Noname manuscript No.
(will be inserted by the editor)

An Empirical Catalog of Code Smells for the Presentation Layer of Android Apps

Suelen Goularte Carvalho · Maurício Aniche · Júlio Veríssimo · Rafael S. Durelli · Marco Aurélio Gerosa

Received: date / Accepted: date

Abstract Software developers, including those for the Android mobile platform, constantly seek to improve their applications' maintainability and evolvability. Code smells are commonly used for this purpose, as they indicate symptoms of design problems. However, although the literature presents a variety of code smells, such as God Class and Long Method, characteristics specific to the underlying technologies are not taken into account. The presentation layer of an Android app, for example, implements specific architectural decisions from the Android platform itself (such as the use of Activities, Fragments, and Listeners) as well as deal with and integrate different types of resources (such as layouts and images). Through a three-step study involving 246 Android developers, we investigated code smells that developers perceive for this part of Android apps. We devised 20 specific code smells and collected the developers' perceptions of their frequency and importance. We also implemented a tool that identifies the proposed code smells and studied their prevalence in 619 open-source Android apps. Our findings suggest that: 1) developers perceive smells specific to the presentation layer of Android apps; 2) develop-

Suelen Gourlart Carvalho
University of São Paulo
E-mail: suelengcarvalho@gmail.com

Maurício Aniche
Delft University of Technology
E-mail: m.f.aniche@tudelft.nl

Júlio Veríssimo
Federal University of Lavras
E-mail: julio.santos@posgrad.ufla.br

Rafael S. Durelli
Federal University of Lavras
E-mail: rafael.durelli@ufla.br

Marco Aurélio Gerosa
Northern Arizona University
E-mail: marco.gerosa@nau.edu

ers consider these smells to be of high importance and frequency; and 3) the proposed smells occur in real-world Android apps. Our domain-specific smells can be leveraged by developers, researchers, and tool developers for searching potentially problematic pieces of code.

1 Introduction

"We are aware that good code matters, because we have had to deal with the lack of it for a long time," argues Robert Martin [1]. However, how do we find potentially problematic pieces of code? One answer might be by searching for smells. Code smells are anomalies that indicate a potential violation of design principles [2]. By looking for code smells, developers find problematic code that can be refactored to improve software quality [3].

Several code smells have been catalogued in the literature [1–4], e.g., Long Methods and God Classes. These code smells are usually defined based on traditional concepts and technologies that emerged during the 1970s and 1990s, such as object orientation and Java. In this paper, we call these "traditional code smells." However, in the last decade, new technologies have emerged, raising questions such as "Do traditional code smells apply to new technologies?" and "Are there code smells which are specific to new technologies?" [5]. Some researchers have already proposed technology-specific code smells for CSS [6], JavaScript [7], MVC [8, 9], and spreadsheets [10], for example.

Android [11], a mobile platform launched in 2008 by Google, has also attracted the attention of researchers. Some scholars have investigated the existence of traditional code smells in Android applications [12–14]. Others have studied Android-specific code smells related to efficiency (i.e., proper use of features like memory and processing) and usability (i.e., software capability to be understood) [15, 16]. Other researchers have focused on understanding Android development features that set them apart from traditional software development [17]. However, to the best of our knowledge, no study has focused on the Android presentation layer, which follows specific concepts and models. In this paper, we investigate the existence of code smells related to the maintainability of the presentation layer of an Android application.

To understand what developers consider code smells, we collected data employing two questionnaires. In the first questionnaire (n=45), we asked developers about good and bad practices they notice in the development of the Android presentation layer. From the responses, we derived 20 code smells. We then conducted a confirmatory questionnaire (n=201) investigating the frequency and importance of the 20 proposed code smells. We also implemented a tool to assist in the identification of the code smells, and measured their prevalence in 619 open-source apps from the F-Droid repository.

Therefore, the main contribution of this paper is the cataloguing and validation of 20 new code smells related to the maintainability of eight types of components and resources of the Android's presentation layer: *Activities*,

Fragments, *Adapters*, and *Listeners* (components), *Layouts*, *Styles*, *String*, and *Drawables* (resources).

2 Background: Android and its presentation layer

Android is a Linux-based mobile development platform launched in 2008 by Google in partnership with several companies [11, 18]. In early 2011, Android became the leading mobile platform, having reached more than 87% market share in 2017. While its main competitor, iOS, is only used by Apple's products, totaling approximately 30 different models [19], Android is used by more than 24,000 different models of mobile devices according to a survey conducted in 2015 [20]. In terms of software development, the wide variety of hardware configurations brings significant challenges: from performance-related issues to issues related to user interface development, screens, and resolutions.

This research focuses on analyzing elements related to the presentation layer of Android apps. We reviewed the official Android documentation for the presentation layer [21], from which we identified the following components: Activities, Fragments, Adapters, and Listeners.

- ***Activities*** represent a screen in the app, which the end-user sees and interacts with.
- ***Fragments*** represent parts of an *Activity* and should indicate their corresponding *layout* feature. Fragments are used inside Activities.
- ***Adapters*** are used to populate the *UI* (User Interface) with collections of data.
- ***Listeners*** are Java interfaces that represent user events.

Resources are also related to the presentation layer [22], and Android provides more than fifteen different resource types [23]. They are "non-Java" files used to build user interfaces, such as image, audio, or XML files. We relied on the existing resources of the project created from the default template[1] of Android Studio [24], which is the official integrated development environment for Android. The selected resources are: Layout, Strings, Style, and Drawable.

- ***Layout Resources*** are XML files used for the development of the *UI* structure of Android components. The development is done using a hierarchy of *Views* and *ViewGroups*. *Views* are text boxes, buttons, etc., while *ViewGroups* are a collection of *Views* with a definition of how these *Views* should be shown.
- ***String Resources*** are XMLs used to define sets of texts for internationalization.
- ***Style Resources*** are XMLs used to define styles to be applied in *layout* XMLs. Their goal is to separate code related to structure from code related to appearance and shape.

[1] Up to version 3.0 of Android Studio, the most current version at the time of this writing, the standard design template, which is pre-selected in the creation of a new Android project, is an Empty Activity.

– ***Drawable Resources*** represent a general concept for a graphic that can be drawn on the screen, including traditional images or specific XML files.

2.1 Developing a presentation layer in Android: A running example

In an Android app, a screen comprises two files: a Java class responsible for creating the screen and responding to the user events, and a layout resource, which is an XML file responsible for creating its visual interface.

An Activity is one of the major components of Android applications. It represents a UI screen, comprising buttons, listings, text input boxes, etc. To implement an Activity, it is necessary to create a class derived from the `Activity`, and to override some inherited methods. We highlight the *onCreate()* method. One of its responsibilities is to configure the user interface. In listing 1, we illustrate the code for creating an Activity. In line 5, we find the UI configuration, which indicates the *layout* "main_activity" feature.

Listing 1: An example of an *Activity* class.

```
1  public class MainActivity extends Activity {
2    @Override
3    public void onCreate(Bundle savedInstanceState) {
4      super.onCreate(savedInstanceState);
5      setContentView(R.layout.main_activity);
6    }
7  }
```

The UI of an Activity is built using layout resources, which are composed of XML files. In the following, we show an example of a layout resource.

Listing 2: An example of a *layout resource*.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout ...
3      android:layout_width="fill_parent"
4      android:layout_height="fill_parent"
5      android:orientation="vertical">
6
7
8      <TextView android:id="@+id/text"
9          android:layout_width="wrap_content"
10         android:layout_height="wrap_content"
11         android:text="ATextView" />
12
13
14     <Button android:id="@+id/button"
15         android:layout_width="wrap_content"
16         android:layout_height="wrap_content"
17         android:text="AButton" />
18
19
20     ...
21 </LinearLayout>
```

Although the examples presented are quite simple, real-world UIs tend to be much more robust and richer in information and interactivity. Such rich and robust UIs may result in large and complex code elements. Moreover, UI components usually have long and complex life cycles. An ACTIVITY, for example, has 7 different states in its life cycle (onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy()), while FRAGMENTS have 11 different stages. These numbers are high compared to the life cycle of non-UI related components (e.g., a SERVICE has only four). In such contexts, challenges in developing maintainable Android presentation code emerge.

3 Related Work

In this section, we present work related to traditional code smells, domain-specific smells, and smells for Android applications.

3.1 Traditional code smells

Webster's [4] book was likely the first code smells catalog, which focused on object-oriented software. Since then, several developers and researchers have studied this subject. As an example, Riel [25] has documented more than 60 different heuristics for object-oriented code. Fowler [3] suggests refactoring strategies for more than 20 smells.

Some researchers have focused on understanding the impacts of code smells on project quality. Khomh et al. [26], for example, conducted an empirical experiment in which they found that classes affected by code smells tend to suffer more changes than classes without code smells. In another study, Khomh et al. [27] noticed that classes affected by code smells are also more prone to defects. Li and Shatnawi [28] also empirically analyzed the impact of code smells and found a high correlation between code smells and detect-proneness. Yamashita and Moonen [29] explored the implications of inter-smell relations and explained how different interactions impact maintainability. On a related research, Abbes et al. [30], showed by means of a controlled experiment that the existence of a single code smell in a class does not significantly diminish developers' performance during maintenance tasks; however, when classes suffer from more than one code smell, performance is significantly reduced.

Other researchers have studied how developers perceive code smells. Palomba et al. [31] conducted an empirical experiment to evaluate the developers' perception of traditional code smells. Their results showed that developers easily perceive "simple" code smells. However, experience and knowledge play a significant role in identifying code smells related to good practices of object-oriented development.

Arcoverde et al. [32] conducted a survey to understand how developers react to the presence of code smells. The results showed that developers postpone removal to avoid API modifications. Peters and Zaidman [33] analyzed the

behavior of developers regarding the life cycle of code smells. Their results showed that awareness of a code smell is not enough to compel immediate refactoring.

3.2 Domain-specific Code Smells

Several researchers have been investigating the existence of code smells that are specific to a given technology, for example, MVC [9], Object-Relational Mapping [34], CSS [6], and formulas in spreadsheets [10].

Chen et al. [34] studied code smells in Object-Relational Mapping (ORM) frameworks, since developers are usually unaware of the impact of their code in database performance. The authors implemented an automated and systematic framework to detect and prioritize anti-performance standards in applications developed using ORM, and mapped two specific anti-patterns to ORM frameworks.

Aniche et al. [8, 9] investigated code smells related to the MVC architecture. After interviewing and surveying developers, the authors proposed a set of six smells related to the layers of an MVC application—Model, View, and Controller—and showed how each of them affects classes' change- and defect-proneness. Aniche et al. [35] also performed an empirical analysis in 120 open source systems and showed that each architectural role has a different code metric values distribution, which is a likely consequence of their specific responsibilities.

Gharachorlu [6] investigated code smells in CSS code, a widely used language in the presentation layer of web applications. According to the author, despite the simplicity of CSS syntax, language-specific features make CSS creation and maintenance a challenging task. A large-scale empirical study indicated that current CSS code suffers significantly from inadequate standards. The author proposes the first CSS quality model derived from a large sample to help developers estimate the total number of code smells in their CSS code. His main contribution was a set of eight new code CSS smells that can be detected with the CSSNose tool.

Finally, Fard and Ali [7] investigated code smells in JavaScript. The authors claimed that because of its flexibility, JavaScript is a particularly challenging language for writing and maintaining code. According to the authors, one of the challenges is that, unlike Android applications, which are compiled, JavaScript is interpreted. This means that there is usually no compiler to help developers detect incorrect or non-optimized code. Besides these challenges, the authors also fault JavaScript's dynamic, weakly typed, and asynchronous nature. They propose a set of 13 code smells for JavaScript: seven as adaptations of traditional code smells and six as language-specific smells. They also proposed an automated technique, called JSNOSE, to detect these code smells.

3.3 Code smells in Android Apps

Mannan et al. [36] state that 10% of the articles published in major software maintenance conferences between 2008 and 2015 considered Android projects in their research. They also observed that, when compared to traditional software, little research has been conducted on code smells in Android applications.

A significant portion of the research dedicated to code smells in Android applications focuses on studying the effects of traditional code smells. For example, Linares-Vásquez et al. [13] used the DECOR tool [37] to perform the detection of object-oriented anti-patterns in mobile applications developed with J2ME. Among their conclusions, the authors noticed a significant difference in the values of quality metrics in applications affected by code smells when compared to those that are not, and that while code smells occur in all domains, some code smells are more prevalent in specific domains.

Verloop [14] investigated the presence of traditional code smells [3] in Android applications to determine whether these code smells occur more often in "core classes," classes in the Android project that need to inherit from Android *SDK* classes, such as Activities, Fragments, and Services (as compared to "non-core" classes). To that aim, the author used four automatic code smell detection tools: JDeodorant, Checkstyle, PMD, and UCDetector. The author states that core classes tend to exhibit God Class, Long Method, Switch Commands, and Type Check code smells due to their nature of having many responsibilities. These smells were particularly high in Activities, which is the main component of the Android presentation layer. The author also found that the traditional code smell Long List Parameters is less likely to appear in core classes, as most of their method signatures come from classes defined in the Android SDK.

Reimann et al. [16] correlated the concepts of code smell, quality, and refactoring to introduce a catalog of 30 smells focused on usability, resource consumption, and security. Hetch [38] used the code smells detection tool Páprika [39] to identify 8 code smells. The author searched for the code smells in 15 popular Android applications, including Facebook, Skype, and Twitter. The author claims that traditional code smells are as prevalent in Android as in non-Android applications, except for the Swiss Army Knife code smell [40]. Mannan et al. [36] conducted a large-scale empirical study to compare the prevalence and effects of code smells on mobile and desktop applications. The authors found that while code smell density is similar in both mobile and desktop systems, some smells occur more often in mobile applications. For example, data classes and data clumps happen more often in a mobile app, while external duplication tends to happen more in desktop systems.

Researchers also showed that Android test code also contains test smells. More specifically, Peruma [41] explored the prevalence of test code smells in several open source Android applications. The author found that Android apps exhibit test smells early on in their lifetime, with varying degrees of co-

occurrences with different smell types, and that the existence of the test smells is also associated with higher change-proneness.

Gottschalk et al. [15] conducted a study on ways to detect and refactor code smells related to energy efficiency. The authors compiled a catalog with six code smells drawn from other research. Linares-Vásquez et al. [42], who also investigated energy consumption, showed that APIs related to user interface and database represent around 60% of the energy-greedy APIs. The authors also propose energy-saving recipes for Android developers, including "limit the use of the Model-View-Controller (MVC) pattern, especially when used in apps with many views" and "carefully design apps that make use of several views."

Other researchers also investigated performance and resource consumption. For example, Hetch et al. [43] studied the effects of three code smells (*Internal Getter/Setter*, *Member Ignoring Method*, and *HashMap Usage*) on the performance and memory-usage of two open source Android apps. Linares-Vásquez et al. [44] investigated the effects of micro-optimization in mobile applications. After a study of more than 3,500 mobile apps, the authors concluded that developers rarely make use of micro-optimizations and that the impact of these micro-optimizations on CPU/memory consumption is often negligible. Although not directly related to code smells, Liu et al. [45] conducted a study of 70 real-world performance bugs collected from eight Android applications. Among their findings, the authors show that most performance bugs (75%) are GUI lagging. In other words, they reduce responsiveness or the smoothness of the user interface. GUI lagging is indeed a concern of developers, as Linares-Vásquez et al. [46] show after surveying 485 open source developers.

Palomba et al. [47] propose 15 Android-specific smells and lightweight rules for their detection (that achieves an average precision and recall of 98%). The proposed code smells relate to different parts of an Android application, ranging from performance issues (e.g., the smell *Data Transmission Without Compression* arises when a method transmits a file over a network infrastructure without compressing it, and the *Inefficient SQL Query*, for which the authors suggest that the use of JDBC over network introduces too much overhead) to thread issues (e.g., the *Leaking Thread* happens when the application does not properly stop unused threads).

Android security code smells have also been explored by Ghafari et al. [48]. After reviewing scientific literature, the authors proposed a catalog of 28 smells that can lead to security vulnerabilities. The smells touch different security problems, such as insufficient attack protection, security validation, access control, data exposure, and input validation. After investigating the frequency of these code smells in around 46,000 open source mobile apps, the authors conclude that these smells occur in practice; some of them, such as *Dynamic Code Loading* and *XSS-like Code Injection*, happen in more than 50% of the apps.

4 Research Goals

The goal of our study is to catalog and empirically validate code smells that occur in the presentation layer source code of Android applications. To that aim, we employed a mixed method approach for understanding developers' perceptions, as their points of view play an important role in defining code smells related to a specific technology [31, 32, 49], especially considering the smells' intrinsic subjective nature [7, 50].

We investigate the following research questions (RQ):

- RQ_1: What code smells do developers observe in the presentation layer of Android apps?
- RQ_2: How often do developers observe the identified code smells and what importance do they give to them?
- RQ_3: How prevalent are the proposed code smells in real Android apps?

We employed two open online questionnaires to collect and confirm the smells, which were answered by 45 and 201 developers, respectively. We also developed a tool that automatically identifies the proposed code smells, and we analyzed the prevalence of the proposed code smells in 619 Android apps.

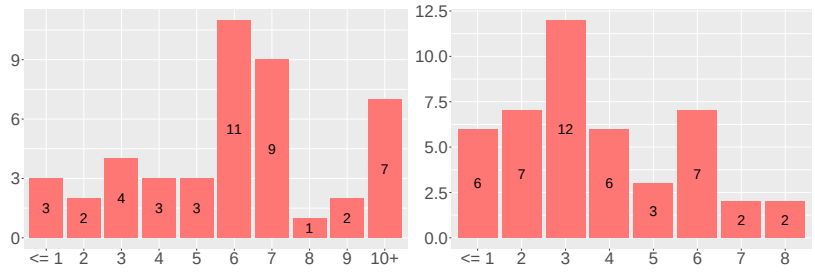
As the results of each RQ influenced the design of the subsequent step of the research, we present the method and results of each RQ in its own section.

5 A Catalog of Code Smells (RQ 1)

The first part of the study aimed to catalog code smells that occur in the presentation layer of Android apps. We employed an online questionnaire asking about good and bad practices related to components and resources of the Android's presentation layer.

5.1 Methodology and Questionnaire

The online questionnaire comprises 25 questions organized into three sections. The first section (6 questions) traces the participant's demographic profile (age, residence, experience in software development, experience with Android development, and schooling). The second section focuses on understanding what developers consider good and bad practices in each element of the presentation layer (*Activities*, *Fragments*, *Adapters*, *Listeners*, *Layout*, *Strings*, *Styles*, and *Drawables*). We asked about good and bad practices since developers may not be able to express code smells directly, but may report the measures they take to avoid problems. This strategy has also been applied in previous work by Aniche et al. [8, 9] to identify code smells in MVC applications. This part of the questionnaire comprises 16 optional open-ended questions: for each of the eight elements of the Android presentation layer, we asked a question related to good practices and another to bad practices. As an example, for the *Activity* element, we ask:



(a) Software experience (b) Android experience

Fig. 1: Participants' experience in the part I of our research ($N = 45$). X axis represents years of experience, Y axis represents the number of participants.

Q1 Do you have any good practices to deal with Activities?
Q2 Do you have anything you consider a bad practice when dealing with Activities?

The last section of the questionnaire comprises two open questions to capture any last thoughts not captured in the previous questions and one inviting participants to provide their email. The complete questionnaire can be seen in the online appendix [51].

Before the release, we conducted a pilot test with three Android developers. In the first configuration of the questionnaire, almost all questions were mandatory. With the result of the pilot test, we realized that developers do not always have good or bad practices to comment on all elements. Thus, we made such questions optional. The responses from the pilot study were disregarded to mitigate bias effects.

The questionnaire was released on Android forums, such as Android Dev Brasil[2], Android Brasil Projetos[3], and Slack Android Dev Br[4]. The authors of this paper also made use of their Twitter social networks to share the questionnaire. The questionnaire was open for approximately 3.5 months, from October 9, 2016, until January 18, 2017.

5.2 Participants

We obtained 45 responses. In Figure 1, we show the experience in software and Android development of our participants. Out of the 45 respondents, 90% had two years or more of software development experience, and 71% had two years or more of experience in Android development. It is noteworthy that the Android platform reached its 10th anniversary in 2018, i.e., five years of

[2] https://groups.google.com/forum/#!forum/androidbrasil-dev
[3] https://groups.google.com/forum/#!forum/android-brasil–projetos
[4] http://slack.androiddevbr.org

experience in this platform represented 50% of Android's lifetime. The questionnaire was replied to by developers from 3 continents and 7 countries. Most responses came from Brazil (81%).

5.3 Data analysis

Our analysis was inspired by the Grounded Theory approach (GT) [52, 53], which is increasingly popular in software engineering research [54]. GT is an inductive approach whereby qualitative data is analyzed to derive a theory. The goal of the approach is to discover new perspectives rather than confirm existing ones. Our analysis started from 45 responses to the questionnaire and occurred in 4 steps: *verticalization*, *data cleaning*, *codification*, and *split*, as detailed in the following.

The *verticalization* consisted of considering each good or bad practice response as an individual record to be analyzed. As each participant provided 18 answers to be analyzed, we started with 810 records.

The next step was *data cleaning*. This step consisted of removing answers that were not specific to the Android presentation layer, i.e., practices that could be applied to any other Android layer or even Java systems. Out of the 810 answers, 352 were considered, and 458 were disregarded. We could note that traditional code smells also apply to the Android context. The high number of responses (352) that were specifically related to the Android presentation layer shows that there are specific practices that take the architecture into account. Out of the 352 answers, 45% of them related to bad practices and 55% to good practices. In Table 1, we show how many answers we collected per survey question.

Next, we performed *codification* for good and bad practices [52, 55]. Codification is the process by which categories are extracted from a set of statements through the abstraction of central ideas and relations between the statements [52]. In our case, categories represented the code smells themselves. For each statement about bad practice, we either defined a new code smell that captured its essence or assigned it to an already identified smell. For the good practices, the authors used their knowledge of the Android platform, analyzed the goal of the good practice, and either defined a new code smell or assigned the practice to an existing one. As a single statement can belong to more than one code smell, some of them received more than one category. In this step, we also disregarded more answers that were not "obviously disposable" in the previous step. For each response not considered in this step, we recorded the reason, which can be found in our online appendix [51].

Finally, we performed the *split* step, which consisted of dividing responses that belonged to more than one category into two or more answers. As an example, *"Do not make Activities to be callbacks of asynchronous executions. Always inherit from support classes, never directly from the platform."* indicates one category in the first sentence and another one in the second. In some cases, the complete response was necessary to understand both catego-

#	Question	Total of participants	Participants
Q1	Good practice / Activities	36 (80%)	P1, P2, P4-P12, P14-P17, P19, P22, P23, P25-P32, P34-P37, P39-P43, P45
Q2	Bad practice / Activities	35 (78%)	P2, P4-P11, P14-P17, P19, P22, P23, P25-P32, P34-P37, P39-P45
Q3	Good practice / Fragments	33 (73%)	P4-P11, P14-P17, P19, P22, P23, P25-P28, P30-P32, P34-P37, P39-P45
Q4	Bad practice / Fragments	31 (69%)	P2, P4-P11, P14, P15, P17, P19, P22, P23, P25-P28, P31,P32, P34-P37, P39-P43, P45
Q5	Good practice / Adapters	30 (67%)	P2, P4-P11, P14, P15, P17-P19, P22, P23, P26, P28, P29, P31,P32, P34-P37, P39-P43, P45
Q6	Bad practice / Adapters	27 (60%)	P2, P4-P8, P10, P11, P14, P18, P19, P22, P23, P26, P28, P31, P34-P37, P39-P45
Q7	Good practice / Listeners	24 (53%)	P2, P4-P6, P8, P9, P11, P14, P22, P23, P26, P28, P29, P31, P32, P34, P36, P37, P39-P43, P45
Q8	Bad practice / Listeners	23 (51%)	P2, P4, P5, P8, P9, P11, P14, P19, P22, P23, P26, P28, P31, P32, P34, P36, P37, P39-P44
Q9	Good practice / Layout Resources	28 (62%)	P4-P9, P11, P14, P19, P22, P23, P26-P29, P31, P32, P34-P37, P39-P45
Q10	Bad practice / Layout Resources	23 (51%)	P4, P5, P7-P9, P11, P22, P23, P26, P28, P31, P32, P34-P37, P39-P45
Q11	Good practice / Styles Resources	23 (51%)	P4-P9, P11, P18, P22, P23, P26, P28, P31, P32, P34-P37, P39-P43
Q12	Bad practice / Styles Resources	22 (49%)	P4-P8, P11, P18, P22, P23, P26, P28, P31, P32, P34-P37, P39-P43
Q13	Good practice / String Resources	28 (62%)	P4-P6, P8-P11, P14, P18, P22, P23, P26-P29, P31, P32, P34-P37, P39-P45
Q14	Bad practice / String Resources	23 (51%)	P4-P6, P8, P9, P11, P14, P18, P22, P23, P26, P28, P31, P32, P34-P37, P40-P43, P45
Q15	Good practice / Drawable Resources	24 (53%)	P4-P6, P8-P11, P14, P18, P22, P23, P26, P28, P31, P32, P34-P37, P39-P43
Q16	Bad practice / Drawable Resources	21 (47%)	P4-P6, P8, P11, P14, P18, P22, P23, P26, P28, P31, P32, P34, P36, P37, P40-P44
Q17	Other good practices	22 (49%)	P2, P4, P8, P10, P11, P14, P18, P22, P23, P26, P28, P31, P32, P34, P36, P37, P39-P43, P45
Q18	Other bad practices	20 (44%)	P2, P4, P8, P10, P11, P18, P22, P23, P28, P31, P32, P34, P36, P37, P40-P45

Table 1: Participants and questions they answered (participants = 45).

rizations, in which case we maintained the original answer. At the end of the analysis, 359 responses were individually categorized into 46 categories.

The first author of this paper conducted the verticalization, data cleaning, codification, split, and categorization steps. The second author of the paper intervened whenever the first author had questions about a specific coding. Both authors discussed until reaching a final agreement. At the end of the coding process, the first and the second authors discussed all the derived codes and together derived the final definition of the code smells.

In the usability community, Nielsen [56] suggests that five repetitions are enough to characterize a recurring problem, and successive repetitions tend not to aggregate new relevant information. After experimenting with the number five as the minimum number of mentions, we obtained 20 smells, which belonged to two different groups: 9 of them related to the Java classes of the Android presentation layer, and 11 related to resources (string, layout, style, and drawable). After some consideration from the authors, we decided that this catalog met our criteria of having a reasonable number of recurrent smells covering the Android presentation layer.

5.4 Results

Activities was the element with the highest number of answers: 35 (78%) out of the 45 respondents answered the question about good practices while 38 (84%) responded to the question about bad practices. The element that received the least number of responses about good practices was *Listener*, which was answered by 10 (22%) participants. The elements that received the fewest responses about bad practices were *Style* resources and *Drawable*, both of which were answered by 9 (20%) participants.

The coding process resulted in 46 categories. As aforementioned, to derive a code smell we considered all 22 categories that presented occurrences greater than or equal to five. Out of the 22, we disregarded 2 categories because they were either (i) too similar to a traditional code smell (Large Class) or (ii) too focused on object-oriented programming (inheritance). In the online appendix, we report the full coding results [51].

In Table 2, we present a summary of each code smell, and in Table 3, we show how often our participants mentioned that smell. In the following paragraphs, we present the definition of the code smells, as well as the elements affected and related symptoms. We provide more information about each smell, such as code examples and refactoring suggestions, in a dedicated website.[5]

Brain UI Component: *Activities*, *Fragments*, and *Adapters* should be responsible for presenting, interacting, and updating the UI only. Business logic should be implemented elsewhere. This idea is similar to what Evans [57] calls the separation of the "UI layer" and the "domain layer." The existence in

[5] http://suelencarvalho.com/android-presentation-layer-code-smells.

Table 2: The proposed code smells in the presentation layer of Android apps. The smells are ordered by the number of times they were mentioned in the survey.

	Name	Summary
Component smells	Brain UI Component	UI components with business logic.
	Coupled UI Component	UI components with concrete references to each other.
	Suspicious Behavior	*Listener* being implemented within an UI component.
	Fool Adapter	*Adapters* that do not use the *ViewHolder* pattern.
	Absence of an Architecture	Presentation layer without a known/clear architecture.
	Excessive Use of Fragments	Use of *fragments* without an explicit need.
	UI Component Doing I/O	UI components making access to I/O, e.g., database.
	No Use of Fragments	The lack of *Fragment*s prevents UI with behavior reuse.
	Flex Adapter	*Adapters* with any (business or view) logic.
Resource smells	No Naming Pattern	No naming pattern in Resources.
	Magic Resource	Strings, numbers, or colors hardcoded.
	Deep Nested Layout	Layout resources with deep levels of nested Views.
	Unnecessary Image	Images that could be transformed into a graphic resource.
	Long or Repeated Layout	*Layout* resources that are too long or with duplicated code snippets.
	Missing Image	Image without all standard resolutions.
	God Style Resource	Long *Style* resources that contain too much data.
	God String Resource	*String* resource without a clear naming pattern.
	Duplicate Style Attributes	Repeated attributes in *layout* or *style* resources.
	Inappropriate String Reuse	*Strings* being reused improperly within resources.
	Hidden Listener	Listeners being configured inside of *layout* resources.

presentation layer elements of code related to business logic, I/O operations, conversion of data, or static fields is a sign of code smell.

No Naming Pattern: This smell happens when resources (*layout*, *string*, *style*, and *drawables*) do not follow a naming pattern. More specifically, it happens when the file where the resource is located and its internal name (i.e.,

Code smell	Qty of codes	# of Participants
Brain UI Component	60	21 (P2, P6-7, P9, P10-11, P16-17, P19, P23, P25, P27-28, P31, P34-37, P39-41)
Coupled UI Component	18	13 (P2, P4, P6, P10, P19, P23, P31, P36-37, P40, P44-45)
Suspicious Behavior	18	11 (P4, P6, P8-10, P32, P34, P37, P42-44)
Fool Adapter	13	12 (P4, P6-8, P11, P17, P31, P35-36, P39, P43, P45)
Absence of an Architecture	13	10 (P1, P4, P8, P12, P15, P26, P28, P31, P42, P45)
Excessive Use of Fragments	9	7 (P2, P4, P7, P11, P30, P39, P41)
UI Component Doing I/O	9	4 (P2, P26, P37, P41)
No Use of Fragments	8	7 (P9-10, P31, P14, P19, P34, P45)
Flex Adapter	6	6 (P2, P7, P23, P39, P40, P41)
No Naming Pattern	23	10 (P4, P6, P8, P11, P27, P29, P34, P37, P39, P43)
Magic Resource	23	14 (P14, P23, P26, P27, P29, P31-32, P34-36, P41, P43-45)
Deep Nested Layout	21	15 (P2, P4, P6-8, P14, P19, P26, P36-37, P39-41, P44-45)
Unnecessary Image	18	13 (P6, P8-9, P11, P14, P23, P28, P35-37, P40-42)
Long or Repeated Layout	15	13 (P4, P6, P7, P9, P23, P26, P28, P32, P34, P36, P40-42)
Missing Image	12	10 (P4, P8, P10, P11, P31, P34, P36, P40, P42, P44)
God Style Resource	8	5 (P7-8, P28, P40, P42)
God String Resource	8	6 (P8, P26, P28, P32, P41, P42)
Duplicate Style Attributes	8	8 (P4, P8, P28, P32, P34, P39-41)
Inappropriate String Reuse	6	5 (P4, P6, P9, P32, P40)
Hidden Listener	5	3 (P34, P39, P41)

Table 3: The origin of each of the code smells (participants = 45). Quantity of codes represent the number of times the smell was mentioned. Note that a participant may have mentioned the same smell more than once in their survey. Thes smells are ordered by the number of times they were mentioned in the survey.

how the resource is called inside the source code) differ. These different names cause confusion among developers.

Magic Resource: A smell that occurs when resources (e.g., *layout*, *string*, and *style*) are hard-coded instead of pointing to an existing resource file.

Deep Nested Layout: Deep nesting when constructing *layout* resources was considered a code smell. Interestingly, the official Android website has more information and provides automated tools to deal with this problem [58].

Unnecessary Image: Android has resources that can replace images. The smell occurs when the system has images with, for example, pure solid colors or gradients, which could be replaced by Android's native *shapes*.

Coupled UI Component: In order to be reused, *Fragments*, *Adapters*, and *Listeners* should not have a direct reference to who uses them. The existence of direct reference to *Activities* or *Fragments* in these elements is an evidence of code smell.

Suspicious Behavior: *Activities*, *Fragments*, and *Adapters* should not contain in their source code the implementation of event handlers. First, event handling code, when embedded into one of these components, is implemented through anonymous or internal classes. As the interfaces that these event handlers need to implement are often complex, the source code of *Activities*, *Fragments*, and *Adapters* becomes less readable. Second, an event handler often makes use of business rules and domain models. A less attentive developer may then write these business rules directly into the event handler (which then leads us to a possible Brain UI Component smell). The use of anonymous classes or internal classes to implement *Listeners* to respond to user events is a sign of code smell.

Long or Repeated Layout: The code smell appears when long or duplicated *layout* resources occur in the source code.

Fool Adapter: This smell occurs when *Adapters* do not reuse instances of the *views* that represent the fields that will be populated for each item of a collection using the *View Holder* pattern.

Absence of an Architecture: This smell happens when one cannot easily identify how the components are organized. Developers cannot identify whether the application makes use of Model-View-Controller (MVC), Model-View-Presenter (MVP), or Model-View-ViewModel (MVVM).

Missing Image: This code smell happens when the system has only a single version of .png, .jpg, or .gif images. The Android platform encourages images to be available in more than one size or resolution to perform optimizations.

Excessive Use of Fragments: This smell emerges when *Fragments* are used without an explicit need. Examples include applications that do not need to support tablets and when *Fragments* are used in only a single screen of the app.

UI Component Doing I/O: *Activities*, *Fragments*, and *Adapters* performing I/O operations, such as database and file access, cause this smell.

No Use of Fragments: Fragments can decouple UI with behavior pieces. The non-use of fragments can be a smell in visually rich apps. Such apps have a high number of different behaviors, animations, and events to handle. If all the implementation relies on a single Activity, for example, this class will be highly complex and hard to understand. Moreover, visually rich apps

are also often responsive, i.e., have different UIs for different screen sizes. In this case, not using fragments will hinder code reuse. This code smell emerges when view components (e.g., *EditTexts* or *Spinners*) are directly used by an *Activity* instead of a *Fragment*.

God Style Resource: Long *style* resources define this smell. Symptoms of this smell happen when all styles are defined in the same *styles.xml*.

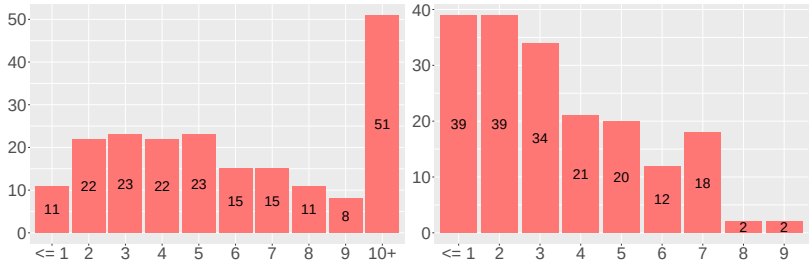
God String Resource: This smell is defined by Long *string* resources. Developers should separate their string resources according to a rule: e.g., one string resource per screen.

Duplicate Style Attributes: Android developers often choose to define the style of a UI element directly in the layout file. However, this might lead to unnecessary duplication (e.g., the same complex style appears in different components). The existence of duplicated style definitions in different components indicates this code smell.

Flex Adapter: *Adapters* should be responsible for populating a *view* from a single object. The code smell emerges when *Adapters* contain business or view logic. As we discussed in the Brain UI Component smell, UI logic and business rules should remain separate from each other.

Inappropriate String Reuse: Developers reuse strings among the different UIs of the application. For example, the string "Name" might appear in many parts of the app; thus, developers write this string only once in a string resource file and reuse it whenever they need it. However, the smell happens when developers reuse the same *string* in different parts of the system because the string is coincidentally the same, and not because they represent the same concept in the UI. For example, in one part of the app, "name" might refer to the name of the user, whereas in another part of the app, "name" might refer to the name of the user's favorite band. Reusing strings simply because of their similarity might lead to two problems: First, if developers decide to change the string, they need to be aware that the changes will be reflected throughout the entire application. Second, when adding support for multiple languages, one language might need two words to express what another language can communicate in one.

Hidden Listener: *Layout* resources should only be responsible for presenting data. This smell appears when these resources also configure the listener that will respond to events, such as the *onClick* event. Event handling in XML files makes it harder for developers to identify which listeners are used and where. Although the most recent versions of IDEs are able to show developers which events are declared in an XML file when reading the respective Java file, events that are declared in XML files are "hidden" from developers who primarily work in Java code.



(a) Software experience (b) Android experience

Fig. 2: Participants' experience in the part II of our research ($N = 201$). X axis represents years of experience, Y axis represents the number of participants.

RQ$_1$. Based on developers' reports of good and bad practices, we cataloged 20 code smells for the presentation layer of Android apps: 9 related to components (Activities, Fragments, Adapters, and Listeners), and 11 related to resources (Layout, String, Style, and Drawable resources).

6 Importance and Frequency of the Code Smells (RQ 2)

The second part of the research aimed to understand the perceptions of the developers regarding the frequency and importance of the proposed smells. We collected these perceptions through another survey.

6.1 Methodology and Survey

This survey has three sections (the full version is available in the appendix). The first section (6 questions), as in the first step, collects the participants' demographic profile (age, residence, software development experience, Android development experience, and education). The second section captures developers' perceptions about how often they come across the smells in their Android systems. The third section captures perceptions of the developers regarding the importance of mitigating the code smells. In this survey, we were not interested in collecting more code smells, but rather in confirming the ones we devised in the first part. However, we did not indicate that code smells would be presented, nor did we mention the names of the smells used in this research. We have chosen this approach to avoid having to fully explain the code smells.

To investigate frequency (second part of the survey), we presented a list of statements derived from RQ$_1$ where each statement described in practical terms how the smells manifest themselves in the source code. For each statement, the participant could choose one of five scale options from the frequency range: very common, frequent, sometimes, rarely, and never. We presented

25 statements to contemplate the 20 code smells from RQ1. The difference in these numbers occurred because, for four of the code smells—Suspicious Behavior, Long or Repeated Layout, God Style Resource, and Duplicate Style Attributes)—more than one statement was presented, each addressing one symptom. We chose to separate the symptoms into statements to understand which ones were frequently perceived by developers.

To investigate importance (third part of the survey), we asked developers about the importance of mitigating the smells. We decided to present mitigation approaches instead of the code smells, since: 1) in the previous question, we had already introduced the smells and asked about how often they see the smells, and that would be too repetitive for the participants; and 2) showing them ways to mitigate the problem would give them a different perspective on the proposed code smells (which, we conjecture, can make them rethink their answers). The participants were asked to indicate how important they considered 21 sentences related to approaches that mitigate the proposed code smells. Again, the divergence of the total number of code smells, and the total of mitigation statements can be explained because of smells with more than one symptom. For each statement, the participant could choose one of the following options: very important, important, fairly important, slightly important, and not important.

Before publishing the questionnaire, we performed pilots with two experienced Android experts, *DEV-A* and *DEV-B*. *DEV-A* has 10 years of software development experience and 5 years of experience in Android development, considers himself proficient in Java, Objective C, Swift, and Android technologies, and holds a bachelor's degree in Information Technology. *DEV-B* has 7 years of software development experience and 6 years of experience in Android development, considers himself proficient in Java, Objective C, and Android technologies, and has a postgraduate degree in Service Oriented Software Engineering. In these pilot studies, we walked the experts through all the sentences we devised to the questionnaire and asked them to think aloud while reading each sentence. Our primary goal was to make sure all sentences made sense and were clear. We used their feedback to improve the formulation of the sentences. After all the improvements, the two experts agreed that all the sentences in the survey were clear and conveyed our intention and that it was ready to go public. Although we did not show the definitions of the code smells to the experts (only the survey), as the survey is intrinsically related to the smells, their feedback also helped us in sharpening the final definition of the smells.

The questionnaire was open for approximately three weeks in mid-September 2017 and was shared in the same venues as in Part 1. The statements were presented randomly, and 201 developers answered the questionnaire. A possible explanation for the difference in the number of answers (Part 1 received 45 answers) is due to the differences in format: while Part 1 was focused on open questions that take a long time to respond to, Part 2 mostly contained closed questions, which take less time to complete and are thus more attractive to participants.

6.2 Participants

In Figure 2, we show the experience of the 201 participants that answered our survey: 94% indicated they had two years or more of experience in software development, and 74% indicated two years or more of experience in Android development. In addition, 15% had one or more post-graduation degrees, and 61% had a bachelor's degree. Most participants were between 20 and 35 years old. We also asked participants about their level of knowledge in various object-oriented languages. More than 80% claim to have intermediate or expert knowledge in Java and Android. Five participants (2%) stated that they did not know about Android, so their answers were disregarded in the analysis. We obtained responses from Android developers from 3 continents and 14 different countries. Similar to the previous survey, 78% of participants are from Brazil.

6.3 Results

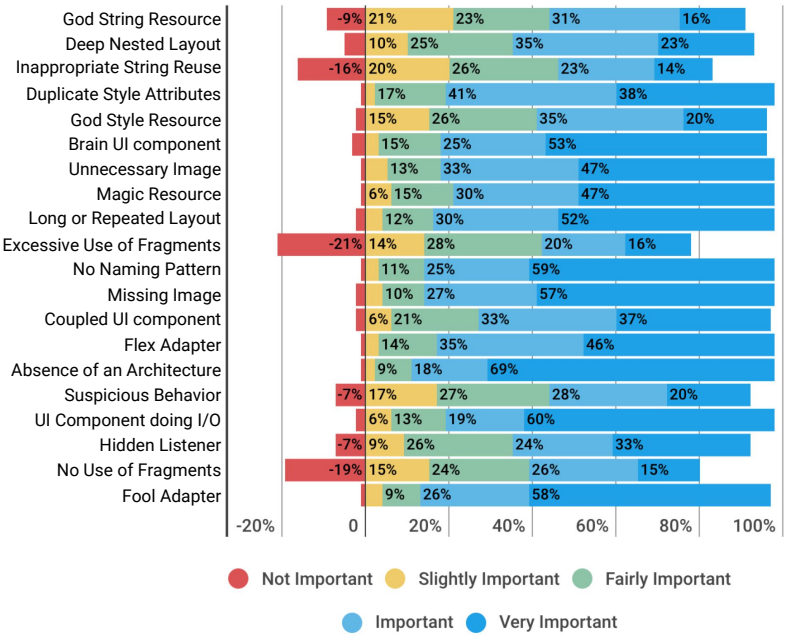
In Figures 3a and 3b, we show the participants' perception of the importance and frequency of the identified code smells. In Table 4, we present the median, mode, and standard deviation of their answers (varying from 1 to 5).

Most code smells are considered highly important by developers. We see that most code smells (either related to components or resources) have a mode equal to or greater than four, meaning that most developers considered them to be from "important" to "highly important."

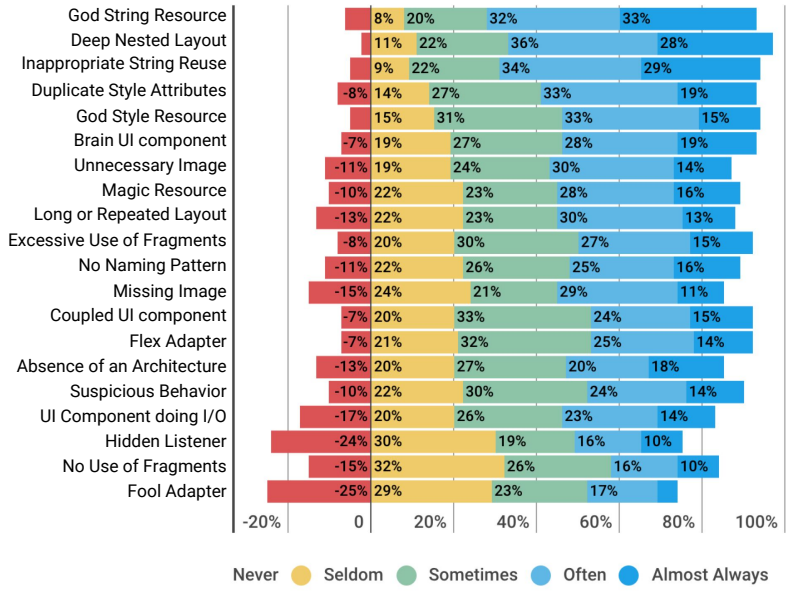
Too many or too few fragments? Two of the code smells are opposite to each other: Excessive Use of Fragments and No Use of Fragments. Our data shows that there is no definite perception of their importance. Interestingly, not even popular Android best practice guides, such as Futurice [59], have clear suggestions on when to use Fragments. Quoting the guide: *"We suggest you sail carefully, making informed decisions since there are drawbacks for choosing a fragments-only architecture, or activities-only."* Our results, together with the current best practice guidelines, suggest that better guidelines for how to use *Fragments* are necessary.

Developers often encounter the proposed code smells in their apps. To all other code smells (except two: Fool Adapter and Hidden Listener), developers' perceptions of frequency range from "sometimes" to "almost always." This means that developers often find the code smells in their apps.

Fool Adapter and Hidden Listener are highly important, but do not occur often. The mode for these two code smells was smaller than 3, meaning that participants "seldom" or "never" noticed them. However, they both are considered highly important: Fool Adapter was considered highly important by 58% of participants (the second most important code smell), and Hidden Listener was considered highly important by 33% of partici-



(a) Importance



(b) Frequency

Fig. 3: Frequency and importance of the proposed code smells, according to our participants' perceptions.

Table 4: Frequency and importance of the proposed code smells, according to our participants' perceptions.

Code smell	**Importance**			**Frequency**		
	Median	**Mode**	**Std Dev**	**Median**	**Mode**	**Std dev**
Brain UI Component	5	5	1.05	3	4	1.19
Magic Resource	4	5	1.00	3	4	1.24
Unnecessary Image	4	5	0.95	3	4	1.23
Long or Repeated Layout	4	5	0.95	4	4	1.07
Missing Image	5	5	0.95	3	4	1.25
Coupled UI Component	4	5	1.02	3	3	1.15
UI Component Doing I/O	5	5	1.03	3	3	1.29
Absence of an Architecture	5	5	0.82	3	3	1.30
Flex Adapter	4	5	0.91	3	3	1.15
No Naming Pattern	5	5	0.88	3	3	1.24
Fool Adapter	5	5	0.93	2	2	1.20
Hidden Listener	4	5	1.23	2	2	1.29
God Style Resource	4	4	1.06	4	5	1.18
God String Resource	3	4	1.22	4	5	1.18
Suspicious Behavior	3	4	1.19	3	4	1.19
Deep Nested Layout	4	4	1.12	4	4	1.06
Long or Repeated Layout	4	4	0.86	4	4	1.11
No Use of Fragments	3	4	1.34	3	2	1.21
Inappropriate String Reuse	3	3	1.29	4	4	1.12
Excessive Use of Fragments	3	3	1.36	3	3	1.17
Average SD			1.05			1.19

pants. These results suggest that developers already know the benefits of the *ViewHolder* pattern [60] and are avoiding the FOOLADAPTER smell. In addition, developers are already avoiding defining events in *layout resources*, and thus, avoiding the HIDDEN LISTENER smell.

RQ$_2$. Developers consider most of the proposed smells as important and frequent.

7 Prevalence of the code smells (RQ 3)

The third part of our study aimed to analyze how prevalent the proposed smells are in real Android apps. To that aim, we devised a tool, named ANDROIDUIDETECTOR.[6] Our tool relies on a combination of AST visitors and heuristics and it was designed based on two parsers: (i) JavaParser[7] and (ii) JDOM[8]. The former is used to parse Java files in a lightweight and straightforward way, while the last is used to process XML files.

[6] https://github.com/julioverissimo88/AndroidUIDetector
[7] https://javaparser.org/
[8] http://www.jdom.org/

7.1 Code Smell Detection Strategies

We implemented detection strategies in our tool for 15 out of the 20 proposed smells. We did not implement five smells: No Naming Pattern, Unnecessary Image, Long or Repeated Layout, Inappropriate String Reuse, and Absence of an Architecture, as they are more subjective and require more than static analysis.

7.1.1 Detection strategies for the component-related smells

This section presents the detection strategies used to identify the eight component-related smells.

Coupled UI Component: `Fragments`, `Adapters`, and `Listeners`, to be reused, should not have direct reference to who uses them. The detection strategy is as follows: we collect all `Fragments`, `Adapters`, and `Activities` of the app. For each component, we check whether any of its fields is a direct reference to another `Activity` or `Fragment`. If so, we mark the component as smelly. Algorithm 1 depicts this detection strategy.

Algorithm 1: Coupled UI Component algorithm.

```
1 foreach j in listOfJavaFiles do
2 |   if (j.isActivity() or j.isFragment() or j.isAdapter()) then
3 |   |   foreach f in j.getFields() do
4 |   |   |   if (f.isActivity() or j.isFragment()) then
5 |   |   |   |   listOfSmells.add(f)
6 |   |   |   end
7 |   |   end
8 |   end
9 end
```

Suspicious Behavior: `Activities`, `Fragments`, and `Adapters` should not be responsible for implementing event behavior. The detection strategy is as follows: we collect all `Fragments`, `Adapters`, and `Activities` of the app. For each component, we verify whether it contains either an (i) inner class or (ii) an anonymous class (as inner and anonymous classes are how developers often implement event behavior). If a component possesses any of them, we mark it as smelly. Algorithm 2 presents this detection strategy.

Algorithm 2: Suspicious Behavior algorithm.

```
1 foreach j in listOfJavaFiles do
2 |   if (j.isActivity() or j.isFragment() or j.isAdapter()) then
3 |   |   if (j.containInnerClass() or j.containAnonymousClass()) then
4 |   |   |   listOfSmells.add(j)
5 |   |   end
6 |   end
7 end
```

Brain UI Component: `Activities`, `Fragments`, `Adapters`, and `Listeners` should only contain code responsible for presenting, interacting, and updating the UI. The detection strategy is as follows: we collect all `Fragments`, `Adapters`, and `Activities` of the app. For each component, we measure its (McCabe) code complexity and identify whether it makes use of I/O operations, database access, or static fields. We use this heuristic as a proxy for business rules, as there is no clear and unambiguous way of deciding whether a piece of code has business logic. Algorithm 3 presents this detection strategy. Please note that α and β are thresholds and we describe how we calculate them in the next section.

Algorithm 3: Brain UI Component algorithm.

```
1 foreach j in listOfJavaFiles do
2 |  if (j.isActivity() or j.isFragment() or j.isAdapter()) then
3 |  |  if (WMC(j) > α or hasIO(j) or hasDB(j) or hasSF(j) > β)
  |  |   then
4 |  |  |  listOfSmells.add(j)
5 |  |  end
6 |  end
7 end
```

Flex Adapter: `Adapters` should be responsible for populating a view from a single object. The detection strategy is as follows: for each `Adapter` in the app, we verify whether its complexity is below a specific threshold. We use complexity as a proxy, as highly complex `Adapters` often deal with more than one object. Algorithm 4 presents this detection strategy.

Algorithm 4: Flex Adapter algorithm.

```
1 foreach j in listOfJavaFiles do
2 |  if (j.isAdapter()) then
3 |  |  if (WMC(j) > α) then
4 |  |  |  listOfSmells.add(j)
5 |  |  end
6 |  end
7 end
```

Fool Adapter: `Adapters` should use the View Holder pattern to reuse instances of the views that represent the fields that will be populated for each item of a collection. The detection strategy is as follows: for each Adapter (or any of its children, e.g., `BaseAdapter`, `ArrayAdapter`, and `CursorAdapter`), we detect whether there is a call to `findViewById()` inside its `getView()`

method. If so, we mark the class as smelly. Algorithm 5 illustrates this detection strategy.

Algorithm 5: Fool Adapter algorithm.

```
1  foreach j in listOfJavaFiles do
2      if (j.isAdapter()) then
3          foreach m in j.getMethods() do
4              if (m.getName().equals(getView)) then
5                  foreach st in m.getMethodStatements() do
6                      if (st.getName().equals(findViewById)) then
7                          listOfSmells.add(st)
8                      end
9                  end
10             end
11         end
12     end
13 end
```

UI Component Doing I/O: `Activities`, `Fragments`, and `Adapters` should not perform I/O operations, such as database and file access. The detection strategy is as follows: for each `Activity`, `Fragment`, and `Adapter`, we check whether they make any call to I/O, database, or internet request APIs. We created the dataset of APIs by scraping the Android manual. Algorithm 6 depicts this detection strategy.

Algorithm 6: UI Component Doing I/O algorithm.

```
1  foreach j in listOfJavaFiles do
2      if (j.isActivity() or j.isFragment() or j.isAdapter()) then
3          foreach m in j.getMethods() do
4              foreach st in m.getMethodStatements() do
5                  if (hasIO(st) or hasIR(st) or hasDB(st)) then
6                      listOfSmells.add(st)
7                  end
8              end
9          end
10     end
11 end
```

No Use of Fragments: UI decoupling is recommended for improving maintenance. `Fragments` are often used to accomplish this task. Thus, the non-use of `Fragments` can represent a highly coupled UI. In practice, we can observe this smell when view components, e.g., `EditTexts`, `Spinners`, and `TextViews`, are directly used by an `Activity`, instead of small `Fragments`. The detection strategy is similar to what we described above: for each `Activity` of the app, we check whether it contains any view component (e.g., `EditTexts`,

`TextViews`, `Spinners`, etc.). If so, we mark the component as smelly. This detection strategy is depicted in Algorithm 7.

Algorithm 7: No Use of Fragments algorithm.

```
1  foreach j in listOfJavaFiles do
2      if (j.isActivity()) then
3          foreach f in j.getFields() do
4              if (containsViewComponents(f)) then
5                  listOfSmells.add(f)
6              end
7          end
8          foreach m in j.getMethods() do
9              if (containsViewComponents(m)) then
10                 listOfSmells.add(m)
11             end
12             foreach st in m.getMethodStatements() do
13                 if (containsViewComponents(st) then
14                     listOfSmells.add(st)
15                 end
16             end
17         end
18     end
19 end
```

Excessive Use of Fragments: Although the use of `Fragments` is important for UI decoupling, these components should not be used without an explicit need. To automate the identification of this smell, we count the number of `Fragments` in an app. If the number is higher than a pre-defined threshold, we mark the app as smelly. We define the threshold later in this paper. In Algorithm 8, we present the detection strategy, where α represents the threshold.

Algorithm 8: Excessive Use of Fragments algorithm.

```
1  foreach j in listOfJavaFiles do
2      if (j.isFragment()) then
3          counter = counter + 1
4      end
5  end
6  if (counter > α) then
7      listOfSmells.add(f)
8  end
```

7.1.2 Detection strategies for the resource-related smells

God Style Resource: This smell happens when a single style is overly complex. We detect this smell by counting the number of lines in all resources

of the app (i.e., XML files). All resources that have the number of lines of code higher than a threshold are marked as smelly. The detection strategy is presented in Algorithm 9, where α represents the threshold.

Algorithm 9: God Style Resource algorithm.

```
1 foreach x in listOfXMLFiles do
2 |   if (x.isStyle()) then
3 |   |   counter = counter + 1
4 |   else if ((x.isStyle) and (x.lenght() > α)) then
5 |   |   listOfSmells.add(x)
6 end
7 if (counter == 1) then
8 |   listOfSmells.add(x)
9 end
```

Deep Nested Layout: Hierarchies of long and deeply nested views in layouts should be avoided. Any resource that has a nested view deeper than a pre-defined threshold is considered smelly. Algorithm 10 depicts the detection strategy, where α represents the threshold.

Algorithm 10: Deep Nested Layout algorithm.

```
1 foreach x in listOfXMLFiles do
2 |   if (deep(x) > α) then
3 |   |   listOfSmells.add(x)
4 |   end
5 end
```

Duplicate Style Attributes: Duplicated styles are considered a smell. We detect this smell by collecting all XML files available in the "res/-values" folder of the app and looking for repeated properties among these files. If we find a repeated property, we mark the resource as smelly. Algorithm 11 depicts the detection strategy.

Algorithm 11: Duplicate Style Attributes algorithm.

```
1 foreach x in listOfXMLFiles do
2 |   if (containsRepeatedProperties(x)) then
3 |   |   listOfSmells.add(x)
4 |   end
5 end
```

Hidden Listener: Layouts should only handle information presentation. It is a sign of smell to use event attributes, such as "`onClick`," directly in *layout* files. We detect this smell by searching for the usage of

`android:onClick` in any layout resource file (i.e., any XML file inside the "`res/layout`" folder of the app). Algorithm 12 depicts this detection strategy.

Algorithm 12: Hidden Listener algorithm.

```
1 foreach x in listOfXMLFiles do
2 |  if (seek(x, "android:onClick")) then
3 |  |  listOfSmells.add(x)
4 |  end
5 end
```

Magic Resource: Every text or color used in the app should be created in its respective resource file, and then reused throughout the app. It is a sign of the smell when strings and colors appear directly in the source code rather than referencing an existing resource. We detect this smell by observing the usage of all Android's text and color markers in layout resources (e.g., `android:text` and `android:textColor`). If the marker has a hard-coded text or color (rather than referencing a resource file), we mark the resource as smelly. Algorithm 13 depicts this detection strategy.

Algorithm 13: Magic Resource algorithm.

```
1 foreach x in listOfXMLFiles do
2 |  if (seek(x, "android:text")) then
3 |  |  if (!regex("@.*/.*")) then
4 |  |  |  listOfSmells.add(x)
5 |  |  end
6 |  else if (regex("android:.*Color.*")) then
7 |  |  listOfSmells.add(x)
8 end
```

God String Resource: It is a good practice to separate string resources according to some rules, e.g., one string resource per screen. To detect this smell, we compare the amount of `Activities` and string resources (i.e., resource files that contain the `string` element in the `res/values` folder of the

app). If they are different, we mark the app as smelly. Algorithm 14 depicts the detection strategy applied to this smell.

Algorithm 14: God String Resource algorithm.

```
1  foreach x in listOfXMLFiles do
2  |  if (x.contain("<string>")) then
3  |  |  counterString = counterString + 1
4  |  end
5  end
6  foreach j in listOfJavaFiles do
7  |  if (j.isActivity()) then
8  |  |  counterActivity = counterActivity + 1
9  |  end
10 end
11 if (!(counterString == counterActivity)) then
12 |  listOfSmells.add(x)
13 end
```

Missing Image: This smell happens when the system contains only a single version of its `.png`, `.jpg`, or `.git` images. We detect this smell by checking whether all images of the app exist in all resolutions (i.e., that the same images exist in `res/folders-hdmi`, `res/folders-xhdpi`, `res/folders-xxhdpi`, and `res/folders-xxxhdpi` folders). We also verify whether the file sizes differ from each other. Algorithm 15 depicts this detection strategy.

Algorithm 15: Missing Image algorithm.

```
1  foreach file in listOfFiles do
2  |  if (containDuplicatedFiles(file) or containSameSizeFiles(file)) then
3  |  |  listOfSmells.add(file)
4  |  end
5  end
```

7.2 Sample dataset

To study the prevalence of the proposed code smells, we randomly selected open-source apps listed in the F-Droid directory[9] and hosted on GitHub[10]. We started with a random dataset of 1,103 repositories. We then followed the guidelines proposed by Kalliamvakou et al. [61] to avoid "non-real apps." For instance, we identified active mobile apps by considering projects that had a reasonable lifespan and number of commits, stars, forks, issues, and committers. We also removed repositories with no lines of Android code (typically, these projects are implemented in non-programming languages, like CSS or HTML). The final selection comprises 619 repositories. The final dataset can be found in our online appendix [51].

[9] `https://f-droid.org/`
[10] `https://github.com`

Table 5: Descriptive statistics summarizing the selected mobile apps. LI=Lifespan, CO=Commits, COn=Contributors, ST=Stars, FO=Forks, IS=Issues.

	Overview of the selected mobile apps							
	Line of Code		**GitHub's Metrics**					
	Java	**XML**	**LI**[†]	**#CO**	**COn**	**ST**	**FO**	**IS**
Max	180,407	154,582	3,340	45,920	295	17,578	7,246	7,341
Min	65	40	52	2	0	0	0	0
Trimmed mean	5772.75	6009.73	1725.7	262.6	5.0	64.0	28.1	48.7
Median	3759	1362	1786	129	3	32	15	22
Std Dev	20910.95	24423.89	723.4	2390.4	21.2	854.5	379.5	579.8
MAD[‡]	4477.45	1622.70	668.6	163.1	2.9	40.0	19.2	29.6

[‡]MAD stands for median absolute deviation. [†]Lifespan is presented in days.

Table 6: Three largest apps and the three smallest apps in the sample (in LoC).

	Project	**Java Files/LOC**	**XML Files/LOC**	**Category**
Largest	OsmAnd	614/175,902	861/154,582	Maps
	GreenBits Wallet	602/180,407	93/7,290	Finance
	Open Explorer	929/130,231	50/4,500	Productivity
Smallest	IcsImport	1/65	3/44	Productivity
	FlashLight	2/77	3/40	Tools
	BMI Calculator	1/93	23/273	Health

Table 5 shows descriptive statistics about the lifespan (in days), number of commits, size (number of *.`Java` files and number of *.`XML` files), and number of contributors, stars, forks, and issues of the selected repositories. For each metric, we report median, trimmed mean, median absolute deviation (MAD), and standard deviation (SD). On average, the apps have around 3,759 lines of java code and 1,363 lines of XML code. Most of the apps have up to 5 KLoC: 395 apps, which accounts for 59.6% of our sample. Approximately 40% of the analyzed apps have more than 5 KLoC (268 projects).

Table 6 presents the three largest apps and the three smallest apps (in LOC). The three smallest apps in our sample are: (i) IcsImport - imports events from calendars, (ii) FlashLight - uses the device as a flashlight, and (iii) BMI Calculator - computes the body mass index (BMI). The three largest projects are: (i) OsmAnd - provides offline access to maps from OpenStreetMap, (ii) GreenBits Wallet - a bitcoin wallet, and (iii) OpenExplorer - helps to manage files from the device.

Since an app project's age might indicate the app's maturity, we also investigated the lifespan of the projects. In the context of our study, a project's lifespan represents the time (in days) since the project's repository was cre-

ated on GitHub. The most mature analyzed repository has existed for 3,340 days – approximately nine and a half years. The least mature repository has 52 days (see Table 5). On average, the selected apps have been developed and maintained for 1,786 days – almost five years.

Another indicator of project maturity is the number of commits to a project repository. The selected apps have on average 129 commits (max = 45,920 commits), with an average of 53.47 commits in the last six months.

GitHub allows its users to "star" projects to show appreciation. Starring in GitHub can be seen as the equivalent of "liking" in other social media platforms. Borges and Valente [62] report that 73% of developers consider the number of stars before using or contributing to GitHub projects. The selected apps have on average 32 stars (max = 17,578 stars).

Forks and issues also indicate potential contributors to the repository [62]. Our sample has a median of 15 forks (Table 5), while 31 projects have never been forked.

7.3 Threshold tuning

In this section, we report how we defined the thresholds used in the detection strategies. We use quantile analysis, similar to what has been done in previous code smells literature [9, 63]. More specifically, we define the threshold as the third quantile plus 1.5 times the inter-quartile range:

$$TS = 3Q + 1.5 \times IQR \tag{1}$$

Table 7: Thresholds used in the detection strategies

Smell	Threshold
God Style Resource	$\alpha = 11$
Deep Nested Layout	$\alpha = 4$
Excessive Use of Fragments	$\alpha = 10$
Brain UI Component	$\alpha = 56$ and $\beta = 9$
Flex Adapter	α =56

The thresholds used in this paper were derived from 200 random apps from our dataset. Table 7 depicts the thresholds obtained for each smell.

7.4 Accuracy of the detection strategies

The ability of our heuristics to automatically detect code smells is intrinsically correlated with the validity of our results. In this sub-section, we discuss the accuracy of our detection strategies. Our smells can be divided into three groups based on their detection strategies:

- **Group 1 (Decidable):** Some of our smells can be detected via decidable, unambiguous, rules. In our case, the smells No Use of Fragments, Duplicate Style Attributes, Hidden Listener, Magic Resource, God String Resource, and Missing Image can be detected via straightforward static analysis. For example, in the case of No Use of Fragments, our tool detects whether FRAGMENTS are present or not in the system.
- **Group 2 (Decidable, threshold-based):** Some smells can also be detected via decidable rules, but they depend on a threshold. This is the case for the God Style Resource, Deep Nested Layout, and Excessive Use of Fragments smells.
- **Group 3 (Heuristic-based):** Other smells do not have decidable rules and require a heuristic (i.e., an approximation) for the detection. This is the case for Brain UI Component, Coupled UI Component, Suspicious Behavior, Flex Adapter, Fool Adapter, and UI Component Doing I/O.

We use software testing to evaluate whether our tool is adequately implementing the detection strategies from Groups 1 and 2. Since the strategies in Group 2 depend on a specific threshold to consider a class smelly, and we use extreme values as thresholds, some smelly classes may not identified. We followed this approach to reduce false positives (at the expense of false negatives). Consequently, the numbers we report for Group 2 smells might underestimate the real amount of smells.

Given the nature of the strategies in Group 3, further validation is required. To measure the accuracy of these detection strategies, we manually produced an oracle and compared the results of the tool against it. To build the oracle, two authors of this paper inspected the entire source code of three apps in our dataset and identified the smells (or the lack thereof) that each class contained. Our selection procedure was as follows:

1. We looked at the aggregated number of smells of each app in our dataset; we used this information to help us find a minimum set of apps that would have all the smells we proposed,
2. We filtered out apps with less than 50 Java classes or with more than 200 classes. The numbers 50 and 200 were chosen arbitrarily; we considered apps with less than 50 classes as too small and apps with more than 200 classes as too expensive for manual analysis.
3. We selected the largest app with the highest diversity in smells. We then repeated the process on the remaining smells.

The selected apps are: *seadroid*, an Android client for Seafile; *tasks*, an app that helps users to organize their tasks; and *yaaic*, an IRC client. *seadroid* has 228 classes; *tasks*, 151 classes; and *yaaic*, 88 classes. Our entire oracle comprises 467 Java classes.

In Table 8, we show the precision, recall, and F1 measures for each of the six smells from Group 3. We observe from these results:

1. The Brain UI Component and Flex Adapter detection strategies achieve a high precision and recall (their F1 measures are 0.93 and 0.86, respectively).

Smell	Precision	Recall	F1	TP	TN	FP	FN
Brain UI Component	0.91	0.94	0.93	93	360	9	5
Coupled UI Component	0.84	0.68	0.75	32	414	6	15
Suspicious Behavior	0.57	0.95	0.71	65	350	49	3
Flex Adapter	0.76	1.00	0.86	30	428	9	0
Fool Adapter	1.00	0.26	0.42	4	452	0	11
UI Component Doing I/O	0.94	0.6	0.73	18	436	1	12

Table 8: Precision ($tp/(tp+fp)$), recall ($tp/(tp+fn)$), and F1 ($2*(precision*recall)/(precision+recall)$) of our detection strategies. N=3 systems, 467 Java classes. The true positive (TP), true negative (TN), false positive (FP), and false negative (FN) columns show the concrete number of instances in each category, used to calculate the precision, recall, and F1.

2. The Suspicious Behavior detection strategy achieves a high recall (0.95), but its precision is just acceptable (0.57). After manual analysis in the false positives, we observed that some classes made use of inner and anonymous classes which are not related to event handling (note that out Suspicious Behavior detection strategy looks for any usage of inner and/or anonymous classes inside Activities, Fragments, or Adapters). Future work should focus on a more precise way of detecting event handling (e.g., take the semantics of the inner/anonymous class into account). 3. The Coupled UI Component and the UI Component Doing I/O detection strategies achieve high F1 measures (0.75, 0.71, and 0.73, respectively). However, while their precision are high (0.84 and 0.94, respectively), their recall are just acceptable (0.68 and 0.6, respectively). After manual analysis, we observed that, for the UI Component Doing I/O detection, our tool makes use of a pre-defined list of APIs that handle I/O (the list is available in our appendix). The false negatives made use of APIs other than the ones in our list. To improve the effectiveness of the detection strategy, we thus suggest the development of a systematic list of Android APIs that make use of I/O. 4. Finally, the Fool Adapter detection strategy present less accuracy: maximum precision (1.0), but low recall (0.26), mostly because of the existence of 11 false negatives. After manual analysis, we noticed our parser failing in case developers pass the Android's `View` class as a parameter to another method, and then invoke the `findViewById()` method (used in the detection strategy). Future work should systematically explore all the ways a developer might make use of the `findViewById()` method, with the goal of refining the parsing strategy.

The manually produced oracle, the complete source code of the apps we used, as well as the script that measures the precision and recall of our tool are available in our online appendix [51].

Table 9: Prevalence of the proposed smells in a sample of 619 Android apps. Percentages are calculated over the total number of Java and XML files analyzed in the 619 Android apps (37,026 Java files and 32,888 XML files.)

Smell	# of Java/XML files	%
Components		
Suspicious Behavior	8,584	≈23%
Brain UI Component	6,697	≈18%
Coupled UI Component	1,906	≈5%
UI Component Doing I/O	810	≈2%
No Use of Fragments	292	≈0.78%
Fool Adapter	187	≈0.50%
Excessive Use of Fragments	87	≈0.23%
Flex Adapter	70	≈0.18%
Total of affected components	18,633	
Resources		
God String Resource	8,581	≈26%
God Style Resource	8,528	≈25%
Deep Nested Layout	7,856	≈23%
Magic Resource	1,093	≈3%
Duplicate Style Attributes	208	≈0.63%
Hidden Listener	195	≈0.59%
Missing Image	48	≈0.14%
Total of affected resources	26,509	

7.5 Results

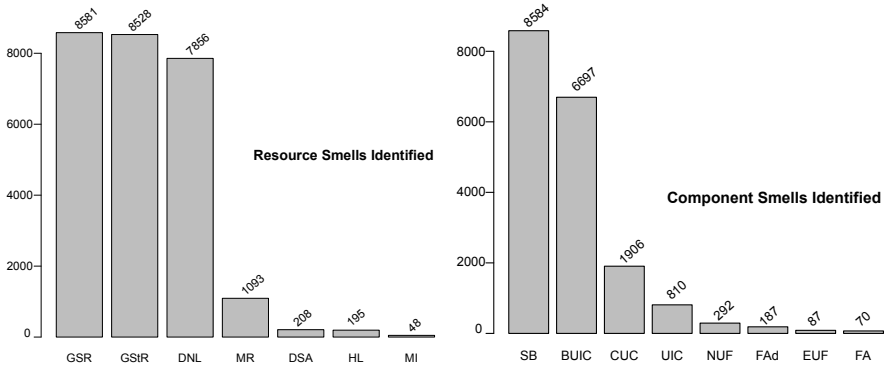
In Table 9, we present the smells identified in the 619 Android apps. The bar chart depicted in Figure 4 presents a macro view of the identified smells.

We found 26,509 instances of resource smells and 18,633 instances of component smells. God String Resource and Suspicious Behavior were the most common smells, with 8,581 resources (26% of all XML files in the sample) and 8,584 components affected (23% of all Java files in the sample), respectively. On the other hand, Missing Image and Flex Adapter were the least identified smells, with 48 resources and 70 affected components, respectively.

In Tables 10 and 11, we show the distribution of each code smell per app.

Although we observe that some projects have a critically high number of smells (e.g., a single project has 153 classes affected by the Brain UI Component smell), having a high number of classes affected by specific smells is not the common behavior. The median number (as well as the third quantile) of classes affected per project is quite low for all the studied smells. God String Resource is the one with the highest median (14).

RQ$_3$. All the proposed smells can be observed in real-world Android apps. At the project-level, the number of classes affected by each smell is low.



(a) Resource Smells identified. (b) Component Smells identified.

Fig. 4: Identified smells. DNL=Deep Nested Layout, DSA=Duplicate Style Attributes, GStR=God Style Resource, HL=Hidden Listener, MR=Magic Resource, GSR=God String Resource, MI=Missing Image. BUIC=Brain UI Component, UIC=UI Component Doing I/O, SB=Suspicious Behavior, EUF=Excessive Use of Fragments, CUC=Coupled UI Component, NUF=No Use of Fragments, FAd=Fool Adapter, FA=Flex Adapter

	BUIC	UIC	SB	CUC	FAd	FA
1st Qu.	1	0	2	0	0	0
Median	4	0	5	0	0	0
Mean	11	1	14	3	0.3	0.11
3rd Qu.	12	1	14	2	0	0
Max.	153	33	279	78	11	6

Table 10: Prevalence of the Component smells per app. BUIC=Brain UI Component, UIC=UI Component Doing I/O, SB=Suspicious Behavior, EUF=Excessive Use of Fragments, CUC=Coupled UI Component, NUF=No Use of Fragments, FAd=Fool Adapter, FA=Flex Adapter.

8 Discussion

In this section, we discuss the main implications of our work.

A catalog of code smells for the presentation layer of Android apps. Developing high-quality user interface code for Android apps is challenging. Developers should make good use of the limited screen space and cannot take advantage of the full range of features that a traditional web application provides [64]. Our work paves the road for a catalog of bad practices that arise in such an important part of the source code. When it comes to the presentation layer, smells not only occur in classes, such as *Activities* and *Fragments*, but also in resources, such as *layout* and *strings*, which are mostly XML-based. It is common for developers to see XML files as "simple" configuration files;

	DNL	DSA	GStR	HL	MR	GSR
1st Qu.	2	0	2	0	0	1
Median	5	0	4	0	0	2
Mean	13	0.33	14	0.31	2	14
3rd Qu.	15	0	10	0	2	9
Max.	219	13	446	11	48	454

Table 11: Prevalence of Resource smells per app. DNL=Deep Nested Layout, DSA=Duplicate Style Attributes, GStR=God Style Resource, HL=Hidden Listener, MR=Magic Resource, GSR=God String Resource, MI=Missing Image.

however, they play a key role in Android apps, and their quality should also be monitored.

In this work, we proposed smells that go beyond the "traditional smells" in the literature. In a more abstract way, one can see how foundational concepts in object-oriented design can help developers tackle smells. For example, a class affected by the Brain UI Component smell would benefit from a better *separation of concerns* [65], and classes affected by the Coupled UI Component smell would benefit from better *dependency management* [66]. Moreover, we observe that developers considered both the intensive use and lack of use of FRAGMENTS problematic. Interestingly, the set of participants who reported the intensive use of FRAGMENTS as problematic differs from the set of participants who saw the lack of FRAGMENTS as problematic (see Table 3). Deciding whether to use small or large fragments is a similar problem as deciding how to *modularize* a software system [65]; in practice, it is hard to know when to stop creating more modules. Thus, as a recommendation, we suggest developers should understand foundational OO practices deeply, as some of the smells can be explained by what is already in the OO literature.

The relationship between our study and the existing body of knowledge. As we present in the Related Work (Section 3), different authors have proposed catalogs of code smells and/or best practices for Android mobile applications, e.g., Palomba et al. [47] proposed a generic catalog of smells, Gharafi et al. [48], of security code smells, Hecht et al. [43] and Linares-Vásquez et al. [44] of performance, and Linares-Vásquez et al. [42] and Gottschalk et al. [15] of energy consumption smells.

Our research complements the existing body of knowledge in the following ways:

– Two security smells from Ghafari et al.'s catalog are related to the presentation layer of Android apps. More specifically, *Broken WebView's Sandbox*, which is relevant to developers rendering web content in an unsafe manner, and *SQL Injection*, which commonly happens when user input goes straight to an SQL query. This result shows that issues in the presentation layer can also lead to security flaws. Therefore, as future work, we sug-

gest researchers study the relationship between the presentation layer code smells and security vulnerabilities.
- Palomba et al. [47]'s code smells catalog does not touch on any presentation layer code smells, and thus, our catalog is complementary to it. However, UI developers should be aware of the *Leaking Thread* code smell proposed in their catalog, as most of what happens in presentation layers occurs in threads (we discuss the life cycle of the components in Section 2).
- Linares-Vásquez et al.'s [42] showed that UI-related APIs (GUI and image manipulation) represent around 40% of the energy greedy APIs in the Android platform. As actionable advice, authors suggest developers carefully design apps that make use of several views and to avoid refreshing views. We see their results as a complementary to ours. Our catalogue has several smells related to complex UIs (i.e., Brain UI Component, UI Component Doing I/O, Deep Nested Layout, and Unnecessary Image). Besides being harmful for maintenance, we conjecture that these smells also impact energy consumption, and therefore suggest developers consider not only the maintenance cost but also the energy costs of classes affected by these smells.
- While performance studies by Hecht et al. [43], Linares-Vásquez et al. [44], and Liu et al. [45] lacked focus on the presentation layer, our smells can be related to performance issues. Linares-Vásquez et al.'s study, in particular, showed that unused strings (and other resources) can be costly to mobile apps. Our catalog indeed has smells related to how developers organize their resources in their software (i.e., God String Resource, Inappropriate String Reuse, Duplicate Style Attributes, God Style Resource, Long or Repeated Layout, Deep Nested Layout). Thus, we suggest developers also consider elements affected by resource-related smells as candidates for performance improvements. In future work, we suggest researchers investigate the relationship between our smells and performance issues.
- Companies and independent developers have been working on guides and best practices catalogs that go beyond "traditional smells," such as Google's Jetpack Best Practices [67,68] and Futurice, a software house which hosts a GitHub repository on Android best practices with around 17,000 stars [59]. Our catalog complements this effort.
- In our research, we focused on smells related to the presentation layer of Android apps. Nevertheless, we noticed that many of our participants often mentioned "traditional" smells, such as Long Methods and God Classes [3, 63] as problems they also face in this layer. As we show in the Related Work section, researchers have also investigated the role of traditional smells in Android apps. Therefore, when developing the presentation layer, we recommend developers be aware of both traditional and presentation layer-specific smells.

Smells in different Mobile platforms. We acknowledge that some of our smells may become less important to practitioners over time. For example, the Flex Adapter smell, although considered an important smell for developers

to tackle, was less perceived in practice. We conjecture this is due to the Android's new Adapter component, the `RecyclerView.Adapter`, which appeared in Android 5.1 Lollipop and facilitates the implementation of the ViewHolder pattern. Before that, developers had to implement the pattern themselves, which required previous knowledge about best practices. We hope that our results can inspire new tools, strategies, and modifications to the underlying technology to make the mitigation strategies easier to implement.

It is also important to notice that our current catalog solely focuses on Android mobile apps. Since its launching in 2008, Android native apps have been developed using the Java language. In May 2017 (after we started this research), Google announced Kotlin as the official language for the platform [69]. Although research [70] shows that Kotlin leads to more concise and clearer code and tends to contain less "traditional code smells" when compared to Java, we argue that the Android framework is still the same. In other words, developers still need to write Activities, Listeners, and all the other components, as well as *resources*. Thus, we see our smells as important for Kotlin-based Android apps as well.

Moreover, although this catalog cannot be directly transported to iOS (Swift and Objective C), Windows Phone, or Xamarin development, it can serve as inspiration for future research on these platforms. Interestingly, previous research [71] has shown that iOS apps contain the same proportions of code smells regardless of the development language (Swift and Objective C), but they seem to be less prone to code smells compared to Android apps. Thus, understanding whether our smells impact iOS apps as much as they do Android apps seems to be the natural continuation of this research.

9 Threats to Validity

Internal validity. Threats to internal validity concern how external factors that we do not consider can affect the variables and relationships investigated. In the literature, code smells are derived from the empirical knowledge of experienced developers [1, 3, 4, 25]. Research also showed that experience and knowledge play an important role in the perception of code smells [31, 72]. We removed from our analysis answers to our questionnaires (parts 1 and 2) from developers with no experience with Android development; most of our respondents have two or more years of experience.

In addition, as the participation was anonymous, we did not control for developers who participated in the multiple steps of our study. However, as all the parts of our research have different goals, and Part 2 had many more participants than Part 1, this threat has a limited effect. Nevertheless, we propose replications of this work as a way to strengthen our findings.

The coding analysis in Part I was conducted by the first author of this paper, and the second author acted as a mediator whenever a question arose (as explained in Section 5). At the time of the analysis, the first author had five years of industry experience with Android development. To improve the valid-

ity of the identification of the smells, in later stages of our research we revisited the proposed smells with external Android experts, and they all agreed with the proposed smells (RQ2 methodology, Section 6.1). Moreover, the fact that many developers also face these smells (RQ2 results, Section 6.3) is also an indication of their validity. However, we acknowledge that different researchers could interpret the data in different ways (and thus derive a different set of smells). We make all our raw data available in our online appendix [51] to enable researchers to further validate our work and make additional analyses.

Regarding the study on the prevalence of code smells in mobile applications, we set out to mitigate the selection bias issue by using randomization. However, no blocking factor was applied to reduce the threat of possible variations in, for instance, the complexity of the apps, usability, and performance. Thus, we cannot rule out the possibility that the chosen apps stem from other quality factors as opposed to the amount of code smells.

Construction validity. Threats to the *construction validity* concern the relationship between theory and observation. The questionnaire in Part 2 aimed at measuring developers' perceptions on the frequency and importance of the code smells. Thus, the questions were derived from our catalog of code smells. In retrospect, we noticed that we had not added any control questions that would help us measure biases from our participants. Nevertheless, while the number of answers we collected in this survey is quite significant (201 responses) and one could argue that biased answers would be a minority, we suggest the replication of this survey as future work.

Finally, in Part 3, given that all code smells have been identified by a tool, it is possible that some data is incorrect due to misguided or ill-identified code smells. Thus, our data might not reflect the actual amount of code smells (given to possible false negatives and false positives). We nevertheless discuss the accuracy of our tool in Section 7.4. As we show there, the proposed detection strategies can be improved, and we leave it as future work.

External validity. Threats to *external validity* refer to the generalization of our results. We define the presentation layer as the eight elements we show in Section 2. Although this definition has been based on official documentation, we acknowledge that there are other resources and there may be less commonly used classes that also relate to the presentation layer. Therefore, we do not claim that the presentation layer is limited to the eight elements studied here.

In addition, the first part of our research aimed at devising the catalog of smells. In the end, we obtained 359 pieces of information from 45 different (mostly Brazilian) software developers. We were able to derive 20 code smells that were observed by more than 5 participants. However, we do not claim that this catalog is complete; as an example, if we reduce the number of required repetitions (e.g., from 5 to 4), we would have a broader set of code smells. In practice, we indeed expect this catalog to continue to expand as the Android framework keeps evolving. We expect researchers to join forces and use our proposed methodology to continuously collect the perceptions of developers on new code smells.

Regarding the study on the prevalence of code smells in mobile applications, the sample we collected from the open source repository might not be representative of the target population. As aforementioned, we randomly selected apps from the F-Droid repository. However, our set is diverse and includes active and largely used projects. Replications are encouraged to explore the smells in industrial settings.

10 Conclusion

In this paper, we propose a catalog of 20 code smells specific to the presentation layer of Android apps, employing two online questionnaires and a study with real projects. Our results show that developers are aware of good and bad practices specific to the Android platform. From the reported practices, we devised the smells, which were validated with a second questionnaire and analysis of the source code of real projects. The proposed smells are particularly relevant, as Android became the world's leading mobile platform in 2011 and since then has increased its share of the market, having reached 86% [73] in 2017.

This study answered the following questions:

RQ1: What code smells do developers observe in the presentation layer of Android apps? We cataloged 20 code smells in the presentation layer of Android apps, 9 related to components (Activities, Fragments, Adapters, and Listeners), and 11 related to resources (Layout, String, Style, and Drawable resources). The complete catalog can be found in Section 5.4.

RQ2: How often do developers observe the identified code smells and what importance do they give to them? Developers perceive most of the proposed smells as important, and most of them have previously encountered these code smells. Their perceptions are discussed in Section 6 of this paper.

RQ3: How prevalent are the proposed code smells in real Android apps? All the proposed smells can be observed in real-world Android apps. Some of them, such as Brain UI Component (29% of all components) happen very often, whereas others, such as Flex Adapter and Duplicate Style Attributes happen less often.

Our contributions are a small but important step in the search for higher code quality on the Android platform. Researchers can use our results as a starting point to conceive tools and heuristics to suggest refactorings in Android applications, and Android developers can use our catalog to search for problematic pieces of codes.

Our work opens space for future research and tool development. More specifically:

Evaluate the effects of the proposed code smells in other contexts. In this paper, we collected diverse empirical evidence about the relevance of the smells. Further investigation is necessary about the impacts the smells bring to software developers. Some suggestions regarding future evaluations are: 1)

controlled experiments about the relationship between the proposed smells and maintainability and other development activities (e.g., do developers take more time to comprehend and maintain a smelly class when compared to a clean class?), 2) to quantitatively measure how these presentation-layer smells affect the change- and the defect-proneness of the smelly classes, 3) understand whether these code smells can have an impact on different quality attributes of a mobile app, such as performance and energy consumption.

Generalizability of our results to other mobile platforms. Although Android has a significant market share of mobile development, it is not the only one; iOS (Apple phones) and Windows phones are also popular platforms. The reasons we focused on Android are twofold: First, our research method requires in-depth knowledge of the platform. The authors of this paper are well versed on the Android platform, but not so on the other platforms. Second, by focusing on a single mobile architecture, we could ask highly focused questions to our participants, which we argue increases the quality of the answers. Nevertheless, coining code smells for other mobile platforms is also important, and our study can be replicated to other platforms.

References

1. R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1 ed., 2008.
2. G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: Managing technical debt*. Morgan Kaufmann, 2014.
3. M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
4. B. Webster F, *Pitfalls of Object-Oriented Development*. M & T Books, 1995.
5. M. Aniche, J. Yoder, and F. Kon, "Current challenges in practical object-oriented software design," in *41st ACM/IEEE International Conference on Software Engineering*, (United States), IEEE, 2019.
6. G. Gharachorlu, *Code smells in Cascading Style Sheets: an empirical study and a predictive model*. PhD thesis, University of British Columbia, 2014.
7. A. M. Fard and A. Mesbah, "JSNOSE: Detecting javascript code smells," pp. 116–125, 2013.
8. M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, "Code smells for model-view-controller architectures," *Empirical Software Engineering*, pp. 1–37, 9 2017.
9. M. Aniche, G. Bavota, C. Treude, A. Van Deursen, and M. A. Gerosa, "A validated set of smells in model-view-controller architectures," pp. 233–243, 2016.
10. M. Pinzger, F. Hermans, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, (Washington, DC, USA), pp. 409–418, IEEE Computer Society, 2012.
11. O. H. Alliance, "Open handset alliance releases android SDK." https://www.openhandsetalliance.com/press_111207.html, 2007. [Last access: 25 de Novembro de 2017].
12. G. Hecht, "An approach to detect android antipatterns," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 766–768, IEEE Press, May 2015.
13. 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," pp. 232–243, 2014.

14. D. Verloop, *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013.
15. M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter, "Removing energy code smells with reengineering services," *GI-Jahrestagung*, vol. 208, pp. 441–455, 2012.
16. J. Reimann and M. Brylski, "A tool-supported quality smell catalogue for android developers," 2014.
17. R. Minelli and M. Lanza, "Software analytics for mobile applications, insights & lessons learned," *In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, 2013.
18. Google, "Android – plataform architecture." `https://developer.android.com/guide/platform/index.html`. [Last access: 25 de Novembro de 2017].
19. Wikipedia, "IOS — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=IOS&oldid=812046680`, 2017. [Last access: 25 de Novembro de 2017].
20. OpenSignal, "Android fragmentation visualized." `http://opensignal.com/reports/2015/08/android-fragmentation`, 2015. [Last access: 25 de Novembro de 2017].
21. Google, "Documentação site android developer." `https://developer.android.com`, 2016. [Last access: 25 de Novembro de 2017].
22. Google, "Android – fundamentals." `https://developer.android.com/guide/components/fundamentals.html`, 2017. [Last access: 25 de Novembro de 2017].
23. Google, "Android – resource type." `https://developer.android.com/guide/topics/resources/available-resources.html`, 2016. [Last access: 25 de Novembro de 2017].
24. Google, "Android – building your first app." `https://developer.android.com/training/basics/firstapp/creating-project.html`, 2016. [Last access: 25 de Novembro de 2017].
25. A. J. Riel, *Object-Oriented Design Heuristics*, vol. 335. Addison-Wesley Publishing Company, 1996.
26. F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, (Washington, DC, USA), IEEE Computer Society, 2009.
27. F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Softw. Engg.*, vol. 17, June 2012.
28. W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.
29. A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, ICSE '13, (Piscataway, NJ, USA), pp. 682–691, IEEE Press, 2013.
30. 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 *Software maintenance and reengineering (CSMR), 2011 15th European conference on*, pp. 181–190, IEEE, 2011.
31. F. Palomba, G. Bavota, M. Penta, R. Oliveto, and A. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," pp. 101–110, 2014.
32. R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 33–36, ACM, 2011.
33. R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 411–416, IEEE, 2012.
34. T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," pp. 1001–1012, 2014.
35. M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. A. Gerosa, "SATT: Tailoring code metric thresholds for different software architectures," in *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, pp. 41–50, IEEE, 2016.

36. U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*, pp. 225–236, IEEE, 2016.
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, 2010.
38. G. Hecht, R. Rouvoy, N. Moha, and L. Duchien, "Detecting antipatterns in android apps," in *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pp. 148–149, May 2015.
39. G. Hecht, R. Rouvoy, N. Moha, and L. Duchien, "Páprika." `https://github.com/geoffreyhecht/paprika`, 2015. Last access on April, 2018.
40. 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.
41. A. S. A. Peruma, "What the smell? an empirical investigation on the distribution and severity of test smells in open source android applications," 2018.
42. M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: an empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 2–11, ACM, 2014.
43. G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pp. 59–69, ACM, 2016.
44. M. Linares-Vásquez, C. Vendome, M. Tufano, and D. Poshyvanyk, "How developers micro-optimize android apps," *Journal of Systems and Software*, vol. 130, pp. 1–23, 2017.
45. Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 1013–1024, ACM, 2014.
46. M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in android apps," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 352–361, IEEE, 2015.
47. F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 487–491, IEEE, 2017.
48. M. Ghafari, P. Gadient, and O. Nierstrasz, "Security smells in android," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 121–130, IEEE, 2017.
49. A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 242–251, IEEE, 2013.
50. E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *In Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pp. 97–106, IEEE Computer Society, 2002.
51. "An empirical catalog of code smells for the presentation layer of android apps: Appendix." `https://doi.org/10.5281/zenodo.3256367`, 2019.
52. J. Corbin and A. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* SAGE Publications Ltd, 3 ed., 2007.
53. B. G. Glaser and A. L. Strauss, *Discovery of grounded theory: Strategies for qualitative research.* Routledge, 2017.
54. S. Adolph, W. Hall, and P. Kruchten, "Using grounded theory to study the experience of software development," *Empirical Software Engineering*, vol. 16, no. 4, pp. 487–513, 2011.
55. J. Saldaña, *The Coding Manual for Qualitative Researchers.* SAGE Publications Ltd, 2 ed., 2015.

56. J. Nielsen, "Why you only need to test with 5 users." `https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users`, 2000. [Last access: 25 de Novembro de 2017].
57. E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
58. Google, "Android – optimizing view hierarchies." `https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies.html`, 2017. [Last access: 25 de Novembro de 2017].
59. Futurice, "Android best practices." https://github.com/futurice/android-best-practices, 2018. Last accessed on October 29th, 2018.
60. Google, "Android – recyclerview." `https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html`, 2017. [Last access: 25 de Novembro de 2017].
61. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, and D. Damian, "The Promises and Perils of Mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositoriesa (MSR)*, pp. 92–101, ACM, 2014.
62. H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.
63. 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.
64. A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pp. 397–400, ACM, 2010.
65. G. Booch, *Object oriented analysis & design with application*. Pearson Education India, 2006.
66. R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.
67. Google, "Guide to app architecture." https://developer.android.com/jetpack/docs/guide, 2018. Last accessed on October 29th, 2018.
68. Google, "Optimizing layout hierarchies." https://developer.android.com/training/improving-layouts/optimizing-layout, 2018. Last accessed on October 29th, 2018.
69. "Kotlin on android. now official." `https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/`, 2017. JetBrains blog.
70. M. Flauzino, J. Veríssimo, R. Terra, E. Cirilo, V. H. S. Durelli, and R. S. Durelli, "Are you still smelling it?: A comparative study between java and kotlin language," in *Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse*, pp. 23–32, ACM, 2018.
71. S. Habchi, G. Hecht, R. Rouvoy, and N. Moha, "Code smells in ios apps: How do they compare to android?," in *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pp. 110–121, IEEE, 2017.
72. D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223–235, 2017.
73. Statista, "Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2017." `https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems`, 2017. [Last access: 25 de Novembro de 2017].