], but also with new categories of emerging Android-specific smells, known as *Android smells* [3], [9]–[12]. The presence of these smells can lead to resource leaks (*e.g.*, CPU, memory, battery, etc.) causing, therefore, several performance and usability problems [11]–[14].

Most of the existing studies focused on traditional OO smells [7], [15]–[21]. In particular, they focused on various aspects of OO code smells including code smells prevalence [11], [22]–[24], co-occurrences [15]–[18], [21], [25] as well as the effects of code smells on software quality and maintainability [21], [26], [27]. 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 [7], [15]–[21]. Furthermore, few studies have recently examined individual instances of code smells in Android apps [4], [11], [12], [14], [24].

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 [3], [4], [10], [11], 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 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. 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 empirical 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 [28] 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 [7], [11], [15]. However, for Android smells, our findings reveal various interesting relationships in the context of Android apps development.
- Some smells that were not involved in co-occurrences (*e.g.*, *Lazy Class*, *Leaking Thread* and *Internal Getter And Setter*) are surprisingly frequent comparing to other smells that were involved (*e.g.*, *Complex Class*, *No Low Memory Resolver*). Therefore, the observed co-occurrences are not be just the result of the high diffuseness of single code smell types.

The paper is structured as follows. Section II presents the design of our empirical study, while Section III presents and discusses our results. Section IV describes the threats to validity of our study while Section V reports the implications of our study. In Section VI, we review the related works. Finally, Section VII draws our conclusions and future works.

## II. 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.

### A. 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.

***RQ3.*** *How prevalent are code smells?*
This question represents an investigation into the distribution of code smells in our dataset. The aim consists of investigating if the co-occurrences found in RQ2 are just the result of the prevalence of some code smell types.

### B. 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. [3]. 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 [4], [9], [11], [29]. We also considered 10 common traditional OO smells defined by Fowler [2] and Brown et al. [1] that have been widely studied in prior works [6], [7], [15]. 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 I and II report the set of OO and Android smells, respectively, that are investigated in our study.

### C. Data Extraction

Figure 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 [30], [31]:

- 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[1], a command-line based tool that implements rules provided by Palomba et al. [10] to identify common Android smells. We selected this tool as it achieves a high detection precision of 98%, and recall of

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

TABLE I: List of traditional Oriented-Object code smells [6], [15].

| 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. |
| RB | 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 II: List of Android-specific code smells [3], [10].

| 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`) [3]. |
| 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 `setInexactRepeating`. |
| 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. |

TABLE III: 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 |

98%, as reported in Palomba et al. [10]. As for the traditional OO smells, we used an existing tool[2], that has been widely used in recent studies [6], [7], [15], [32]. The tool detects 10 common types of OO smells and implements simple detection rules published by Bavota et al. [6] 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 III summarizes the statistics about the collected dataset.

## D. Data Analysis

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

*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.

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

*2) Analysis method for RQ2:* To answer *RQ2*, we employ association rule mining (also known as market basket analysis) using the *Apriori* algorithm [28]. 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 $\Rightarrow$ 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 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 [28], confidence [28] and lift [33] 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 $S_j$.

$$Support(S_i \Rightarrow S_j) = P(S_i \cup S_j) \tag{1}$$

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

$$Confidence(S_i \Rightarrow S_j) = P(S_i \cup S_j)/P(S_i) \tag{2}$$

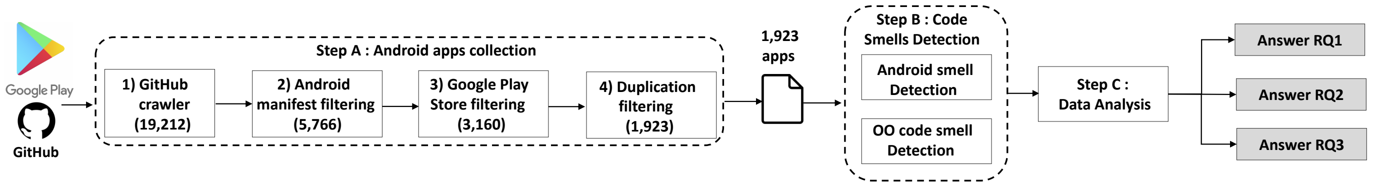
Google Play
GitHub
Step A : Android apps collection
1) GitHub crawler (19,212)
2) Android manifest filtering (5,766)
3) Google Play Store filtering (3,160)
4) Duplication filtering (1,923)
1,923 apps
Step B : Code Smells Detection
Android smell Detection
OO code smell Detection
Step C : Data Analysis
Answer RQ1
Answer RQ2
Answer RQ3

Fig. 1: Overall process to conduct our empirical study.

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

$$Lift(S_i \Rightarrow S_j) = P(S_i \cup S_j)/(P(S_i) \times P(B)) \quad (3)$$

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 [34] 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 4:

$$V = \sqrt{\frac{\chi^2}{n \times min(row - 1, col - 1)}} \quad (4)$$

*3) Analysis method for RQ3:* To answer *RQ3*, we use the collected Android and traditional OO smells for each application as described in *Step B*. Specifically, for each code smell type, we computed the diffusion of each smell type, $S_i$ which corresponds to the ratio of the number of classes infected by the smell $S_i$ over the total number of classes.

### *E. Replication package*

Our dataset is available in our replication package for future replications and extensions [35].

## III. Empirical Study results

### *A. RQ1: To what extent code smells co-occur in Android apps?*

Table IV 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 IV.

When combining both Android and OO smells, we observe from Table IV 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[3] app, version dev-20201223[4]. 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 [5] 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 [10]. 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 [2], [13], [32], [36], [37].

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.

[3]https://github.com/nextcloud/android
[4]https://github.com/nextcloud/android/releases/tag/dev-20201223
[5]https://scitools.com/

TABLE IV: 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% | 5,825 | 30% | 5,696 | 30% |
| Two smells | 1,644 | 9% | 3,663 | 19% | 3,526 | 18% |
| Three smells | 1,477 | 8% | 1,893 | 10% | 2,278 | 12% |
| Four smells | 788 | 4% | 787 | 4% | 1,429 | 7% |
| Five smells | 629 | 3% | 202 | 1% | 1,049 | 5% |
| Six smells | 290 | 2% | 37 | <1% | 716 | 4% |
| Seven Smells | 51 | <1% | 7 | <1% | 525 | 3% |
| Eight smells | 2 | <1% | 1 | <1% | 334 | 2% |
| Nine smells | 0 | 0% | 0 | 0% | 209 | 1% |
| Ten smells | 0 | 0% | 0 | 0% | 112 | 1% |
| Eleven smells | 0 | 0% | 0 | 0% | 43 | <1% |
| Twelve smells | 0 | 0% | 0 | 0% | 18 | <1% |

**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.*

## B. 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 II-D. 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 VI 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 V 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 *confidence*$(S_i \Rightarrow S_j)$ = 0.5, while *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.

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 VI, 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[6]. 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()` 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 VI. 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

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

TABLE V: 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 | × | × |

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 VI, 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 [11], [12], [29]. Moreover, as shown in Table VI 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 [7], [11], [15]. However, for Android smells, our findings reveal various interesting relationships that have not been yet explored previously in the context of Android apps development.

**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).*

## C. RQ3: How prevalent are code smells?

Figure 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 83% of all classes are affected by code smells. This number reflects the widespread of code smells in Android apps. However, it is worth noting that not all code smells are frequently diffused. Indeed, the figure shows

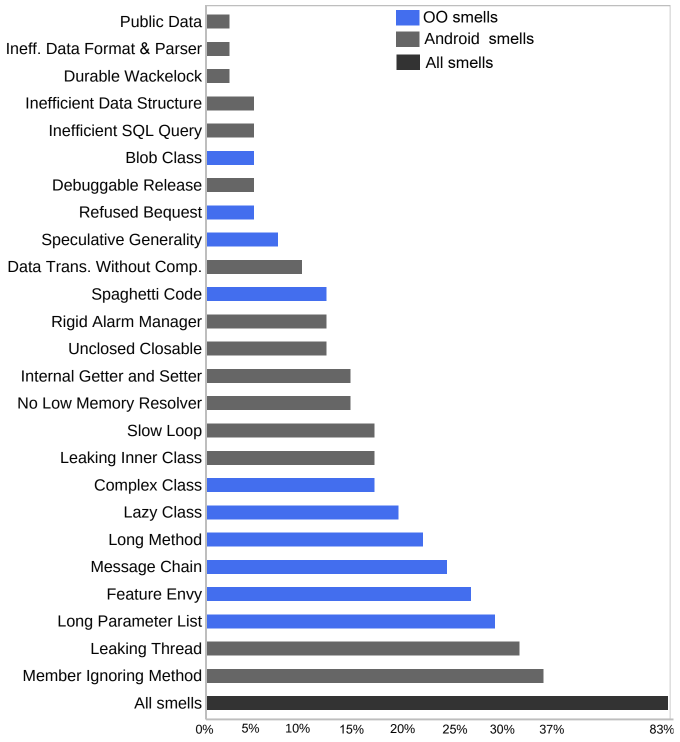

Fig. 2: The diffusion of each code smell type across the studied applications.

a significant disparity between the different code smell types. Overall, we can observe three main levels of diffusion, (*i*) frequently, (*ii*) moderately and (*iii*) low diffused smells.

- The most frequently diffused smells are the *Member Ignoring Method*, *Leaking Thread*, *Long Parameter List* and *Feature Envy* affecting each on average from 21% to 37% of classes.
- The moderately diffused smells include the *Message Chain*, *Long Method*, *Lazy Class*, *Complex Class*, *Internal Getter And Setter*, *No low Memory Resolver*, *Leaking Inner Class* and *Slow Loop* affecting each on average from from 11% to 19% of the classes.
- The rest of smells are the least diffused ones affecting each less than 10% of the classes.

As a consequence, these results highlight two interesting observations: (1) the earlier cited smells which are the the most diffused in our dataset are the ones that appear more frequently in Android apps, and (2) the frequent associations between the considered smells indicated in Table IV 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.

TABLE VI: 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 RQ3.** *Overall, code smells are not diffused equally: the Member Ignoring Method, Leaking Thread, Long Parameter List and Feature Envy are the most diffused, each affecting from 21% to37% of classes. Furthermore, not all these smells constitute co-occurrences which indicate the inferred co-occurences are not necessary the result of the high diffuseness of some single smell types.*

## IV. Threats to validity

This section discusses threats to validity of the study.

*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% [6], [10], [11]. For OO smells, we used organic which is an implementations of rules published by Bavota et al. [6]. However, we cannot exclude that some code smells were missed by our analysis or false positives were considered.

*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.

*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.

*Threats to reliability validity* reflect whether the study has been conducted and reported in a way that other researchers and practitioners can replicate it and reach the same results. To mitigate any reliability threats, we report all steps followed to obtain the dataset for the investigated research questions and provide links and/or references to the employed tools. Moreover, the employed dataset along with the variable values for the statistical analysis is publicly available in our replication package [35].

## V. 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 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.
- **Detecting and prioritizing code smells.** The main goal of our empirical study is to determine which code smells co-occur together in Android apps, with the purpose of exploring the relationships between them. Such knowledge can help in improving code smell detection and prioritization by researchers to design build recommendation systems based on the interaction between code smells. Furthermore, prioritizing the detection and the removal could be beneficial for practitioners based on the most (1) frequent and (2) harmful co-occurrences of code smells. Such techniques can also notify developers on the consequences of other smells that can co-occur with the detected ones. For example, as depicted in Table VI, a class affected by the *Feature Envy* or *Message Chain* smell can be at a risk of becoming a *Complex Class* in the future. Such techniques can be integrated in conjunction with refactoring tools in modern development environments such as Eclipse and IntellJ IDE to recommend appropriate refactorings.
- **Predicting future introduction of code smells.** The use of co-occurrence information can provide an important source of information to predict the potential appearance

of the same or another code smell in the future. Such prediction can help in preventing these anomalies, and serve as a warning to developers to increase their awareness before these smells appear in their software systems. For instance, our second research question shows that *Long Method* and *Long Parameter List* tend to co-occur together. The use of such information suggests that developers should be careful about the emergence of these co-occurrences in code base.

- **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 [15], [20], [25], [38] 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 [39]. As our study indicates that developers can be often confronted with the phenomenon of code smell co-occurrences in their codebase, 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.

## VI. Related work

A number of studies have focused on detecting bad code smells defined by Fowler et al. [2] and fixing them via the application of refactoring. The code smell phenomenon has been broadly investigated from different angles. Several research works devised approaches and tools to automatically detect code smells [10], [40]–[44], while other research efforts focused on analyzing their relationship [7], [17]–[19], [39], [45], [46], evolution [4], [47]–[50], diffusion [11], [22]–[24], [51] and their impact on software quality attributes [21], [26], [27]. In this section, we review the literature studying the interactions between code smells and their effect, along with the current state-of-the-art studies related to code smells and refactoring in mobile applications.

### A. Code smells and their impact on quality

Palomba et al. [15] 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. [17] 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. [16] 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. [19] 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. [21] 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. [18] 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 [25] 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

### B. Code smells and refactoring in mobile apps

Code smells have also been studied in the context of mobile applications. Linares-Vásquez et al. [52] used DECOR to detect object-oriented code smells through an analysis of 1,343 mobile apps. Their study show that code smells has a negative impact on the fault-pronounce of apps. Tufano et al [53] investigated the appearance of code smells in the code. The authors studied the evolution of 200 open source projects in which 70 of them were Android apps, finding that, the code smell was most likely introduced when the file was created. Mannan et al. [24] conducted a large-scale study of 500 apps in order to compare the distribution of code smells in mobile and desktop apps. Their findings show that Android and desktop apps are similar in terms of the detected code smells using the tool InFusion. In another study that focused on the energy efficiency of mobile apps, Morales et al. [54] analyzed 59 apps and detected 8 code smells. The authors found that number of refactorings have a positive effect on the energy efficiency of mobile apps, while applying other type of refactorings have a negative impact. In follow-up work, Morales et al. [14] proposed an energy-aware refactoring tool, named EARMD, that takes into account the energy consumption when refactoring code smells defined by Fowler et al. [2]. In the same context, Palomba et al. [11] performed an analysis of 9 Android-specific code smells instead of traditional code

TABLE VII: A summary of the literature on the impact of refactoring activities on Android/Mobile apps.

| Study | Year | Focus | # of Android/Mobile Apps | Traditional / Android smell | Smell Detection Tool |
|---|---|---|---|---|---|
| Linares-Vásquez et al. [52] | 2014 | Antipatterns & quality metrics | 1,343 | Yes / No | DECOR |
| Tufano et al. [53] | 2015 | Code smell & their introduction | 70 | Yes / No | DECOR |
| Hecht et al. [36] | 2015 | Tracking the quality of Android apps | 106 | Yes / Yes | PAPRIKA |
| Hecht et al. [12] | 2016 | Performance impacts on Android smell | 2 | No / Yes | PAPRIKA |
| Mannan et al. [24] | 2016 | Code smell in Android vs Desktop apps | 500 | Yes / No | inFusion |
| Morales et al. [54] | 2016 | Antipatterns & energy efficiency | 59 | Yes / Yes | ReCon |
| Carette et al. [29] | 2017 | Code smell & energy consumption | 5 | No / Yes | HOT-PEPPER |
| Morales et al. [14] | 2018 | Impact of antipatterns on Andriod apps | 20 | Yes / No | EARMO |
| Palomba et al. [11] | 2019 | Code smell & energy consumption | 60 | No / Yes | ADOCTOR |

smells in order to understand the impact of these smells on energy efficiency along with understanding role of refactoring on improving the performance of mobile apps. Their main findings show that refactoring is considered to be a powerful technique to reduce the energy consumption of methods. In another studies that have been carried out by Hecht et al. [12], [36], a code smell detection tool called PARPIKA, was proposed to detect smells by analysing the bytecode of Android apps. They found a positive correlation between performance improvement in terms of delayed frame and CPU usage and the smells considered in their study. Carette et al. [29] designed a tool on top of PARPIKA to automatically correct the smells after detecting them. Particularly, they measured the energy consumption before and after the removal of the smells. Recently, Habchi et al. [13] conduct a study that covers eight Android code smells from 324 Android apps having 255k commits, and contributions from 4,525 developers to investigate if the main reason behind the accrual of code smells is developers' ignorance and oversight. Their results revealed important research findings. Notably, they showed that pragmatism, prioritization, and individual attitudes are not relevant factors for the accrual of mobile code smells. The problem is rather caused by ignorance and oversight, which are prevalent among mobile developers. Furthermore, they highlighted several flaws in the code smell definitions currently adopted by the research community. For the sake of clarity, we summarize these state-of-the-art studies in Table VII.

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.

## VII. Conclusion and Future Work

The emergence of multiple code smells in the same code element can have a significant impact on the system understandability and reduce the ability of developers to maintain a software project, as indicated in several previous works [21], [25], [55]. To further understand this phenomenon, we investigated, in the paper, 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, (2) code smell pairs that most tend to co-occur, and (3) the prevalence of individual code smells to assess whether or not the co-occurrence are just the result of high diffusion of a specific smell type. The key 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.
- there exist 14 smell pairs that frequently co-occur together.
- The observed co-occurrences are not be just the result of the high diffuseness of single code smell types since we found 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 in smell pairs (*e.g.*, Complex Class, No Low Memory Resolver).

Our results have several actionable insights advocating the necessity of awareness mechanisms for Android developers to identify and prevent code smell occurrences in their code bases. The gained knowledge from the phenomenon of code smell co-occurrences can be used as a basis to further support the detection, prioritization, prediction and correction of code smells in the context of mobiles apps.

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.

## References

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

[2] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[3] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," in *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*, vol. 2014, 2014.

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

[5] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *Joint Meeting on Foundations of Soft. Engineering*, 2017, pp. 465–475.

[6] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

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

[8] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[9] M. Kessentini and A. Ouni, "Detecting android smells using multi-objective genetic programming," in *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 122–132.

[10] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *IEEE 24th international conference on software analysis, evolution and reengineering*, 2017, pp. 487–491.

[11] ——, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, vol. 105, pp. 43–55, 2019.

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

[13] S. Habchi, N. Moha, and R. Rouvoy, "The rise of android code smells: Who is to blame?" in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 445–456.

[14] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: An energy-aware refactoring approach for mobile apps," *IEEE Trans. on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, 2017.

[15] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Inf. and Soft. Technology*, vol. 99, pp. 1–10, 2018.

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

[17] A. Garg, M. Gupta, G. Bansal, B. Mishra, and V. Bajpai, "Do bad smells follow some pattern?" in *International Congress on Information and Communication Technology*. Springer, 2016, pp. 39–46.

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

[19] B. A. Muse, M. M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the prevalence, impact, and evolution of sql code smells in data-intensive systems," in *International Conference on Mining Software Repositories*, 2020, pp. 327–338.

[20] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, "Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study," *34th SBES*, pp. 1–10, 2020.

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

[22] M. Delchev and M. F. Harun, "Investigation of code smells in different software domains," *Full-scale Software Engineering*, vol. 31, 2015.

[23] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: an exploratory study." in *CASCON*, 2019, pp. 193–202.

[24] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Int. Conference on Mobile Software Engineering and Systems*, 2016, pp. 225–236.

[25] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *International Conference on Software Engineering*, 2013, pp. 682–691.

[26] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *10th International Conference on Quality Software*, 2010, pp. 23–31.

[27] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Soft. Eng.*, vol. 17, no. 3, pp. 243–275, 2012.

[28] R. Agrawal, T. Imielinski, and A. Swami, "Mining associations between sets of items in large databases," in *International Conference on Management of Data*, 1993, pp. 207–216.

[29] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *Int. Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 115–126.

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

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

[32] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[33] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *International conference on Management of data*, 1997, pp. 255–264.

[34] H. Cramér, *Mathematical methods of statistics*. Princeton university press, 1999, vol. 43.

[35] Dataset:, Link hidden for double blinded review, 2021.

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

[37] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

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

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

[40] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 305–314.

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

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

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

[44] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[45] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[46] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 4–15.

[47] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 106–115.

[48] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 411–416.

[49] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *International symposium on empirical software engineering and measurement*, 2009, pp. 390–400.

[50] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *Working Conference on Reverse Engineering*, 2009, pp. 145–154.

[51] N. Bessghaier, A. Ouni, and M. W. Mkaouer, "On the diffusion and impact of code smells in web applications," in *International Conference on Services Computing*. Springer, 2020, pp. 67–84.

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

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

[54] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Anti-patterns and the energy efficiency of android applications," *arXiv preprint arXiv:1610.05711*, 2016.

[55] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2012.