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

An empirical study of Android behavioural code smells detection

Dimitri Prestat[1] · **Naouel Moha**[1,2] · **Roger Villemaire**[1]

Received: date / Accepted: date

Abstract Mobile applications (apps) are developed quickly and evolve continuously. Each development iteration may introduce poor design choices, and therefore produce code smells. Code smells complexify source code and may impede the evolution and performance of mobile apps. In addition to common object-oriented code smells, mobile apps have their own code smells because of their limitations and constraints on resources like memory, performance and energy consumption. Some of these mobile-specific smells are behavioural because they describe an inappropriate behaviour that may negatively impact software quality. Many tools exist to detect code smells in mobile apps, based specifically on static analysis techniques. In this paper, we are especially interested in two tools: Paprika and aDoctor. Both tools use representative techniques from the literature and contain behavioural code smells. We analyse the effectiveness of behavioural code smells detection in practice within the tools of concern by performing an empirical study of code smells detected in apps. This empirical study aims to answer two research questions. First, are the detection tools effective in detecting behavioural code smells? Second, are the behavioural code smells detected by the tools consistent with their original literal definition?

We emphasise the limitations of detection using only static techniques and the lessons learned from our empirical study. This study shows that established static analysis methods deemed to be effective for code smells detection are inadequate for behavioural mobile code smells detection.

Keywords Android · code smells · detection · empirical study · mobile apps · behavioural

Dimitri Prestat
prestat.dimitri@courrier.uqam.ca
[1] Université du Québec à Montréal, Montréal, Canada
[2] École de Technologie Supérieure, Montréal, Canada

1 Introduction

Mobile apps are becoming complex software systems that have evolved and grown increasingly through the years. In 2020, there were more than five million apps available in various app stores [44], with more than 200 billion downloads in 2020 [45]. Apps must therefore become more and more efficient. They are thus developed quicker and evolve continuously to fit the new user requirements. These modifications are made by developers to solve bugs or add missing features in a constrained time frame, forcing them to adopt poor design or implementation choices, also known as code smells [10], which reinforce the technical debt.

Even if these mobile apps are mostly developed with Object-Oriented (OO) languages, and many questions on OO code smells have already been addressed in the literature [30] [34], mobile apps bring new concerns involving, for instance, screen sizes, energy consumption, limited memory and limited performance [29]. To address these resource limitations, the research community introduced new Android-specific code smells, describing them carefully to detect and correct them [12] [20] [38].

We define a behavioural code smell as characteristics in the source code inducing an inappropriate code behaviour during the execution that may negatively impact software quality in terms of performance, energy consumption, memory. The term "behaviour" refers specifically to code execution behaviour, i.e. an occurrence or a sequence of observable code events or actions during execution.

For example, the code smell Durable WakeLock (DW) manifests when the lock of a WakeLock is not released, causing battery drain. The WakeLock is the mechanism allowing an app to keep the device on. The DW code smell describes the following inappropriate behaviour: A call to the *acquire* method is not followed by a call of the *release* method. In this case, the source code characteristics are the invocations of the *acquire* and *release* methods of the *WakeLock* class. As another example, the code smell HashMap Usage (HMU) indicates that a *HashMap* structure should be used for large sets of objects and *SimpleArrayMap*/*ArrayMap* should be used for small sets of objects to be more memory-efficient. The HMU code smell describes the following inappropriate behaviour: The *HashMap* structure is used for small sets of objects or *ArrayMap* / *SimpleArrayMap* structures are used for large sets of objects. In this case, the source code characteristics are the use of the *HashMap* / *ArrayMap* / *SimpleArrayMap* classes. Behavioural code smells are not explicitly mentioned or defined in the literature to the best of our knowledge. Also, following the empirical study we carried out in this work, we were able to identify three behavioural code smells categories. The first one is characterised by the misuse of a method call or a sequence of method calls during the execution. The code smell DW belongs to the first category. The second is characterised by runtime issues, such as a long execution time of a method or an excessive use of the memory. Finally, the third is characterised by undesired data variations during execution, such as the size of a structure becoming excessively

large. The code smell HMU belongs to the third category. This does not exclude that other categories may be added in the future if new behavioural code smells are introduced.

So far, several tools have been proposed for the detection of Android-specific code smells. In our study, we used a systematic procedure to collect and select the mobile code smells detection tools for consideration in this study, and we narrowed down to aDoctor [38] and Paprika [20]. This systematic procedure is detailed in Section 3.2.1.

aDoctor and Paprika are still in use, as shown by their recent extensions released to meet new requirements. For instance, a new version of aDoctor as an Eclipse plugin [21] has been released. A new extension of Paprika called Sniffer [18] has been developed to perform analysis and detection of code smells based on GitHub project commits. These tools are also the most widely cited and used in the literature within the available tools. Both tools use term search and metric computation techniques. These are the techniques most often used by recent Android code smells detection tools, as shown in the categorisation of tools by Rasool *et al.* [41]. Furthermore, to the best of our knowledge, there are no dynamic detection tools that allow the detection of Android code smells, in particular behavioural code smells, which might require dynamic analysis for their detection. These two tools are therefore representative of the existing tools for the detection of Android code smells.

We focus on behavioural code smells in mobile apps for two main reasons. First, they may hinder the software quality of mobile apps, and specifically in terms of energy consumption, memory and performance. Secondly, existing research has not specifically addressed their detection. Therefore, we want to assess if the current code smells detection tools are effective to identify such code smells.

In this paper, we study the effectiveness of behavioural code smells detection in practice as well as identifying the limitations of current detection techniques on the tools. To this end, we conduct an empirical study comparing the textual definitions of behavioural code smells as described in the literature against their detection techniques within the concerned tools. The empirical study also consists of investigating the issues of the detection rules defined in the tools to check whether the detection results are partial or erroneous. Concretely, we analysed 676 instances of seven behavioural code smells detected in 318 apps manually with the tools of concern.

More precisely, this empirical study aims to answer the following two research questions:

***RQ*$_1$:** Are the tools of concern effective for detecting behavioural code smells?

Finding: We found that the tools of concern return a significant part of false negatives, up to approximately 42% for some code smells, and false positives, up to approximately 73% for some code smells. False positives affect the effectiveness because erroneous detection results are returned to the developers. False negatives affect the effectiveness since in that case only partial detection results are returned to the developers.

***RQ*$_2$:** Are the behavioural code smells detected by the tools of concern consistent with their original literal definition?

Finding: We found that the detection rules do not and often cannot check the characteristics mentioned in the literal definition of the code smells. This therefore explains the prevalence of both false positives and false negatives.

This is the first empirical study of its kind studying the effectiveness and consistency of behavioural code smells detection in practice as well as identifying the limitations of current detection techniques.

To summarise, our first contribution is to provide an empirical study that compares the detection tools' results against the textual definitions of seven behavioural code smells described in the literature. Furthermore, our second contribution is a thorough analysis of the tools' detection techniques to identify their limitations and share the lessons learned from our empirical study.

This paper is organised as follows. Section II discusses the related work, Section III presents our empirical study and Section IV shares the lessons learned from our empirical study. We finally conclude the paper in Section V.

2 Related Work

In this section, we discuss the relevant literature about mobile apps' code smells analysis and detection.

2.1 Code Smells Analysis

Reimann *et al.* [42] propose a catalogue of 30 quality smells dedicated to Android. These code smells originate mainly from the good and bad practices documented online in Android documentation or by developers reporting their experience on blogs. These code smells concern various aspects like implementations, user interfaces or database usages. They are reported to have a negative impact on properties, such as efficiency, user experience or security. Paprika [20] and aDoctor [38] use some code smells originating from this catalogue. Unfortunately, this catalogue's website is no longer accessible.

Security smells are another category of smells focusing on the vulnerabilities on the mobile apps [12]. In this case, this paper identifies 28 smells whose presence may indicate a security issue in app, and presents a static analysis tool to study the prevalence of such smells. Moreover, symptoms and mitigation for each of these smells are described.

Mannan *et al.* [29] compares code smells in Android versus desktop applications. They examine these differences in terms of their variety, density and distribution of code smells. They found that the density of code smells is very similar, whereas the distribution and the variety are much superior on Android apps. They also shed light on the gap between the code smells that appear in practice and those studied in the literature.

It has been suggested that code smells are not only introduced by the least experienced developers on a project, but by any developers [17]. The removal

of these code smells does not seem to be affected by the developers' experience as well. Another study shows that code smells can stay in the code for years before being removed [18]. These two major points show that most projects are likely to be infested with smells, and detecting them to correct them is of utmost importance.

The qualitative study of Habchi *et al.* [15] investigated the perception of performance bad practices by Android developers. This study found that developers may lack interest and awareness about Android code smells. Moreover, the study showed that some developers challenge the relevance and impact of code smells in practice. Moreover, Johnson *et al.* [23] conducted 20 interviews to understand the lack of interests from developers to use static analysis tools to find bugs. One of the important conclusions of this study is that the false positives and warnings are the main barriers of tools adoption.

Malavolta *et al.* [28] study the frequency and evolution of maintainability issues of Android apps. These results show that the most recurrent maintainability issue is code duplication caused by the Android programming model. The maintainability issue density grows until it stabilises, and is usually seldom fully resolved. It also shows that the maintainability issues are independent from the type of development activity, so every type of app is involved.

2.2 Code Smells Detection Techniques

Multi-Objective Genetic Programming has also been used to detect Android smells [24]. This approach generates rules, which consist of a combination of quality metrics with their threshold values to detect a specific type of code smell. This approach takes as input a set of Android-specific code smell examples, and find the best set of rules to cover most of the expected Android code smells.

Gottschalk *et al.* [13] propose a re-engineering approach to detect energy code smells and to fix them through code refactoring. Although the authors propose a platform-agnostic detection technique together with a list of seven mobile energy code smells, they illustrate the applicability of their approach for the detection of Android code smells. They provide only a simplistic description of the code smells to be detected, but nothing is said on how the actual detection could be performed. The approach only focuses on code smells that can impact on energy consumption. They did not perform any analysis on the presence of such code smells in apps from any app stores or empirically demonstrate its importance on the energy consumption.

Sniffer [18] is an open-source toolkit that tracks the full history of Android-specific code smells. However, Sniffer is not another detection tool because it relies on Paprika to detect Android code smells. Paprika being slightly modified to be able to analyse the source code directly instead of the byte code.

Recently, Iannone *et al.* [21] proposed a new version of aDoctor, which helps developers refactor the smells automatically. This extended version is

open-source and available in Android Studio as a plugin published in the official store. The identification algorithms of the smells are the same as defined in aDoctor [38] previous version. In order to be able to recover a lot of data more easily, we use the other version of aDoctor, although we are aware that this extension has been released.

Several other tools are available to detect Android code smells. These tools perform the detection by different *static analysis* techniques. Such techniques can be mainly based on *patterns* or based on the computation of *source code metrics*, mainly OO code metrics. For instance, Ghafari *et al.* [12] developed a static analysis tool to study the prevalence of security smells. Furthermore, Rasool *et al.* [41] describe an approach that is able to recover 25 Android code smells by source code analysis and computes source code metrics.

EARMO [35] reported an approach able to detect and correct code smells related to energy consumption from mobile apps. This approach, when used to correct these smells, is able to extend the battery life considerably.

Emden and Moonen [8] developed jCosmo for Java code smells detection and defined two code smell aspects: Primitive smells aspects and Derived smells aspects. Primitive smells aspects can be observed directly from the code. Derived smell aspects are inferred from other aspects. They also suggested that some code smells need runtime information and thus dynamic analysis can be used. JSNose [9] is a Javascript code smells detection tool that combines static and dynamic analysis of the client-side code. It considers behaviour by monitoring the creation/update of functions, objects, and their properties at runtime. Feature Envy smell [25] has been detected dynamically by considering the actual execution performance instead of the static behaviour.

There is a lot of research that has focused on detecting code smells in mobile apps, which demonstrates the interest of this topic in the community. Some of these works have provided detection approaches and tools, including Paprika and aDoctor.

However, to the best of our knowledge, no study has specifically investigated the effectiveness of these tools and the relevance of the results returned by the tools conforming to their original definitions.

3 Empirical study

In this section, we present the results of our empirical study. The detection results of Paprika and aDoctor are presented and examined in detail to evaluate if these tools are effective to detect behavioural code smells and further whether detection results are consistent with the literal definition of these code smells. We follow Wohlin *et al.* guidelines [46] to describe this study.

3.1 Research Questions

The empirical study aims to respond to the following two research questions:

- ***RQ*$_1$: Are the tools of concern effective for detecting behavioural code smells?** We attempt to identify the extent to which the tools return false positives and false negatives. This is one of the main motivations of the paper. Some instances defined as having a code smell do not finally have a code smell and are therefore *false positives*. Having too many false positives impacts the effectiveness of the detection tools because the results are erroneous. Similarly, some instances defined as not having a code smell do finally have a code smell and are therefore *false negatives*. Having too many false negatives impacts the effectiveness of tools as the results are partial.

- ***RQ*$_2$: Are the behavioural code smells detected by the tools of concern consistent with their original literal definition?** This question aims to check if the detection rules specify well or not the literature definition of behavioural code smells. Detected code smells are inconsistent with their literal definition if elements of the definition are missing in the tool's detection rules. This assumes that there are common definitions for these code smells. This is indeed the case for the behavioural code smells considered in this paper, for which [19, 20, 38] all give the same textual definitions that vary only in the use of some synonyms. We hence followed these definitions with the single exception of the NLMR and IOD code smells, as we will explain in Section 3.2 on page 10.

To answer our research questions, we want to reject the null hypothesis formulated as:

- H_1^{posi}: There are no false positives of detected code smells returned by the tools.
- H_2^{nega}: There are no false negatives of detected code smells returned by the tools.
- H_3^{rule}: There is no difference between the detection rules and the literal definition of code smells.

3.2 Subjects

3.2.1 Tools

The study is based on two tools, Paprika and aDoctor. These two tools are easy to use and open-source. Both tools use static detection techniques and are fully automatic.

We used a systematic procedure to collect and select the mobile code smells detection tools for consideration in this study, and we narrowed down to aDoctor [38] and Paprika [20]. We first started with the list of mobile code smells detection tools provided by Rasool *et al.* [41]. To the best of our knowledge, this list is the most extensive and up-to-date list on mobile code smells detection tools. Indeed, we did not find any other work that reports

such list and we did not find any other tool that was not reported in this list. Rasool *et al.* reports 25 publications of mobile code smells detection and gives in each case the technique used for the detection (search based, metric based and symptom based), the tool that implements the technique and the availability of the tool (yes or no). This list includes 19 different tools for the 25 papers. Among these 19 tools, 5/19 are prototypes and are not available [22, 24, 35, 39, 43]. Among the 14/19 remaining tools, 4/14 are commercial or private and are not available [3, 7, 27, 29]. In the 10/14 tools left, 4/10 are extensions of aDoctor [2, 21] and Paprika [16, 18] and among the 6/10 remaining tools, 4/6 address categories of code smells (permission code smells [4], security code smells [11], unit test smells [40] and object-oriented code smells [26]) that are not behavioural, in contrast to the focus of this study. The last 2/6 remaining tools are aDoctor [38] and Paprika [20], and we have therefore chosen to focus on these two tools.

A major difference between these tools is that Paprika is much more suitable for large-scale analysis. First of all, in terms of analysis time, Paprika is much faster, and especially useful when we aim to analyse a large number of apps. While Paprika with the extension Sniffer only expects a list of code source folders, aDoctor needs to run a different execution for each app and for each code smell.

Although both are essentially static in detection, aDoctor simply makes use of regular expressions and string search. Paprika, on the other hand, uses queries on graphs, and thus on the structure of apps and classes, allowing more options in the detection rules. This difference is felt on the results of aDoctor, much larger in number of apps and number of instances affected by code smells.

Paprika uses software metrics, such as the number of instructions and cyclomatic complexity, for the detection of code smells. aDoctor, which is described as a lightweight tool, considers none of these aspects. Paprika uses these software metrics mainly for the detection of OO smells, but also for Android-specific code smells, such as HAS, HBR and HSS, which are described later in the paper.

aDoctor uses specific regEx or character strings in the code according to predefined detection rules to determine whether a code smell is present or not. It parses the input Java source code and outputs a result table for each code smell. Paprika, on the other hand, analyses the bytecode of the app and parses it to fill an internal graph database. The nodes of this graph represent methods, classes and attributes. Thanks to the queries executed in this database, the code smells are determined to be detected or not.

3.2.2 Code Smells

We are interested specifically in behavioural code smells. Those behavioural code smells are related to various important concerns of mobile apps, such as energy consumption, memory and performance. Even though other code smells address those same concerns, they are not necessarily behavioural. This is the

Code Smell	Is Behavioural	Studied	Tool
DR			aDoctor
DTWC	✓		aDoctor
DW	✓	✓	aDoctor
HAS	✓	✓	Paprika
HBR	✓	✓	Paprika
HMU	✓	✓	Paprika
HSS	✓	✓	Paprika
IDFP			aDoctor
IDS			aDoctor
IGS			aDoctor & Paprika
IOD	✓	✓	Paprika
ISQLQ			aDoctor
IWR			Paprika
LIC			aDoctor & Paprika
LT	✓		aDoctor
MIM			aDoctor & Paprika
NLMR	✓	✓	aDoctor & Paprika
PD			aDoctor
RAM			aDoctor
SL			aDoctor
UC	✓		aDoctor
UCS			Paprika
UHA			Paprika
UIO	✓		Paprika

Table 1 Summary of the behavioural detected code smells in Paprika and aDoctor.

case, for instance, for the IDS (Inefficient Data Structure) Android code smell [38]. Indeed, the definition of the IDS code smell is the following: "The mapping from an integer to an object through the use of a *HashMap<Integer,Object>* is slow, and should be replaced by other efficient data structures, such as the *SparseArray* [42]". This is not specifically a characteristic in the code *inducing* an inappropriate code behaviour during the execution, but rather a characteristic in the code (the use of a *HashMap<Integer,Object>*) *that negatively impacts* the software quality of the app (lower performance of the app). It is therefore not a behavioural code smell.

In this study, among the 11 behavioural code smells detected in Paprika and aDoctor (see Table 1), we consider the following seven behavioural code smells: DW, HAS, HBR, HMU, HSS, IOD, NLMR. Each is handled by a single tool, with the exception of NLMR that is handled by both Paprika and aDoctor. However, we exclude the four behavioural code smells: DTWC, LT, UC and UIO. LT is deprecated due to the deprecation of the *Thread.stop()* method [37]. The three other code smells, DTWC, UC and UIO, have very few occurrences because they appear in rarer contexts and their few occurrences do not allow for enough relevant cases to be studied. We relate the seven studied behavioural code smells to the three categories of behavioural code smells described earlier. For the first category characterised by the use of a method call or a sequence of method calls, we include DW. For the second

category characterised by runtime information, we include HAS, HBR, HSS, IOD and NLMR. For the third one characterised by undesired data variations during execution, we include HMU.

Still, each code smell may imply thousands of detected code smells and undetected potential code smells within our 318 apps. It hence represents quite some investigation in app source code. We now present these seven selected code smells following the definitions given by [19], [20] and [38]. We provide for each behavioural code smell the associated inappropriate behaviour and characteristics in the source code.

Durable Wakelock (DW): A Wakelock is the mechanism allowing an app to keep the device on in order to complete a task. However, when such task is completed, the lock should be released to reduce battery drain [42]. In Android, the class *PowerManager.WakeLock* is in charge to define the methods to acquire and release the lock. If a method using an instance of the class *WakeLock* acquires the lock without calling the release, a smell is therefore identified.
Inappropriate Behaviour: A call to the *acquire* method is not followed by the call of the *release* method.
Characteristics: The use of *acquire* and *release* methods of the *WakeLock* class.

No Low Memory Resolver (NLMR): When the Android system is running low on memory, the system calls the method *onLowMemory* of every running activity. This method is responsible for trimming the memory usage of the activity. If this method is not implemented by the activity, the Android system automatically kills the process of the activity to free memory. This may lead to an unexpected program termination. Furthermore, when the method *onLowMemory* is declared, it should contain an action reclaiming memory. Therefore, an *Activity* class that does not define *onLowMemory* or that does define this method but in which case *onLowMemory* does not perform any memory reclaiming action, is considered a code smell.

We diverge slightly from the definitions [19, 20, 38] for the NLMR code smell. Indeed, in these papers the code smell is detected by checking that every activity implements the *onLowMemory* method whether or not this method contains any instruction. However, the discussion around the NLMR code smell clearly shows that the intention is to reclaim memory as we do in the above definition.

The NLMR code smell refers to the *onLowMemory* method, which is now deprecated and has been replaced by the *onTrimMemory* method [5]. We nevertheless included it in this study since even if the *onLowMemory* method is deprecated, it is still widely used. Indeed, we will show in Section 3.5.2 that many NLMR instances occur in our dataset. Furthermore, the replacement of a method by another does not change the detection principle and our detection method will still be applicable when the new recommended method is in general use.

Inappropriate Behaviour: The *onLowMemory* method does not reclaim memory when executed.
Characteristics: The implementation of the *onLowMemory* method within an *Activity* class.

HashMap Usage (HMU): The Android framework provides *ArrayMap* and *SimpleArrayMap* as replacements from traditional Java *HashMap*. These structures are considered to be more memory-efficient and to trigger less garbage collection with no significant difference on operations performance for maps containing up to hundreds of values [6, 14]. So, unless a complex map for a large set of objects is required, the use of *ArrayMap* should be preferred over the usage of *HashMap* in Android apps. Therefore, creating small *HashMap* or large *SimpleArrayMap/ArrayMap* instances is considered as a code smell.
Inappropriate Behaviour: A *HashMap* structure is used for a small set of objects or *ArrayMap* / *SimpleArrayMap* structures are used for a large set of objects.
Characteristics: The use of *HashMap* / *ArrayMap* / *SimpleArrayMap* classes.

The three following code smells: Heavy AsyncTask (HAS), Heavy Service Start (HSS) and Heavy BroadcastReceiver (HBR) are very similar. The detection rules for these code smells have the same form: there is a method that occurs on the main process that should be non-blocking and short in execution. Only the name of the method changes.

Heavy AsyncTask (HAS): In Android, the *AsyncTask* API allows developers to perform short background operations. However, three out of the four steps of *AsyncTask* are executed on the main UI thread and not in the background. Thus, these steps should not be time-consuming or blocking operations to avoid: i) the GUI to become unresponsive to user interactions or ii) the ANR dialog to be shown. Thus, a class extending *AsyncTask* should never contain time-consuming or blocking *onPostExecute*, *onPreExecute*, or *onProgressUpdate* methods [33].
Inappropriate Behaviour: The *onPostExecute* / *onPreExecute* / *onProgressUpdate* methods are time-consuming or blocking.
Characteristics: The implementation of *onPostExecute* / *onPreExecute* / *onProgressUpdate* methods within an *AsyncTask* class.

Heavy Service Start (HSS): Services in Android can perform heavy operations in background. However, Android services run in the main thread of their hosting process. By default, the service execution starts with a call to the *OnStartCommand* of the service, which is run in the main UI thread. When the service executes time-consuming or asynchronous operations, a new thread should be created by the method *OnStartCommand* to handle these operations outside the main UI thread, since otherwise it may cause the app to freeze or to display an ANR dialog [32]. Thus, the *OnStartCommand* method should never contain time-consuming or blocking operations.

Inappropriate Behaviour: The *onStartCommand* method is time-consuming or blocking.
Characteristics: The implementation of the *onStartCommand* method within a *Service* class.

Heavy BroadcastReceiver (HBR): Android apps can use a broadcast receiver to manage broadcast communications with the system or other apps. However, the *onReceive* method of *BroadCastReceiver* runs in the main UI thread and may cause the app to freeze or to show an ANR dialog [31]. Thus, the *onReceive* method should never contain time-consuming or blocking operations.
Inappropriate Behaviour: The *onReceive* method is time-consuming or blocking.
Characteristics: The implementation of the *onReceive* method within a *BroadCastReceiver* class.

Init OnDraw (IOD): *OnDraw* routines are responsible for updating the GUI of an Android app. These routines are invoked each time the GUI is refreshed (up to 60 times per second), and thus any extra computational work done in *OnDraw* is magnified by that frequency. Moreover, a high rate of memory allocations may lead into high memory consumption and numerous calls to garbage collection activities [36]. Thus, *OnDraw* routines should never contain *init* instructions to allocate memory (either new or calls to factory/-constructor) or be an excessive time-consuming method.

We diverge slightly from the definitions [19, 20] for the IOD code smell. Indeed, in these papers the code smell is detected by checking that every *onDraw* method does not have memory allocations. However, the discussion around the IOD code smell clearly shows that the intention is not to have any extra computational work done in *onDraw* so the method must not be time-consuming as we do in the above definition.
Inappropriate Behaviour: The *onDraw* method is time-consuming or initialises objects.
Characteristics: The implementation of the *onDraw* method within a *View* class.

3.3 Objects

The apps used in the study come from the dataset published by Habchi *et al.* [18], where it was used to analyse and detect code smells based on GitHub project commits. This dataset consists of real open-source apps from F-Droid published on GitHub. F-Droid provides a dataset of real apps that are neither dummy apps, nor templates, nor libraries. We study 318 of the 324 apps originally published in the dataset of Habchi *et al.* For the remaining six cases, four apps are no longer available, their repositories became private or deleted, and two of them cannot be processed by Paprika. The reasons are not known or

detailed, it may be because their repositories have evolved using files/classes not supported by PAPRIKA.

We present the dataset of the apps from different viewpoints in terms of category, size, interest and contributions to illustrate its diversity and representativeness. The list of apps and the metrics associated with the apps used for the study are published [1] for the sake of evaluation and replication.

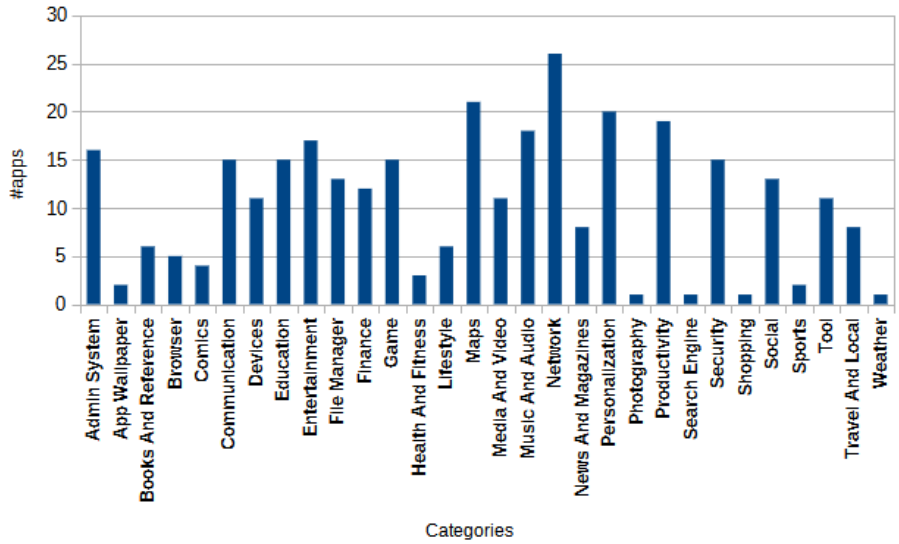

Fig. 1 Distribution of the apps with regards to their category.

1) *Category:* We classify the apps of the dataset according to the category they belong to. We determine categories by analysing manually the related information in the GitHub repository of each app. We found 30 categories in the dataset. Figure 1 shows the number of apps per category. For example, *opensudoku* belongs to the category *Game* and *Wifi-Fixer* belongs to the category *Network*. Two apps have no categories because their associated GitHub repository has disappeared in the meantime. The apps of the dataset are well represented uniformly within these 30 different categories.

2) Size: Finally, we describe our dataset in terms of app size. We focus in particular on the metrics of the number of classes and the number of lines of code. The distribution of the number of classes is presented in Figure 2 and the distribution of the number of lines is presented in Figure 3. The number of classes varies from 1 to 1,326 and the number of lines varies from 108 to 166,611. Apps of all sizes are present in the dataset, both in number of classes and number of lines. The majority of apps have between 15 and 138 classes

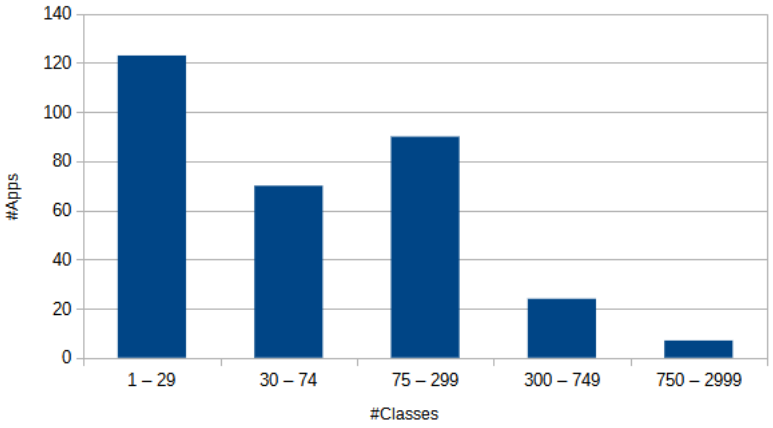

Fig. 2 Distribution of the apps with regards to their number of classes.

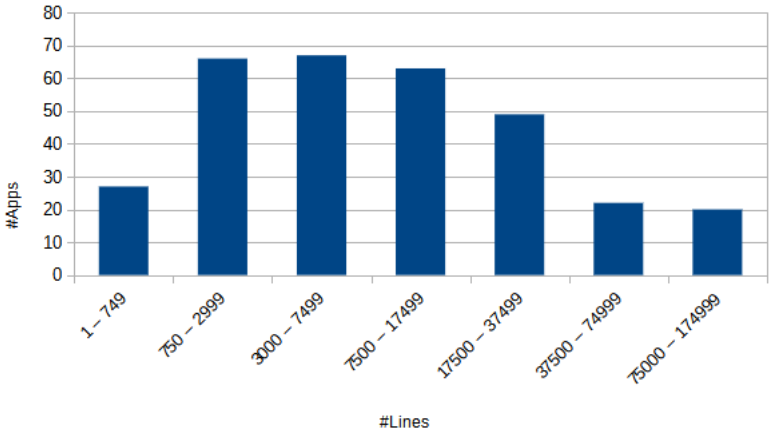

Fig. 3 Distribution of the apps with regards to their number of lines.

and between 2,032 and 21,535 lines of code.

3) Interest: We describe our dataset from the point of view of the GitHub community. The stars and watchers allow us to measure the interest of the community towards the different repositories. Giving a star to a project allows you to show interest in a project so that you can easily retrieve it. Becoming a watcher of a repository allows you to show your interest and to be kept up to date on the activities of the repository. The distribution of stars and watchers is presented in Figure 4. We note that the dataset represents all kinds of interest, with some repositories having very high interest with star counts reaching thousands of stars, up to 14,767. The majority of apps have between 29 and

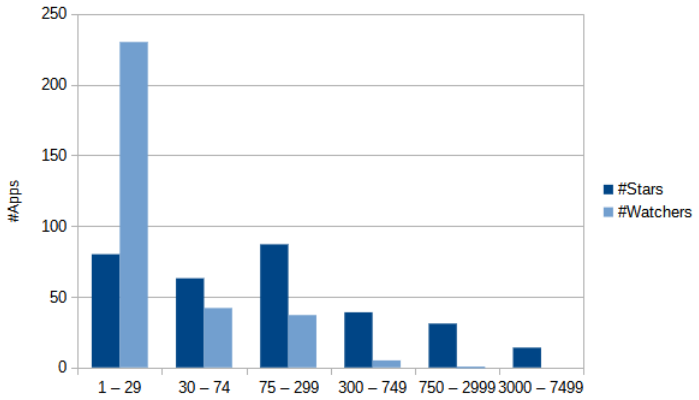

Fig. 4 Distribution of the apps with regards to their stars and watchers.

335 stars and between 6 and 31 watchers.

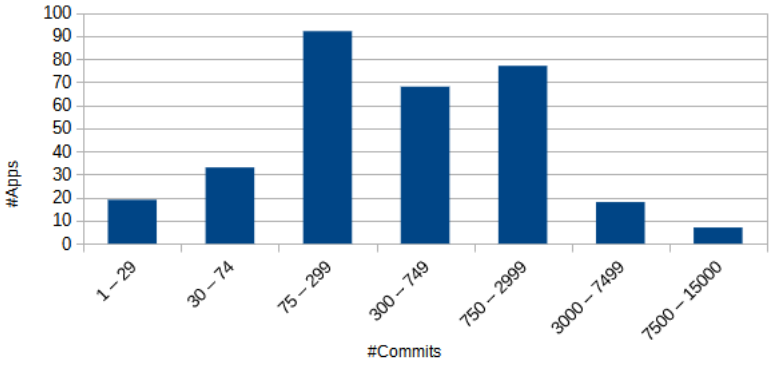

Fig. 5 Distribution of the apps with regards to their commits.

4) *Contribution*: We also describe the dataset according to the contribution of users in the different repositories. For this purpose, we use the number of contributors to the repositories as well as the number of commits. The distribution of commits is presented in Figure 5 and the distribution of contributors is presented in Figure 6. The majority of apps have between 127 and 1043 commits and between 3 and 21 contributors.

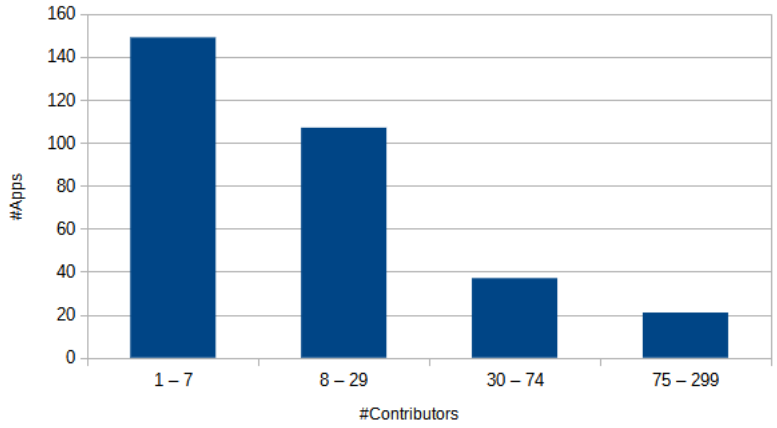

Fig. 6 Distribution of the apps with regards to their contributors.

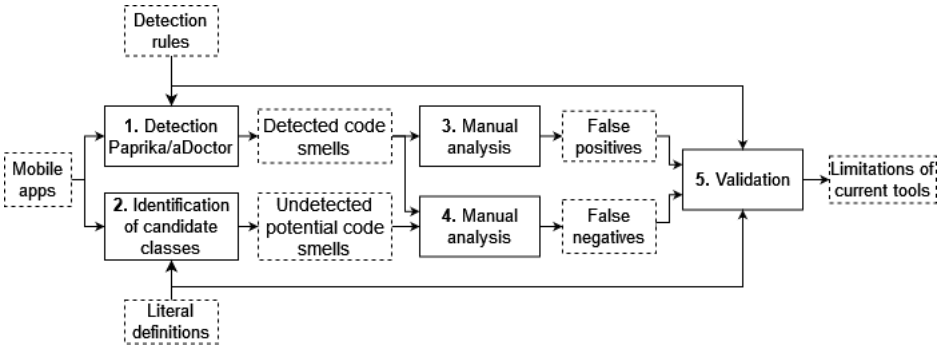

Fig. 7 Overview of the process.

3.4 Process

The process of the study as illustrated in Figure 7 consists of five main steps described in the following.

Step 1. We use Paprika and aDoctor on the dataset of 318 apps to detect code smells, hence retrieving the detected code smells (i.e., positive instances).

Step 2. In parallel, in Step 2 we identify the candidate classes for the undetected potential code smells across the 318 apps. These candidate classes are all the classes possibly affected by a code smell conforming to its definition. The undetected potential code smells (i.e., negative instances) are the candidate classes not detected as code smell by either Paprika or aDoctor. We chose the candidate classes on the basis whether they contain a method or an instruction mentioned in the code smell's definition. Clearly, any other class cannot be an instance of the code smell. For example, the IOD code smell's definition refers to the inherited *onDraw* method containing memory alloca-

tions or having a too long execution time. The candidate instances are hence all classes implementing the *onDraw* method. We then obtain the undetected potential code smells of these candidate classes by removing all classes already detected by Paprika or aDoctor.

Furthermore, it is important to note that in Step 3 and Step 4, depending on the code smell, there may be a huge number of instances to analyse manually. For instance, as we will see further in Table 2 of Section 3.5, in our dataset some code smells may contain few instances such as 19 instances for IOD, while others have more than 1000 instances, such as NLMR.

The manual inspection of each instance is a complex task, especially given the number of instances. Therefore, to cope with the manual analysis of Steps 3 and 4, while still considering a statistically significant sample for each code smell, we rely on a stratified sample. This stratified sample ensures that the proportion of each code smell is preserved in the sample.

More precisely, we randomly selected a set of 312 positive and 244 negative code smell instances. The sample for each code smell consists of the detected code smells fetched by Step 1 and the undetected potential code smells fetched by Step 2. This represents a 95% statistically significant stratified sample with a 10% confidence interval of the instances obtained by Step 1 and Step 2. The confidence interval is set to 10% to use the same interval as the reference dataset of Habchi et al. [18].

Step 3**.** We analyse manually which detected code smells are false positives.

Step 4**.** We analyse manually which undetected potential code smells are false negatives.

The manual analysis was performed as follows. Each class contained in the sample of the code smells was studied manually by three people: one of the authors and two other PhD students with expertise in mobile apps and their code smells. For each class to be verified, each participant examined the class thoroughly to determine whether the code smell really occurs. We only share the results at the end to avoid influencing each other. In the few cases where there were discrepancies, we revisited the case to reach a consensus of the majority. This process took two weeks. All the results of the consensus can be found in the artefacts [1]. The three participants received specific criteria for each code smell to determine whether an instance is a true positive, false positive, true negative or false negative. For example, for the HMU code smell which refers to data structure size, we consider how these structures are filled. If they are filled a finite number of times or within small loops, we consider them small. If they are filled by loops iterating over a possibly very large variable, we assume they are possibly large. For code smells like HSS, where the execution time is crucial, participants searched for big loops, many calls to complex methods or many objects used or allocated. The rules for determining whether a code smell really occurred are defined in the qualitative analysis in Section 3.5.2, in the Manual Analysis parts.

Step 5**.** Finally, we identify the inconsistencies between the detection rules of Paprika and aDoctor with the literal definitions while relying on the false positives/negatives and we deduce the limitations of these tools. This

validation has been performed by a developer, who has a deep understanding of Android code smells. This step of validation is not subject to interpretation and does not require the involvement of several developers because it is a straightforward process as indicated in the following. The step of validation includes the following three sub-steps. The first sub-step consists in comparing the detection rules of the tools with the literal definitions and deducing the inconsistencies between them. The second sub-step consists in confirming these inconsistencies with the false positives/negatives that illustrate concrete instances of detected code smells and undetected potential code smells. In the last third sub-step, based on the inconsistencies confirmed, the limitations of the tools are identified. For example, the literal definition of the HMU code smell indicates that an HMU code smell manifests when *HashMaps* are used for small structures or *ArrayMaps* are used for large structures. However, the detection rule for the HMU code smell indicates only that there is a code smell when a *HashMap* structure is used. In terms of inconsistencies, the detection rule does not take into consideration the sizes of structures nor does refer specifically to the *ArrayMap* structure. These inconsistencies are being confirmed among the false positives/negatives: the false positives include large *HashMaps* and the false negatives include large *ArrayMaps*. Based on these inconsistencies, the limitation of current tools is that they cannot adequately detect code smells characterised by undesired data variations during execution.

3.5 Results

In this section, we respond to the two research questions using the results of the empirical study. We answer the first research question using a quantitative analysis. On the other hand, we answer the second research question with a qualitative analysis.

3.5.1 $\boldsymbol{RQ_1}$*: Are the tools of concern effective for detecting behavioural code smells?*

To answer this research question, we will focus on the quantitative results of our study.

	Verified instances (V)	
	Positive	**Negative**
Detected code smells (D) (positive instances)	True Positive (TP) $D \cap V$	False Positive (FP) $D \setminus V$
Undetected potential code smells (D^c) (negative instances)	False Negative (FN) $D^c \setminus V^c$	True Negative (TN) $D^c \cap V^c$

Fig. 8 True/False Positive/Negative instances.

Code Smell	#Apps	#Instances	Tool
DW	81	199	aDoctor
NLMR	292	2826	aDoctor
NLMR	259	1132	Paprika
HMU	136	729	Paprika
HAS	53	144	Paprika
HSS	38	48	Paprika
HBR	132	513	Paprika
IOD	16	19	Paprika

Table 2 Results reported by the tools Paprika and aDoctor.

First, let clarify the different types of instances handled in the study. The process of Figure 7 distinguishes two types of code smell instances. As shown in Figure 8, it first considers the *Detected code smells* (D), which are the instances returned by the tools in Step 1 of Figure 7. Conversely, the complement of this set, the *Undetected potential code smells* D^c are the instances not detected by the tools and hence returned in Step 2 of Figure 7. Secondly, the *Verified instances* (V) are those determined by the manual analysis of Step 3 and Step 4 in Figure 7 and conform with the definition of the code smells. In this case, the complement V^c is simply those instances not in conformance with the code smells definition.

We can now distinguish the following four types of instances, as shown in Figure 8. The true positives are the instances detected by the tools and validated manually in conformance with their code smells definition: $TP = D \cap V$. The false positives are the instances detected by the tools but validated manually not in conformance with the code smells definition: $FP = D \setminus V$. The false negatives are the instances not detected by the tools but validated manually to be in conformance with the code smells definition: $FN = D^c \setminus V^c$. Finally, the true negatives are the instances not detected by the tools and validated manually not in conformance to the definition: $TN = D^c \cap V^c$.

We will now present our results. Table 2 shows the number of apps and the number of detected code smells returned by Paprika and aDoctor in Step 1 of Figure 7. It should be noted that a single code smell is common to both tools, the NLMR code smell. It is also worth mentioning that even if, in this case, both tools return *Activity* classes not defining the *onLowMemory* method, the number of classes and even more the number of instances differ. This is due to the fact that Paprika checks if one of the superclasses of the class candidate is the *Activity* class, while aDoctor checks if the class extends one of the 95 Android classes from a predefined list.

Table 3 presents the number of undetected potential code smells as returned by Step 2 of Figure 7. As described in the process, the undetected potential code smells are the candidate classes affected by a code smell but not detected by the tools. Here the number of undetected potential code smells for NLMR is the same for both tools. Indeed, the identification of candidates classes in Step 2 of Figure 7 is independent of the tools and filtering out the

Code Smell	#Apps	#Instances	#Tool
DW	46	64	aDoctor
NLMR	18	33	aDoctor
NLMR	18	33	Paprika
HMU	13	24	Paprika
HAS	159	846	Paprika
HSS	72	113	Paprika
HBR	112	376	Paprika
IOD	102	274	Paprika

Table 3 Number of undetected potential code smells identified.

Code Smell	#detected code smells	#undetected potential code smells	#false positives	#false negatives	Precision	Recall	F_1-Measure	Tool
DW	66	64	0 (0%)	25 (39%)	100%	73%	84%	aDoctor
NLMR	93	33	0 (0%)	6 (18%)	100%	94%	97%	aDoctor
NLMR	89	33	0 (0%)	6 (18%)	100%	94%	97%	Paprika
HMU	85	24	42 (49%)	10 (42%)	50%	81%	62%	Paprika
HSS	48	52	35 (73%)	10 (19%)	27%	57%	37%	Paprika
IOD	19	71	0 (0%)	12 (17%)	100%	61%	76%	Paprika
Total	400	277	88 (28%)	62 (25%)	N/A	N/A	N/A	N/A
Average	N/A	N/A	N/A	N/A	64%	69%	64%	N/A

Table 4 Results of the sample of code smells studied.

instances returned by these tools gives, in this case, the same result. Even if the definitions in the literature are similar, the detection rules may vary from one tool to another. Both tools detect an NLMR code smell if an *Activity* does not implement the *onLowMemory* method. While Paprika only focuses on classes inheriting from Activity, aDoctor is also interested in other classes inheriting from *ComponentCallBacks*, such as *Fragments*. This is reasonable since the *onLowMemory* method is defined in *ComponentCallBacks*. We followed the definition and were only interested in *Activity* classes for candidate classes, which both tools detect at similar level. As a result, the detected code smells differ from the tools while the undetected potential code smells are the same.

Table 4 presents the number of false positives and false negatives as determined in Steps 3 and 4 of Figure 7. There is quite some variation from one code smell to another, but each code smell has its batch of false positives and false negatives. For the false positives, we observe that we can go from 0% for code smells whose detection ensures that the code smell indeed occurs, to

72.92% for results for which manual verification is necessary and essential. On the other hand, for false negatives, we observe that we can go from 17% for the behavioural code smells for which the tools cover almost every instance, to 42% for the code smells where tools are missing a lot of instances. Since HBR, HAS and HSS are very similar and differ only in the name of the method, and because there is a huge number of instances for each of them (see Table 2), the study focuses only on HSS. The means of detection for these three code smells are exactly the same. In some cases, the instance numbers shown in Table 4 coincide with those of Table 2 or Table 3. This is a simple consequence of stratification. Indeed, code smells with few instances will often have the same or nearly the same number of instances after stratification.

The false positives and false negatives come from the detection rules used in Paprika and aDoctor. One can therefore conclude that those tools capture far more code smells than necessary. This is a consequence of a static analysis based on search patterns and metrics computation that does not allow a perfect accuracy leading to false positives. For similar reasons, the detection techniques do not capture all the code smells that should be captured, which leads to false negatives. This is mainly due to technical limitations brought by a purely static detection on behavioural code smells, as we discuss later in this section. Code smells with many instances may lead to a surplus of work to check whether they should be considered. Indeed, Paprika [19] indicates that for some studied code smells, like NLMR and HMU, the responsibility is left to the developer to check if an instance identified by Paprika deserves a correction, or not. False positives and false negatives are further analysed and discussed in Section 3.5.2. The rules causing false negatives and false positives are defined in the qualitative analysis in Section 3.5.2.

One can also consider precision, recall, and F_1-Measure. Precision is the proportion of true positives among all positives, recall is the proportion of true positives among true positives and false negatives, and the F_1-Measure is the harmonic mean of precision and recall. Precision is mostly high, but in some cases, like HMU and HSS, half or more of the detected instances are not relevant. In such cases, it is hence left to the developers to determine whether the detected code smells are relevant. Recall tends to be lower than precision, hence many relevant instances are missing from the detected instances. It is hence left to the developers to find these missing code smell instances. Based on the results of Table 4, we hence reject H_1^{posi} and H_2^{nega}.

Answer RQ_1: The tools return false positives up to approximately 73% for HSS and false negatives up to approximately 42% for HMU. They both impact the effectiveness. The false positives detected by the tools of concern affect their effectiveness because they return to the developers erroneous detection results. The false negatives affect the effectiveness by returning to the developers partial detection results. The precision may be low for specific code smells like HMU with 27% or HSS with 50%. The recall drops down to 57%. Overall, the precision for the code smells studied is 74%, the recall is 69% and the F_1-Measure 64%, which is quite a bit low.

3.5.2 RQ_2: Are the code smells detected by the tools of concern consistent with their original literal definition?

While the previous section focused primarily on the quantitative results, this section discusses the nature of the results depicted in Table 4 with a qualitative analysis. We hence study in detail the selected code smells to check to which extent the results of the detection go against their literal definition.

For each code smell, we follow a common description pattern, which refers to the terms in Figure 7: a) an excerpt of the literal definition of the code smell from Section 3.2 as a reminder. b) the concrete detection rules implemented in the aDoctor and Paprika tools to detect the code smell regardless of what has been described in the related papers; c) the nature of the detected code smells, i.e., how the classes that contain such code smell are characterised; d) the nature of the undetected potential code smells; e) the criteria used during the manual analysis for the detected code smells; f) the criteria used during the manual analysis for the undetected potential code smells; g) the nature of the false positives with an illustrative example, if applicable; h) the nature of the false negatives with an illustrative example, if applicable; i) a discussion on the results; j) the technical limitations identified of the aDoctor and Paprika tools with the problems associated with the current detection rules. Each description pattern associated to a code smell allows us to respond to the research question RQ_2, i.e., whether the code smell detected by aDoctor and Paprika is consistent with its original literal definition.

Durable Wakelock (DW)
a) Literal definition: A DW code smell manifests when there is an instance of the *WakeLock* class that acquires the lock (using the *acquire* method) without releasing it (using the *release* method).
b) Detection rules: (aDoctor) A DW code smell manifests if there is a call of a method with a name that contains the string of characters *"acquire"*. This detection rule does not check specifically the *acquire* method but a string of characters *"acquire"*. Also, it does not check if there is an instance of the *WakeLock* class and if the *release* method is absent. However, the detection

```
private void acquireProximityWakeLock() {
    synchronized (AudioPlayer.LOCK) {
        if (wakeLock != null) {
            wakeLock.acquire();
        }
    }
}

private void releaseProximityWakeLock() {
    synchronized (AudioPlayer.LOCK) {
        if (wakeLock != null && wakeLock.isHeld()) {
            wakeLock.release();
        }
    }
    messageAdapter.setVolumeControl(AudioManager.STREAM_MUSIC);
}
```

Fig. 9 Example of an DW false negative within the *Conversations* app and *AudioPlayer* class. As the calls are in two dissociated methods, we can not be sure they are called in the proper order.

rule presented in the paper of aDoctor [38] indicates to check also the absence of *release* and that both methods are from the *WakeLock* class. We hence manually checked within the detected code smells the *WakeLock* instance and the absence of *release*. We only report the results on this enhanced version of the rule.

c) Detected code smells: With this enhanced rule, the DW detected code smells in aDoctor are classes that call the *acquire* method of the *WakeLock* class , but do not implement the *release* method.

d) Undetected potential code smells are the classes that contain both methods from the *WakeLock* class.

e) Manual analysis of the detected code smells: No manual analysis is required because all detected code smells are true positives. Indeed, the related classes do not have the method *release* implemented.

f) Manual analysis of the undetected potential code smells: If both methods are present, several criteria are applied. If the two method calls are separated by a whole bunch of function calls and conditions tests that we are not sure are executed/satisfied, then we cannot be sure that both methods are properly called.

g) False positives: Each detected code smell is irrevocably a true positive because the detected code smells do not have the method *release* implemented.

h) False negatives: The true negatives ones are the undetected potential code smells where both methods will be called, while the false negatives are the undetected potential code smells where this is not the case. Almost 39% of the undetected potential code smells are false negatives, meaning we cannot be confident enough that the two methods are called correctly to minimise the battery drain. This may be due to the fact that the two methods are far apart from many methods and conditions, which may not always be re-

spected as depicted in Figure 9. As the calls are in two dissociated methods, here *acquireProximityWakeLock()* and *releaseProximityWakeLock()*, it is also necessary to check if they are called correctly. However, we have no clues as whether these functions are properly called in the correct order. It can also be the situation that the *release* is only done when the app is destroyed, which does not conform to the definition of the code smell. This last scenario is the most frequent among the false negatives encountered.
i) Discussion on the results: This code smell is one of the most difficult to determine its effectiveness manually. Determining whether two distant methods will be called on apps whose structure is not fully known is difficult.
j) Limitations of current tools: The current aDoctor implementation of the DW detection simply identifies the presence of the *acquire* method without checking the presence of the *release* method. And only because both methods are present does not mean that they are necessarily called.

```
public void onLowMemory() {}
```

Fig. 10 Example of the *onLowMemory* method within the *osmeditor4android* app and *MapViewLayer* class. Although the method is defined, it is empty and therefore has no reaction on the memory.

```
public void onLowMemory() {
  Logger.info("low memory for application");
  super.onLowMemory();
 }
```

Fig. 11 Example of the *onLowMemory* method within the *trackworktime* app and *WorkTimeTrackerApplication* class. Although the method is defined, it is only composed of a log and therefore has no reaction on the memory.

No Low Memory Resolver (NLMR)

a) Literal definition: An Activity class that does not define *onLowMemory* or that does define this method but in which case *onLowMemory* does not perform any memory reclaiming action, is considered a code smell.
b) Detection rules: (Paprika & aDoctor) An *Activity* class does not have an *onLowMemory* method. The detection rules are the same for aDoctor and Paprika, but their implementations are slightly different. Paprika checks if one of the superclasses of the class candidate is the *Activity* class, while aDoctor checks if the class extends one of the 95 Android classes from a predefined list. This implementation difference has an impact on the number of detected code smells as shown in Table 2.
c) Detected code smells are the *Activity* classes that do not define the *onLowMemory* method.

d) Undetected potential code smells are the *Activity* classes that define the *onLowMemory* method and do not perform any memory reclaiming action.
e) Manual analysis of the detected code smells: No manual analysis is required because all detected code smells are true positives.
f) Manual analysis of the undetected potential code smells: We check if the *onLowMemory* methods do not perform any memory reclaiming action.
g) False positives: All the detected code smells do not have the *onLowMemory* method implemented, so they are all true positives. There are therefore no false positives.
h) False negatives: When inspecting the *onLowMemory* methods of undetected potential code smells, some methods do not perform any memory reclaiming action. In particular, this occurs mainly for the following reasons: (1) The method body is empty as in Figure 10; (2) It only serves to log purpose as shown in Figure 11, (3) or it has no effect like calling the *super.onLowMemory*, which is also empty.
i) Discussion on the results: This code smell is not frequent. Only 18% of the investigated methods (i.e., 6 instances) have no action on the memory. Indeed, when the *onLowMemory* method is declared, there is usually a memory reclaiming action.
j) Limitations of current tools: The tools only consider the absence of the *onLowMemory* method but they do not consider its presence with no memory reclaiming action.

```
private Pair<List<String>, List<String>> _changes() {
    Map<String, Object[]> cache = new HashMap<>();
    try (Cursor cur = mDb.getDatabase().query("select fname,
        csum, mtime from media where csum is not null", null)) {
        while (cur.moveToNext()) {
            String name = cur.getString(0);
            String csum = cur.getString(1);
            Long mod = cur.getLong(2);
            cache.put(name, new Object[] { csum, mod, false });
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    (...)
}
```

Fig. 12 Example of an HMU false positive within the *Anki-Android* app and *Media* class. Here, the *HashMap cache* structure can be huge because it is filled from a database which is in accordance with the definition.

HashMap Usage (HMU)

a) Literal definition: An HMU code smell manifests when a *HashMap* structure is used for small set of objects and *SimpleArrayMap/ArrayMap* for large set of objects.

```
private void getBucketData(final ArrayMap<String, String>
    bucketNamesList, final String buckedId, final String
        buckedName) {
 final String[] projection = { buckedId, buckedName, "count(*) as
     media_count" };
 final String groupBy = "1) GROUP BY (2";
 final Cursor cursor =
     PBApplication.getApp().getContentResolver().query(imagesUri,
     projection, groupBy, null, "media_count desc");

 if (cursor != null && cursor.moveToFirst()) {
  String name;
  String id;
  String count;
  final int bucketIdColumn = cursor.getColumnIndex(buckedId);
  final int bucketNameColumn = cursor.getColumnIndex(buckedName);
  final int bucketCountColumn =
      cursor.getColumnIndex("media_count");
  do {
   id = cursor.getString(bucketIdColumn);
   name = cursor.getString(bucketNameColumn);
   count = cursor.getString(bucketCountColumn);
   bucketNamesList.put(id, name + " (" + count + ")");
  } while (cursor.moveToNext());
 }
 (...)
}
```

Fig. 13 Example of an HMU false negative within the *client-android* app and *PBMediaStore* class. Here, the *ArrayMap bucketNamesList* structure can be huge because it is filled from a database while it should remain a small structure.

b) Detection rules: (Paprika) A class uses an *HashMap* structure.
c) Detected code smells are the classes that use an *HashMap* structure.
d) Undetected potential code smells are the classes that use a *SimpleArrayMap* or *ArrayMap* structure.
e) Manual analysis of the detected code smells: We consider how these structures are filled. If they are filled a finite number of times or within small loops, we consider them small. If they are filled by loops iterating over a possibly very large variable, we assume they are possibly large.
f) Manual analysis of the undetected potential code smells: The same criteria are used, as it is also necessary to examine the size of the structures.
g) False positives: Conforming to the definition of the code smells, a true positive is an instance that uses an *HashMap* for a small structure, and a false positive is an instance that uses an *HashMap* for a large structure. An example of the HMU code smell raised by Paprika is shown in Figure 12. Here Paprika raises a code smell because it detects a *HashMap* and leaves it up to the user to check whether the detection is effective. However, in this example, the *cache* has no limitations on the *.put()* associated with it. One would expect this *HashMap* to be large because the request may return a lot of results. Therefore, it may not be considered as a smell.

h) False negatives: Conversely, a true negative is an instance that uses a *SimpleArrayMap/ArrayMap* for a short structure, and a false negative is an instance that uses a *SimpleArrayMap/ArrayMap* for a large structure. The false negatives should also include *HashMap* used for a small number of map entries. However, there are no such cases in our sample. PAPRIKA reports all *HashMaps* as code smells, therefore all instances containing *HashMaps* are detected code smells. An example of an undetected potential code smell is shown in Figure 13. Here PAPRIKA does not detect a code smell because it only focuses on *HashMap*. However, in this example, the *bucketNamesList ArrayMap* has no limitations on the *.put()* associated with it. One would expect this *ArrayMap* to be large because it is filled using a query from a media database. Therefore, it may be considered as a smell.
i) Discussion on the results: The findings are quite similar, whether in the case of detected code smells with 49% false positives or undetected potential code smells with 42% false negatives, meaning in both cases the detection seems to be insufficient to correctly identify the code smell. It is important to note that developers do not seem to specifically use one structure or another depending on the size envisaged, which suggests that a detection by looking at one of the structures is not efficient enough. However, *HashMap* is the most used structure.
j) Limitations of current tools: There is no notion of structure sizes, the tools only detect the presence of an *HashMap*.

Heavy Service Start (HSS)
a) Literal definition: An HSS code smell manifests when the *OnStartCommand* method contains time-consuming or blocking operations.
b) Detection rules: (PAPRIKA) The method *onStartCommand* has more than 17 instructions or a cyclomatic complexity greater than 3.5 (when the threshold is set on low).
c) Detected code smells are the classes defining the method *onStartCommand* having a cyclomatic complexity greater than 3.5 or a number of instructions greater than 17.
d) Undetected potential code smells are the classes defining the method *onStartCommand* having a cyclomatic complexity lower than 3.5 and a number of instructions lower than 17.
e) Manual analysis of the detected code smells: We determine if a method is not time-consuming by investigating the source code according to several indicators. Here is a non-exhaustive list of indicators: 1) The method does not have loops or small ones; 2) It only has trivial method calls whose content is known and recognised; 3) There are no operations that are potentially costly, such as queries on large items in a database; 4) There are few memory allocations, we can assume it is not time-consuming. If we cannot determine the time-consuming aspect because of unknown snippets of code, we consider them potentially time-consuming for the detected code smells and potentially not time-consuming for the undetected potential code smells. Indeed detected code smells have already been identified with long and complex methods while

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
 onHandleIntent(intent);
 return START_NOT_STICKY;
}
@Override
protected void onHandleIntent(Intent intent) {
 (...)
 Cursor cursor = getContentResolver().query(content, projection,
     selection, null, null);
 (...)
 newSongsMetadata = new ArrayList<>();
 while (cursor.moveToNext()) {
  String artist = cursor.getString(0);
  String title = cursor.getString(1);
  (...)
  newSongsMetadata.add(new String[]{artist, title});
 }
 (...)
 for (String[] track : newSongsMetadata) {
  String artist = track[0];
  String title = track[1];
  threadPool.execute(() -> {
   try {
    Request request;
    File musicFile =
        Id3Reader.getFile(BatchDownloaderService.this, artist,
        title, false);
    (...)
    request = LyricsChart.getVolleyRequest(lrc,
        BatchDownloaderService.this, BatchDownloaderService.this,
        fingerprint, artist, title);
    requestQueue.add(request);
   } (...)
  });
 }
}
```

Fig. 14 Example of an HSS false negative within the *QuickLyrics* app and *BatchDownloaderService* class. Here, the method includes a long sub-method composed of complex operations, queries, iterating loops. Such method is expected to have a long execution time.

undetected potential code smells have short and not complex methods.

f) Manual analysis of the undetected potential code smells: The same criteria are used, as we also need to investigate whether the method is time-consuming.

g) False positives: A detected code smell is a false positive if the method is not time-consuming. Very often, these functions consist of a large switch or a sequence of if-else, resulting in a large function but without a lot of executing operations. Figure 15 is an excerpt of a detected code smell. Although the *onStartCommand* method is long with 57 lines, it consists of a big switch case. Each branch of the switch case is composed of minor operations. Such an instance may not be considered as a code smell, as one can expect a short

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
 int action = intent != null ? intent.getIntExtra("action", 0) :
     0;
 int id = intent != null ? intent.getIntExtra("task_id", 0) : 0;
 switch (action) {
  case ACTION_ADD:
   (...)
   break;
  case ACTION_CANCEL:
   (...)
   break;
  case ACTION_PAUSE:
   if (id == 0) break;
   saveTask = mTasks.get(id);
   if (saveTask != null) {
    saveTask.pause();
   }
   break;
  case ACTION_RESUME:
   (...)
   break;
  case ACTION_CANCEL_ALL:
   (...)
   break;
 }
 return START_REDELIVER_INTENT;
}
```

Fig. 15 Example of an HSS false positive within the *OpenManga* app and *SaveService* class. Although this is a long method, it is subdivided into small blocks of simple instructions. Such method is expected to have a short execution time.

execution time.

h) False negatives: The false negatives are the time-consuming undetected potential code smells. There are some cases where despite having a short method, it may still be time-consuming or blocking operations. These false negatives are usually composed of costly loops, interweaving of many sub-methods. Figure 14 is an excerpt of an undetected potential code smell. Although the *onStartCommand* method is very brief, it consists of a long *onHandleIntent* method of 70 lines. The latter is composed of many complex operations, such as queries on a large data set and loops performing operations on this data. Such an instance can be considered as a code smell, as one can expect a long execution time.

i) Discussion on the results: Despite similar criteria, there are many more false positives (73%) than false negatives (19%). The difference between false positives and false negatives shows that the metrics are effective enough to detect short operations, but ineffective in determining long operations.

j) Limitations of current tools: The detection is based on arbitrary heuristics. Rather than really checking whether the concerned methods involve long or blocking operations, heuristics are used to report as smells the methods

with the number of instructions and cyclomatic complexity metrics greater than an arbitrary value.

```
@Override
protected void onDraw(Canvas c, IMapView osmv) {
    boolean squareTiles = myRendererInfo.getTileWidth() ==
        myRendererInfo.getTileHeight();

    for (int y = tileNeededTop; y <= tileNeededBottom; y++) {
        for (int x = tileNeededLeft; x <= tileNeededRight; x++) {
            (...)
            Bitmap tileBitmap = null;
            if (tile.zoomLevel >= minZoom && tile.zoomLevel <=
                maxZoom) {
                tileBitmap = mTileProvider.getMapTile(tile,
                    owner);
            }
            if (tileBitmap == null) {
                while ((tileBitmap == null) && (zoomLevel -
                    tile.zoomLevel) <= maxOverZoom &&
                    tile.zoomLevel > minZoom) {
                    tile.reinit();
                    (...)
                    tileBitmap = mTileProvider.getMapTile(tile,
                        owner);
                }
            }

            if (tileBitmap != null) {
                c.drawBitmap(tileBitmap, new Rect(tx, ty, tx +
                    sw, ty + sh),
                        new Rect(destRect.left + xPos,
                            destRect.top + yPos, destRect.right +
                            xPos, destRect.bottom + yPos),
                            mPaint);
            } (...)
    }
}
```

Fig. 16 Example of an IOD false negative within the *osmeditor4android* app and *MapTilesLayer* class. The method consists of a lot of nested loops, calculations and zoom operations. Such method is expected to be costly.

Init OnDraw (IOD)

a) Literal definition: An IOD code smell manifests when the *onDraw* method contains *init* instructions to allocate memory or contains time-consuming operations.

b) Detection rules: (PAPRIKA) An *onDraw* method calls constructors.

c) Detected code smells: The detected code smells in PAPRIKA are the classes using the *onDraw* method containing constructor calls.

d) Undetected potential code smells: The undetected potential code

smells are the classes using the *onDraw* method not detected by PAPRIKA, and thus excluding the ones that contain directly constructor calls.
e) Manual analysis of the detected code smells: As there are no false positives as explained in the following (see g)), there is no manual analysis of the detected code smells.
f) Manual analysis of the undetected potential code smells: The criteria are the same as the HSS code smell, as the criteria concern memory and time-consuming operations.
g) False positives: A detected code smell is a false positive if despite being flagged as a code smell, the method is not time-consuming and does not have memory allocations. As each detected code smell has memory allocations, there are no false positives.
h) False negatives: A false negative is an undetected potential code smell where the method *onDraw* has time-consuming operations and improper memory allocation. Figure 16 is an excerpt of an undetected potential code smell. There are no direct initialisations, but it is still a long method of 150 lines that performs a lot of computations, which could have been done outside this method. For example, a whole set of zoom-related computations that could be done elsewhere along with nested loops. Such a method can have strong consequences on the performance of the app as indicated by the definition of IOD. It could therefore be considered as a code smell.
i) Discussion on the results: Only a small proportion of the undetected potential code smells appears to be time-consuming with 17% of false negatives. Like HSS, the detection seems quite effective not to overlook the instances that need to be detected. It means that usually the methods without initialisations usually have no time-consuming operations, but there are still instances with a lot of time-consuming loops of computing at each iteration of *onDraw*.
j) Limitations of current tools: Looking at the presence of initialisation of new objects is not enough to handle the execution time.

Summary of the qualitative results. The code smells are inconsistent with their literal definition if elements of the definition are missing in the tool's detection rules. We listed inconsistencies in the "Limitations of current tools" part for each code smell. For example, the literal definition of HMU states that *HashMap* should be used only for big structures, while *SimpleArrayMap* is preferred for small ones. But, in this case, the instances detected by the tool contain the *HashMap* structure, when large *HashMap* are consistent with the literal definition. There is also no consideration for *SimpleArrayMap* in the tool's detection rule despite its importance in the literal definition. One more example: for DW, when a *WakeLock* is acquired, a release must be called. However, the tool considers only the definitions of the acquire and release methods. However, this does not necessarily mean that these methods are called, as required by the literal definition.

Based on the limitations of current tools for each code smell, we hence reject H_3^{rule}.

Answer RQ_2: By analysing the detection rules, problems occur for each code smell studied. These problems have an impact on the detection results, which return false positives and false negatives. Observing the results shows that the detected code smells are not always consistent with their literal definition. For each code smell, limitations of current tools due to static analysis have been identified.

3.6 Threats to Validity

Internal Validity. The main threat to our internal validity could be an imprecise detection of behavioural code smells. The results in Table 4 are also subject to imprecision, even though they are bound to be very close to reality, being checked manually by two other developers. The disparity in the propagation of behavioural code smells may affect the results because some code smells are widely spread, like the NLMR or HMU, while some are not widespread, like IOD. Indeed, we may not find the best studying cases in our sample of results that are not widespread. Moreover, the fact that one is forced to reduce the number of samples to be analysed may also have an impact for the same reasons.

External Validity. The main threat to external validity is that our study only concerns seven Android-specific behavioural code smells. Without a further investigation, these results should not be generalised to other code smells or development frameworks. We therefore encourage future studies to replicate our work on other datasets and with different behavioural code smells and mobile platforms. However, this is the first study of its kind on this issue. Another main threat to external validity is the dataset used in the study, which is the same external validity from Habchi *et al.* [18] where the dataset is taken from. We used a set of 318 open-source Android apps from F-Droid, which is relatively small. It would have been preferable to also consider closed-source apps to build a more diverse and representative dataset. However, we did not have access to any proprietary software that can serve this study. We also encourage future studies to consider other datasets of open-source apps to extend this study. However, the number of apps is reasonable and consistent compared to the tools validation in the literature. In comparison, aDoctor was evaluated with 18 apps [38] and Paprika with 106 apps [20].

Repeatability/Reliability Validity. The results of the validation are repeatable and reliable because we use freely open-source programs that can be freely downloaded from the internet. The results are available in our artefacts [1].

Generalisability. The main threat to generalisability is that only two tools are considered, Paprika and aDoctor. However, both tools are popular, available, representative of the techniques used in the literature and have been widely studied by the scientific community. They are therefore representative

Challenge	Associated studied code smells
Identify code smells characterised by the use of a method call or a sequence of method calls.	DW
Identify code smells characterised by runtime information.	HAS, HBR, HSS, IOD, NLMR
Identify code smells characterised by undesired data variations during execution.	HMU

Table 5 Challenges of using static analysis to detect behavioural code smells.

of the tools of concern for the detection of Android code smells using static analysis. The use of other tools for further validation should be considered.

4 Lessons learned from empirical study

We have seen the issues with behavioural code smells detection rules in aDoctor and Paprika. However, as we detail in Threats to Validity in Section 3.6, we can generalise the findings to other static detection tools. The challenges encountered are related to the three categories of behavioural code smells identified earlier in the study. The challenges with their associated studied code smells are depicted in Table 5. Through static analysis, we can hardly detect code smells characterised by the use of a method call or a sequence of method calls, such as DW where an *acquire* must be followed by a *release*. It is also difficult to detect code smells characterised by runtime information. For example, HAS, HSS and HBR where the execution of the methods should not be too long or blocking. This is also the case for IOD where the execution time should be short and NLMR where memory should be released. Finally, it is challenging to detect code smells characterised by undesired data variations during execution. For example, for HMU, the size of an *ArrayMap* should not become excessively large and *HashMap* should not be used for short structures. The findings can therefore be generalised to any behavioural code smells that fall into one of these three categories.

At this point, we have seen the limitations of static-only detection for code smells that show a strong behavioural aspect. Several approaches can be used to consider the behaviour of the app. Modelling the application's behaviour, applying model checking or performing dynamic analysis. We will explore solutions through dynamic analysis.

First, for each behavioural code smell discussed in this paper, we show how such a dynamic detection can be done. Then, we discuss and look further at which state-of-the-art code smells are behavioural and could use a dynamic analysis.

4.1 Recommendations on Code Smells Studied

From the analysis and the different feedback of the real app examples, we show how the different behavioural code smells studied could be addressed by a dynamic analysis.

DW: A static analysis makes it rather difficult to know if the two methods, *acquire* and *release* are called during execution. Looking at the execution trace by a dynamic analysis would permit to see if these two methods are properly called. If they are properly called, we can verify if the time between the two calls is not excessively long. Furthermore, one could also check that the battery has not been drained too heavily by monitoring the battery.

NLMR: Here the goal is to determine when the app is in the background or when it starts to run out of memory, if it frees the unnecessary resources. A dynamic analysis could hence monitor, during the execution of the app, that in such cases the memory is really freed. Such a dynamic analysis would complete a static analysis, which already allows detecting the classes that have not defined the method.

HMU: In conformance with the initial definition, it is better to use *SimpleArrayMap* or *ArrayMap* for small sets up to hundreds of elements, and use *HashMap* for bigger instances. A dynamic analysis consists in verifying two assertions by monitoring the evolution of the content of the instances of these structures. Firstly, if a *SimpleArrayMap* at a given time exceeds the limit given by the Android recommendations, then we report a code smell. Secondly, if during an entire execution of the app, an *HashMap* is below this limit then a code smell is raised.

HBR, HAS and HSS These code smells, which are very similar, indicate that we do not expect these methods to be too time-consuming or blocking because they run on the main process and not on a dedicated process. Rather than looking statically at whether they seem too long using metrics such as the number of instructions, a dynamic analysis would be more precise to determine if a method was indeed too time-consuming during the execution of the app.

IOD: Whether the *onDraw* method has a too long execution time or is too memory-hungry, it is complex to determine without taking into consideration the execution of the app. A static analysis can simply give indications by looking at specific initialisations or operations. A dynamic analysis could tell, by observing the execution of the app, whether the *onDraw* method is too time-consuming or memory-hungry. The search for memory allocations by initialisation is, however, possible by a static analysis by searching for instructions involving memory allocation.

Tool	Could Use	Do not need
Paprika (13)	**7**(**55**%) NLMR, HMU, IOD, HAS, HSS, HBR, UIO	**6**(**45**%) MIM, LIC, UCS, UHA, IGS, IWR
aDoctor (15)	**5**(**33**%) DW, NLMR, DTWC, UC, LT	**10**(**67**%) DR, IDFP, IDS, ISQLQ, LIC, MIM, PD, RAM, SL, IGS
Total (28)	12(43%)	16(57%)

Table 6 Number of code smells that could use or do not need a dynamic analysis for the detection.

4.2 Code Smells on the State of the Art

The code smells detected by Paprika usually offer greater opportunity for dynamic detection. This is due to the nature of the smells addressed by Paprika, with smells impacting the performance, the graphic display, the memory and the process blocking. These kinds of smells have a clear behavioural aspect suggesting that inspecting the execution of apps will lead to a better detection.

As for the aDoctor tool, it focuses on different types of code smells that are usually related to the good uses of Android-specific methods. Yet, some code smells still could benefit from a dynamic detection, but most of them do not need it because it often consists of checking the presence of methods or replacing some methods with another one. We recall that Paprika and aDoctor have four code smells in common: NLMR, MIM, LIC and IGS. Static and dynamic analysis are not exclusive, both methods can be combined to detect code smells.

Table 6 gives the number of code smells detected by the two tools that could use or do not need dynamic techniques for a detection closer to the reality. These results are obtained by an analysis of the definition of the code smells, among those for which the behaviour of the app is to be taken into consideration to know if the code smell is well respected. Above all, the nature of the returned results is taken into consideration, as well as the extent to which the detected instances must be verified. Many code smells (57%) are not suitable for dynamic techniques due to their definition consisting of checking the presence of a method or an attribute. For example, Unsupported Hardware Acceleration (UHA) is a code smell that suggests avoiding *drawPath* operation of the Android class *Canvas*, advising replacing it by multiple *drawLine* calls. Such a description only needs a static analysis. The code smells that do not need dynamic analysis are MIM, LIC, UCS, IGS, IWR and UHA for Paprika and DR, IDFP, IDS, IGS, ISQLQ, LIC, MIM, PD, RAM and SL for aDoctor. Meanwhile, nearly 43% of the code smells seem to have an insufficient detection with the current methods due to their dynamic nature, like the seven studied code smells in this paper. In addition to the code smells studied in the paper,

the code smells in this category are UIO for Paprika, and DTWC, UC and LT for aDoctor. The entries in this table are correlated with those in Table 1.

We decide to put aside the code smells described as security smells [12]. Even if there are many described, we concentrate on the smells impacting the quality of the app rather than the security. Indeed, security code smells usually consist of the use of a safer method/structure/class, along with checking the presence of a parameter or a variable. The triviality of these code smells does not seem to be a good fit for dynamic techniques.

Taking into consideration the dynamic aspect makes it possible to extract new code smells that were not previously categorised. In particular, code smells concerning memory, execution time, process blocking. This is because the other classifications of Android-specific code smells in the state of the art did not describe code smells that they could not detect at all.

Although dynamic analysis is more precise than static analysis, it requires executing the app. However, we must take into consideration that in many cases, developers do not have ready-to-go test cases. Using dynamic analysis could take more time and also requires artefacts not always so common in mobile apps, like test cases.

Finally, some code smells, such as HMU and HSS, define some thresholds in their detection rules. For example, Paprika detects the HSS code smell by computing two metrics, the number of instructions and the cyclomatic complexity. It defines a threshold for both of them. Each method having a number of instructions greater than 17 or cyclomatic complexity greater than 3.5 is detected as a code smell. Through dynamic analysis, more appropriate metrics can be applied. For the HSS code smell, the execution time can be taken into consideration. Detecting them dynamically would still be difficult as we still have to define a threshold, like the threshold for the execution time for the HSS code smell. However, a dynamic detection allows having more appropriate metrics and thresholds, like execution time or released memory, which leads to a better detection for the behavioural code smells.

5 Conclusion and Future Work

In this paper, we presented an empirical study on the detection of code smells within mobile apps. As this study is the first of its kind on the detection of mobile code smells, it is still conducted on a small scale because it requires extensive manual work. We analysed and compared Paprika and aDoctor on their approach, both of which are based on static analysis. We focused only on specific code smells concerning the behaviour of the app. We showed that the tools return false negatives and false positives for these behavioural code smells, which affect the effectiveness of the tools. We have shown with a qualitative study on specific code smells that some apps that were not flagged as having smells, did not, however, meet the literal definition of the smells. A non-negligible portion of the detected code smells are false positives, i.e., detected code smells that should not have been detected. We have investigated

the detection rules to discover the problems that cause these errors in the detection. Moreover, we discussed the impacts of these detection rules on the results. This is the first preliminary study of its kind on this topic, to raise awareness in the community to provide detection tools that are more adapted to address behavioural code smells. It is essential to avoid false positives and false negatives to assist the developer as much as possible in the development of the app. Besides, we promote the need to consider behaviour in the area of Android code smells detection where many concerns are essentially behavioural. In future work, we are planning to extend our study to more apps and code smells of a dynamic nature not handled by the current tools. We are also planning to propose an automated tool-based approach that detects behavioural code smells through dynamic analysis.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Study artifacts. https://figshare.com/s/8235a0575ff4a88f0deb (2021). Online, Accessed: February 2022
2. Almalki, K.: Bad droid! an in-depth empirical study on the occurrence and impact of android specific code smells. Ph.D. thesis, Rochester Institute of Technology (2018)
3. Banerjee, A., Chong, L.K., Chattopadhyay, S., Roychoudhury, A.: Detecting energy bugs and hotspots in mobile apps. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (2014)
4. Dennis, C., Krutz, D.E., Mkaouer, M.W.: P-lint: A permission smell detector for android applications. 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft) pp. 219–220 (2017)
5. Developer.Android: Onlowmemory. https://developer.android.com/reference/android/content/ComponentCallbacks#onLowMemory(). Online, Accessed: May 2021
6. Developer.Android: Arraymap. https://developer.android.com/reference/android/ support/v4/util/ArrayMap.html (2015). Online, Accessed: May 2021
7. Elsayed, E.K., ElDahshan, K.A., El-Sharawy, E.E., Ghannam, N.E.: Reverse engineering approach for improving the quality of mobile applications. PeerJ Computer Science **5** (2019)
8. Emden, E.V., Moonen, L.: Java quality assurance by detecting code smells. Ninth Working Conference on Reverse Engineering, 2002. Proceedings. pp. 97–106 (2002)
9. Fard, A.M., Mesbah, A.: Jsnose: Detecting javascript code smells. 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM) pp. 116–125 (2013)
10. Fowler, M.: Refactoring - Improving the Design of Existing Code, 1 edn. Addison-Wesley (1999)
11. Gadient, P., Ghafari, M., Frischknecht, P., Nierstrasz, O.: Security code smells in android icc. Empirical Software Engineering pp. 1–31 (2018)
12. Ghafari, M., Gadient, P., Nierstrasz, O.: Security smells in android. 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM) pp. 121–130 (2017)
13. Gottschalk, M., Josefiok, M., Jelschen, J., Winter, A.: Removing energy code smells with reengineering services. In: GI-Jahrestagung (2012)
14. Haase, C.: Developing for android ii the rules: Memory. https://medium.com/google-developers/developing-for-android-ii-bb9a51f8c8b9 (2015). Online, Accessed: May 2021

15. Habchi, S., Blanc, X., Rouvoy, R.: On adopting linters to deal with performance concerns in android apps. 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE) pp. 6–16 (2018)
16. Habchi, S., Hecht, G., Rouvoy, R., Moha, N.: Code smells in ios apps: How do they compare to android? 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft) pp. 110–121 (2017)
17. Habchi, S., Moha, N., Rouvoy, R.: The rise of android code smells: Who is to blame? 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR) pp. 445–456 (2019)
18. Habchi, S., Rouvoy, R., Moha, N.: On the survival of android code smells in the wild. 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft) pp. 87–98 (2019)
19. Hecht, G.: Détection et analyse de l'impact des défauts de code dans les applications mobiles. (detection and analysis of impact of code smells in mobile applications). Ph.D. thesis, Université du Québec à Montréal (2016)
20. Hecht, G., Rouvoy, R., Moha, N., Duchien, L.: Detecting antipatterns in android apps. 2015 2nd ACM International Conference on Mobile Software Engineering and Systems pp. 148–149 (2015)
21. Iannone, E., Pecorelli, F., Nucci, D.D., Palomba, F., Lucia, A.: Refactoring android-specific energy smells: A plugin for android studio. Proceedings of the 28th International Conference on Program Comprehension (2020)
22. Ibrahim, R., Ahmed, M., Nayak, R., Jamel, S.: Reducing redundancy of test cases generation using code smell detection and refactoring. Journal of King Saud University - Computer and Information Sciences **32**(3), 367–374 (2020)
23. Johnson, B., Song, Y., Murphy-Hill, E.R., Bowdidge, R.W.: Why don't software developers use static analysis tools to find bugs? 2013 35th International Conference on Software Engineering (ICSE) pp. 672–681 (2013)
24. Kessentini, M., Ouni, A.: Detecting android smells using multi-objective genetic programming. 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft) pp. 122–132 (2017)
25. Kumar, S., Chhabra, J.K.: Two level dynamic approach for feature envy detection. 2014 International Conference on Computer and Communication Technology (ICCCT) pp. 41–46 (2014)
26. Lim, D.: Detecting code smells in android applications. Master's thesis, TU Delft (2018)
27. Lin, Y., Okur, S.: Study and refactoring of android asynchronous programming (2015)
28. Malavolta, I., Verdecchia, R., Filipovic, B., Bruntink, M., Lago, P.: How maintainability issues of android apps evolve. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) pp. 334–344 (2018)
29. Mannan, U.A., Ahmed, I., Almurshed, R.A.M., Dig, D., Jensen, C.: Understanding code smells in android applications. 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft) pp. 225–236 (2016)
30. Marinescu, C., Marinescu, R., Mihancea, P.F., Ratiu, D., Wettel, R.: iplasma: An integrated platform for quality assessment of object-oriented design. In: ICSM (2005)
31. Mariotti, G.: Antipattern: freezing a ui with broadcast receiver. http://gmariotti.blogspot.ca/2013/02/antipattern-freezing-ui-with-broadcast.html (2013). Online, Accessed: May 2021
32. Mariotti, G.: Antipattern: freezing the ui with a service and an intentservice. http://gmariotti.blogspot.com/2013/03/antipattern-freezing-ui-with-service.html (2013). Online, Accessed: May 2021
33. Mariotti, G.: Antipattern: freezing the ui with an asynctask. http://gmariotti.blogspot.com/2013/02/antipattern-freezing-ui-with-asynctask.html (2013). Online, Accessed: May 2021
34. Moha, N., Guéhéneuc, Y.G., Duchien, L., Meur, A.F.L.: Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering **36**, 20–36 (2010)
35. Morales, R., Saborido, R., Khomh, F., Chicano, F., Antoniol, G.: [journal first] earmo: An energy-aware refactoring approach for mobile apps. 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE) pp. 59–59 (2018)

36. Ni-Lewis, I.: Avoiding allocations in ondraw() (100 days of google dev). https://youtu.be/HAK5acHQ53E (2015). Online, Accessed: May 2021
37. Oracle: Java thread primitive deprecation. https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/thread PrimitiveDeprecation.html (2020). Online, Accessed: May 2021
38. Palomba, F., Nucci, D.D., Panichella, A., Zaidman, A., Lucia, A.D.: Lightweight detection of android-specific code smells: The adoctor project. 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) pp. 487–491 (2017)
39. Paternò, F., Schiavone, A.G., Conte, A.: Customizable automatic detection of bad usability smells in mobile accessed web applications. Proceedings of the 19th International Conference on Human-Computer Interaction with Mobile Devices and Services (2017)
40. Peruma, A.S.: What the smell? an empirical investigation on the distribution and severity of test smells in open source android applications (2018)
41. Rasool, G., Ali, A.: Recovering android bad smells from android applications. Arabian Journal for Science and Engineering **45** (2020). DOI 10.1007/s13369-020-04365-1
42. Reimann, J., Brylski, M., Aßmann, U.: A tool-supported quality smell catalogue for android developers. Softwaretechnik-Trends **34** (2014)
43. Rubin, J., Henniche, A.N., Moha, N., Bouguessa, M., Bousbia, N.: Sniffing android code smells: An association rules mining-based approach. 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft) pp. 123–127 (2019)
44. Statista: Number of apps available in leading app stores as of 1st quarter 2020. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/ (2020). Online, Accessed: May 2021
45. Statista: Number of mobile app downloads worldwide from 2016 to 2019. https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/ (2020). Online, Accessed: May 2021
46. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B.: Experimentation in software engineering. In: The Kluwer International Series in Software Engineering (2000)

COMMENTS FOR THE AUTHOR:

One reviewer has already given accept in the last round and Reviewer 2 in this round also gave accept. Reviewer 1 still has a minor comment, which is about the use of wording "source code describing an inappropriate behaviour" being misleading. Please clarify if it's about API misuse or it's just the semantics of the code is inappropriate.

I will check the revision on this minor point without the need of forwarding to reviewers for another round.

Reviewer 1: I would like to thank the authors for carefully addressing my comments in the previous revision. In general, I'm mostly happy with the revisions and clarifications.

After reading the clarification on R1C3, I still hold reservations on what exactly "describing" means. I'm still not sure whether I correctly understand what the authors mean. It seems that the code should explicitly show some API misuse and the authors are defining such misuses as inappropriate behaviours. So, the word "behaviour" is not referring to app runtime behaviour? If this is the case, for IDS, why can't we consider "the use of HashMap<Integer,Object>" itself as an inappropriate behaviour and thereby, IDS is also describing an inappropriate behaviour?

I do hope that the authors can further polish this definition in the final version.

It is true that the term "behaviour" can be confusing as it can have a dual meaning. As stated by the reviewer, it can refer to the app runtime behaviour, for example a low performance of the app as in the IDS code smell. In the current paper, we use the term "software quality" to refer to such app runtime behaviour but also to refer to other concerns involving performance, energy consumption, and memory. The term "software quality" is thus more general and includes app runtime behaviour (also referring to performance). But in our case, we use "behaviour" to refer specifically to code execution behaviour, i.e. an occurrence or a sequence of observable code events or actions during execution. For example, the DW code smell describes the following inappropriate behaviour: A call to the acquire method is not followed by a call of the release method.

All studied code smells (including behavioural and non-behavioural ones) have a bad software quality impact. However, behavioural code smells have the specificity to refer to characteristics in the source code inducing an inappropriate code behaviour during the execution. The code characteristics can refer to API misuse but not only, it can also refer to common code usage, for example the use of specific code structures such as *HashMap* vs. *SimpleArrayMap/ArrayMap*.

We refine the definition of "behavioural code smells" to remove any ambiguity. We also rephrase the related explanation of the IDS code smell.