An Empirical Study on the Impact of Android Code Smells on Resource Usage

Johnatan Oliveira[1], Markos Viggiato[2], Mateus Santos[1],
Eduardo Figueiredo[2], Humberto Marques-Neto[1]

[1]Department of Computer Science, Pontifical Catholic University of Minas Gerais (PUC Minas)
[2]Department of Computer Science, Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
{johnatan.oliveira,mateus.freira}@sga.pucminas.br, {markosviggiato, figueiredo}@dcc.ufmg.br,
humberto@pucminas.br

Abstract

Code smells are symptoms that something may be wrong with the app. Aiming at removing code smells and improving the maintainability and performance of the app, we may apply the refactoring technique, which could reduce hardware resource use, such as CPU and memory. However, a few studies have evaluated the impacts of the refactoring in Android. This paper presents a study to assess the effects of smartphone resource use caused by refactoring of 3 classic code smells: God Class, God Method, and Feature Envy. To this purpose, we selected 9 apps from GitHub. The results show that refactoring used in desktop software may not be appropriate for Android apps. For example, the refactoring of God Method had increased CPU consumption by more than 47%, while the refactoring of the 3 code smells reduced memory consumption in average 6.51%, 8.4%, and 6.37%, respectively, in one app. Our results can support the community in conducting research and future implementation of new tools. Also, it guides app developers in refactoring and thus improving the quality of their apps.

keywords: Code Smells, Android Smells, and Consumption of Smartphone Resources

1 Introduction

In recent years, mobile applications (apps) have become one of the most popular and profitable software products in society, especially context-aware apps that are pervasive in people's lives [10]. Applications developed for the Android platform are mainstream [13]. This platform has around 80% of the market for mobile Operating Systems (OS), and it has far surpassed its main competitor Apple [10]. One of the critical factors for the success of this software segment is related to the ease of rapidly developing and making the apps available to millions of users with lower costs [12].

Mobile app development is different from desktop software development since the mobile devices, like smartphones and tablets, have limited resources, such as CPU, battery, and memory [10]. Furthermore, app development is affected by deadlines even more restricted than those for desktop, once apps are usually designed for many users that get used to regular addition of new features and high speed in the correction of flaws [1, 7]. Mobile apps of context-aware can identify the context in which they are inserted and adapt their behavior according to the environment [4, 12].

When these apps are erroneously programmed, they can quickly drain device resources, such as the memory and CPU [13]. The presence of code smells can lead to poor software quality, which makes the evolution of features harder, deteriorated quality, and therefore, causes a bad experience for the final user [9]. In the literature, the presence of these issues in Android apps is known as Android smells [3].

Performance and optimization of the resources are crucial factors for the success of mobile apps [10]. Users can uninstall the app from the device if it starts to lock-in or drains resources quickly [3, 10]. Therefore, correction of Android smells can not only improve the performance of the apps without affecting their behavior and also improve user experience [3, 9, 10]. For this reason, applying refactoring techniques in Android smells could contribute to the evolution and maintenance of these apps.

Previous works [1, 2, 10] investigated the impact of resources usage, such as memory, battery, and CPU. However, we still lack empirical study on the effect of fixing Android smells through refactoring, mainly concerning memory and CPU usage. Studies regarding automatic detection and analysis of the impact of the refactoring on the Android platform are still premature [5].

This paper starts to fill this gap by analyzing the impacts caused by the adoption of refactoring in the Android platform. For this purpose, we selected 100 mobile apps from GitHub and filtered this data set to choose the most representative apps, because the focus this paper are context-aware apps. After the filtering process, 9 apps remained to be analyzed. To detect code smells in the Android platform, we rely on JDeodorant tool, and 3 classic code smells for analysis: God Class, God Method, and Feature Envy. We believe our results might better support the community in conducting researches about Android smells and also guide app developers in refactoring activity, aiming to improve the quality of the mobile apps.

DOI reference number: 10.18293/SEKE2018-157

2 Background

This section presents an overview of the main concepts used in this paper. Section 2.1 introduces the 3 selected code smells. Section 2.2 describes some the tools able detect and refactor code smells.

2.1 Code Smells

A code smell is any symptom that may indicate a deeper quality problem in the software [9]. The code smells documented by Fowler [9] are considered classic in object-oriented software. In this study, we evaluated 3 types of code smells, God Class, God Method and Feature Envy, mainly because of two reasons. First, these code smells are classic in software engineering. Second, several tools are able to detect these code smells. However, a few tools can detect and automatically apply refactoring. Next, we present the selected code smells.

God Class occurs when a class has many attributes or methods in its interface, and it does not use all of them [9]. Usually, in this kind of situation, it is possible to extract part of these attributes or methods to another class, thus separating responsibilities and leaving them more coherent [11]. *God Method* can be described as a long method or a method that does more than a task. Therefore, it is not related only to the size of the method itself [9]. This code smell might get solved by creating smaller methods and moving code from the main method to others, keeping the same behavior [11]. *Feature Envy* is a code smell that occurs when a method uses more attributes or methods from another class than from its own class [9]. The suggested refactoring for this code smell is to move the envy method, replacing then a large number of calls to the other class by only one call to the moved method [9].

2.2 Automated Code Smell Detection and Refactoring

Several tools have been developed to automate the process of detecting code smells. For instance, PMD [8], which is a general purpose tool for code smells detection in some languages, such as Java and JavaScript. However, a few tools are able to apply refactoring technique automatically. Therefore, we selected JDeodorant tool because it is able to detect and apply the refactoring technique automatically to the 3 kinds of code smells selected.

Some tools are specific to Android platform, such as aDoctor [14]. Rigid Alarm Manager, Durable Wake lock, and Debuggable Release are some examples of code smells detected by aDoctor. These code smells would not occur on other platforms because they refer to Android-specific problems. We have not analyzed these Android-specific code smells because our goal is to target well-known code smells as discussed in Section 2.1. Besides, aDoctor cannot perform refactoring automatically.

3 Study Settings

This section describes the evaluation settings. Section 3.1 presents the study goal and the research questions. Section 3.2 presents the evaluation steps. Section 3.3 presents our data set.

3.1 Goal and Research Questions

The primary goal of this study is to evaluate whether refactoring in Android improves the source code concerning the use of mobile devices resources. In particular, we aim to verify if this technique can reduce the consumption of CPU and memory. Based on this goal, we also conceived the following research questions (RQs) to guide our study.

RQ1. *Does the refactoring of Android code smells improve the CPU consumption of the Smartphone?*

RQ2. *Does the refactoring of Android code smells improve the memory consumption of the Smartphone?*

3.2 Evaluation Steps

We analyze the impacts of applying the refactoring in Android from a set of 9 apps. To minimize the risk of bias in our study, we executed each app 18 times, and we use the arithmetic mean (and its standard deviation) to evaluate the impact of refactoring. The executions were conducted 3 times for each code smell before applying the refactoring (resulting in 3*3=9 executions) and 3 times for each code smell after using the refactoring, i.e., a total of 18 performances for each app. Since we ran each app 18 times and given that we have 9 apps, we performed a total of 162 executions. Due to the high number of executions and the fact that all tests are performed manually, it would not be feasible to investigate a higher number of apps at this moment, and therefore we kept our data set with the selected apps.

To performer our study, we used a smartphone running Android OS version 5.1, model Moto G XT1032, equipped with a quad-core CPU of 1.2 GHz. The smartphone consists of 1 GB DDR3 of RAM and 16 GB of disk. The original OS of this smartphone was Android 4.3, called Jelly Bean, with later updates to 5.1. We considered this smartphone suitable for our experiments, mainly because there is a lot of apps compatible with it from Google Play Store[1]. We used in our work a smartphone with only essentials apps of the Android OS to avoid interference in the results of other apps. Moreover, for each execution, the app was uninstalled and installed again with the Android APK of the version under analysis. By following this procedure, all user data were erased at the beginning of a new test, and each execution of the test had a similar initial state. In our study, we performed five steps to analyze the adopted refactoring. These steps are illustrated in Figure 1 and described as follows.

(1) Code smell detection – We run the selected tool called JDeodorant to detect the 3 types of code smells. In

[1]https://play.google.com/store

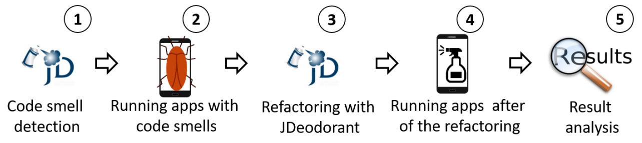
1
Code smell detection
2
Running apps with code smells
3
Refactoring with JDeodorant
4
Running apps after of the refactoring
5
Results
Result analysis

Figure 1: Evaluation Steps

this step, each app was imported into Eclipse, because the detection tool is a plug-in exclusive for this IDE. After importing these apps, we identified the 3 types of code smells in each app. We create a list of classes, methods, and code smells of each analyzed app.

(2) Running apps with code smells in Android – From the data obtained in the previous step, we run each app individually, 18 times on the smartphone. At the end of the 18 tests per app, we achieve the results of memory use and CPU use through the mean of 3 executions in each app. We collect the data about use of CPU and memory through an app named *AnotherMonitor*, version 3.1.0. *AnotherMonitor* was selected for several reasons. First, this app is compatible with the Android version evaluated. Second, it has an excellent reputation in Play Store with more than 4 stars. Third, it is possible to export the results about CPU and Memory usage to a CSV file.

(3) Refactoring using JDeodorant –After the previous steps, we obtained the consumption of the apps in the smartphone, such as CPU and memory. In this step, we re-import the apps to Eclipse, similar to Step 1, to apply the automated refactoring with JDeodorant to the 3 selected code smells.

(4) Running of the Apps after of the Refactoring – After applying the automated refactoring using the JDeodorant tool, we evaluated whether the consumption of memory and CPU improved or deteriorated. These characteristics of usage can be assessed based on the access provided by Android. To this goal, we run again each app 18 times and compute the average memory and CPU consumption.

(5) Result analysis – At the end of all the previous steps, we obtained the data regarding the consumption of resources in Android before and after refactoring. Through these results, it is possible to compare the feasibility of this technique in mobile devices.

3.3 Corpus of Android Apps

In order to investigate the impacts caused by Android smells and refactoring techniques on resource consumption, we chose only apps from the context-aware domain for several reasons. First, there are many apps available for download on GitHub. Second, these apps are actively used. Also, the authors of this paper believed that it would be easy to find code smells in this app domain. The reason might be because this type of app can contain source code with high complexity caused by the use of several sensors to identify characteristics of the context [16]. Third, one of the features of this type of app is the need to use various hardware resources to identify the context, such as GPS and movement sensors [6].

The apps that compose our data set were retrieved from GitHub in November 2017. We searched for apps of context-aware sorted by stars. To retrieve apps, we used the following keywords related to the context-aware domain: *context-aware*, *context awareness*, and *pervasive computer*.

To minimize the risk of biasing our results, we apply a strict set of criteria for defining our data set as illustrated in Figure 2 and described by the three phases as follows.

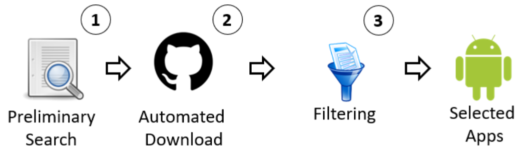

Figure 2: Phases for Collecting Apps from GitHub

Phase 1: Preliminary Search – We performed a preliminary search in GitHub to evaluate the feasibility of collecting these apps of the context-aware domain. We conducted this phase manually to identify the diversity of apps on GitHub. Also, this pre-evaluation was necessary to obtain the domain variation names.

Phase 2: Automated Download – We implemented an algorithm to clone the apps from GitHub automatically. This phase is necessary because we know that several apps it are hosted on GitHub, and manual cloning would be unfeasible. In this phase, we obtained 100 apps.

Phase 3: Filtering – From the cloned apps, we used the following exclusion criteria to filter these apps: i) non-Java app, once the chosen tool requires apps developed in Java, ii) apps with less than 1k lines of code (LOC), because we considered that these apps might represent only toy apps, iii) apps that are not compatible with Eclipse, since the selected tool requires the use this IDE, and iv) all apps that required login due to evaluation convenience. After applying all these filters, we obtained 9 apps able to be evaluated.

4 Results and Discussion

This section discusses the results obtained in this study. Section 4.1 presents of amount of code smells detected in each app. Section 4.2 focuses on answering each research question. Section 4.3 presents some guidelines to support future implementation of tools.

4.1 Detection of Code Smells

In this section, we present the results about code smells detected by JDeodorant tool in our data set. Table 1 shows the occurrences for each code smell investigated in this study. In the last column of Table 1, we show the total number of code smells for each app, and the last line presents the total number of occurrence of each code smell across all the apps. By looking at the column Total, we note that RunnerUp was the app with the highest number of code smells. Besides, in the line Total, it is possible to identify that the code smells Feature Envy and God Class occurred most frequently, with 102 and 94 occurrences, respectively. For instance, Feature Envy appeared 34 times in RunnerUp and 27 times in Activity Tracker, while God Class showed up 50 times in RunnerUp and 15 times in Calendula.

Table 1: Code Smells Detected

App	God Class	God Method	Feature Envy	**Total**
Activity Tracker	4	0	27	31
Calendula	15	8	14	37
CycleStreets	13	2	8	23
Forecastie	3	0	1	4
NoiseApp	0	1	1	2
Pedometer	1	2	3	6
RunnerUp	50	16	34	100
Steptastic	5	2	3	10
Travel-Mate	3	1	11	15
Total	94	32	102	

4.2 Answering the Research Questions

In a first moment, we present the results regarding the CPU use. Therefore, we answer RQ1 as follows. *Does the refactoring technique of Android code smells improve the CPU consumption of the Smartphone?*

Tables 2 to 4 present some descriptive statistics of the experiments results. The column *Mean* ($\bar{x}$) presents the mean of the analyzed data. A *Standard Deviation* is represented by σ. Also, all columns have letter S or R. S means app with code smells, and R stands refactored apps

Table 2 shows a particular case of the one app named CycleStreets that before refactoring the average use of CPU was 26.2% and it started to use 32.47% of CPU after the refactoring. The consumption of CPU increased by more than 6%. This number may seem somewhat small, but we are investigating code smells on a platform that already has limited resources, and any unnecessarily consume harmfully. It is also possible to observe that from 9 apps evaluated, the consumption of CPU increased in 7 apps and drastically decreased in only one app (Steptastic). Also, the standard deviation changed from 11.64 to 14.11 in CycleStreets, an increase of 2.47%.

Table 2: Descriptive Statistics for CPU (God Class)

App	$\bar{x}$ (S)	$\bar{x}$ (R)	σ (S)	σ (R)
Activity Tracker	17.07	20.01	6.42	4.8
Calendula	16.35	17.44	7.74	9.2
CycleStreets	26.2	32.47	11.64	14.11
Forecastie	17.07	20.01	6.42	4.8
NoiseApp	22.55	23.65	12.66	12.54
Pedometer	19.47	21.17	9.75	10.43
RunnerUp	14.26	17.33	7.34	9.42
Steptastic	24.77	15.12	13.23	9.27
Travel-Mate	19.7	19.61	10.01	10.34

Regarding God Method, Table 3 shows the results achieved. In this table, we may highlight the Forecastie app, in which the consumption of CPU almost doubled. As we can observe, the use of CPU in this app soared from 25.43% to 47.14%. However, for the app Steptastic, it was possible to reduce CPU consumption by more than 12% through the refactoring technique. Finally, refactoring God Method caused an increase in CPU consumption in 5 apps and a decrease in 3 apps.

Table 3: Descriptive Statistics for CPU (God Method)

App	$\bar{x}$ (S)	$\bar{x}$ (R)	σ (S)	σ (R)
Activity Tracker	26	30.93	14.68	16.82
Calendula	20.97	26.08	13.23	15.51
CycleStreets	25.29	34.01	14.23	20.35
Forecastie	25.43	47.14	15.72	12.87
NoiseApp	26.49	26.24	16.18	15
Pedometer	32.98	28.17	19.3	16.3
RunnerUp	19.32	26.42	10.37	15.9
Steptastic	30.87	18.16	18.09	12.11
Travel-Mate	22.35	17.77	12.64	11.08

Table 4 presents the results for Feature Envy. In all apps, but Travel-Mate it was possible to improve source code quality through refactoring technique. Overall, it was possible to improve the source code quality with an average increase of less than 1% of use of resources.

Table 4: Descriptive Statistics for CPU (Feature Envy)

App	$\bar{x}$ (S)	$\bar{x}$ (R)	σ (S)	σ (R)
Activity Tracker	16.82	16.88	8.11	8.14
Calendula	19.14	20.82	10.35	9.33
CycleStreets	21.52	23.09	10.94	11.61
Forecastie	23.24	24	12.28	11.6
NoiseApp	20.02	22.86	10.55	12.11
Pedometer	25.13	23.84	12.52	13.07
RunnerUp	21.29	21.56	11.89	11.89
Steptastic	21.55	21.68	11.54	12.04
Travel-Mate	16.35	17.44	7.74	9.2

We believe that the refactoring technique adopted by JDeodorant tool and as indicated by Fowler [9] is not appropriate to use in Android since most apps consumed more CPU after we performed refactoring. A potential reason for the increase can be found in the Android's site of the excellent programming practice[2]. For instance, when there are several recursive calls, it is necessary to call the `onDestroy()` method more times, which may increase CPU usage since it will be required to destroy more objects.

The refactoring adopted to solve this problem is Extract Method. This refactoring consists in splitting the God Method and creating a new method [9]. Therefore, applying the refactoring Extract Method may cause the app to invoke these resources several times, increasing CPU consumption. After presenting the use of CPU of each app under analysis, we show the results concerning the memory consumption. Therefore, we answer RQ2 as follows. *Does the refactoring technique of Android code smells improve the memory consumption of the Smartphone?*

[2]https://developer.android.com/training/best-performance.html

In general, the adoption of the refactoring increased the memory usage, but in some cases, it can help to avoid unnecessary use of resources. We believe that the refactoring technique in the case of the app Steptastic decreases the memory usage because of two main reasons. First, Steptastic, for example, is smaller regarding LOC and their methods are well distributed among classes. Second, this app has only one developer, which may be a indicate that the number of developers may be correlated with the number of code smells. Tables 5 to 7 presents the same configurations presented in RQ1.

Table 5 presents a similar result of Table 4, but this table shows the results regarding the code smell God Class and usage of memory. For 7 apps, namely: Calendula, CycleStreets, Forecastie, NoiseApp, Pedometer, RunnerUp, and Steptastic, it was possible to reduce the use of memory by 26, 26, 58, 13, 13, 7, and 33, respectively. These values were measured in megabyte (MB).

Table 5: Descriptive Statistics for Memory (God Class)

App	$\bar{x}$ (S)	$\bar{x}$ (R)	σ (S)	σ (R)
Activity Tracker	9.54	67.6	2.29	41.01
Calendula	60.45	34.24	17.89	12.55
CycleStreets	73.8	47.08	9.28	9.23
Forecastie	9.54	67.6	2.29	41.01
NoiseApp	145.22	159.02	24.29	21.09
Pedometer	144.82	158.24	24.63	20.86
RunnerUp	88.53	80.66	17.48	8.5
Steptastic	61.99	28.53	13.71	14.72
Travel-Mate	80.65	83.86	39.64	39.21

Table 6 shows the results of the use of memory concerning God Method. Overall, it is possible to observe that the results demonstrate an increase in memory usage after refactoring. Also, in this table, there is a specific case in which the app (Travel-Mate) used 98.39 MB before we apply refactoring and started to consume 625.58 MB after the refactoring. This number represents an increase of approximately 500 MB, i.e., the app began to use an alarming quantity of memory. Also, the app Activity Tracker had a hight disparity regarding the standard deviation.

Table 6: Descriptive Statistics for Memory (God Method)

App	$\bar{x}$ (S)	$\bar{x}$ (R)	σ (S)	σ (R)
Activity Tracker	94.38	268.38	22.17	148.84
Calendula	51.95	48.43	9.44	13.19
CycleStreets	74.1	47.91	10.65	11.05
Forecastie	93.49	41.46	15.36	0.04
NoiseApp	144.73	158.2	24.67	21
Pedometer	74.95	171.91	17.65	21.61
RunnerUp	87.38	81.09	17.49	9.33
Steptastic	61.95	29.89	13.48	15.45
Travel-Mate	98.39	625.58	28.34	13.64

In general, all descriptive statistics for memory showed that standard deviation varied too much, except for code smell Feature Envy. Table 7 presents, in general, a lower variation in relation to the standard deviation. A low standard deviation shows that the data are clustered tightly around the mean. Besides, in general, all apps started to use more memory after the refactoring, such as the Activity Tracker app.

Table 7: Descriptive Statistics for Memory (Feature Envy)

App	$\bar{x}$ (S)	$\bar{x}$ (R)	σ (S)	σ (R)
Activity Tracker	94.82	264.83	21.49	150.22
Calendula	77.5	98.59	21.35	38.26
CycleStreets	62.51	46.78	22.15	11.26
Forecastie	93.26	41.43	14.63	0.02
NoiseApp	143.39	158.69	23.85	20.93
Pedometer	74.33	171.7	17.92	22.2
RunnerUp	87.86	80.42	18.28	9.17
Steptastic	61.23	28.36	13.58	15.22
Travel-Mate	60.45	34.24	17.89	12.55

4.3 Guidelines for Android Smell Tools

From this study, we uncover 2 guidelines (G1 and G2) to support future implementation of tools for Android.

G1. In order to support results analysis through statistical methods, graphical visualization or at least numerical indicator, such as percentage – We identified that the evaluated tool does not show statistical numbers related to the code smells for each type. Also, when using the detection and refactoring tool in apps, we should be able to determine if the refactoring will lead to more methods calls. Then, if more method calls will be necessary, applying the refactoring will probably imply in increasing the consumption as the OS adds more data in the Android memory stack.

G2. In order to combine different techniques (token, tree, etc.) [14, 8, 15] for detecting several codes smells in Android – It could be interesting to combine different techniques that can be more useful to detect code smells. This characteristic may increase the precision of the tool, mainly in refactoring technique. A solution to this problem is to identify the number of cycles of the clock that refactoring can cause.

5 Related Work

Studies have investigated code smells in Android [1, 2, 10, 13]. For instance, Boussaa et al.[2] proposed an automated approach to generate rules based on software quality metrics and threshold to detect code smells in Android. This work argues that identifying Android code smells is extremely important, since the presence them may lead to higher use of CPU, memory, and battery. We also found other of papers related to refactoring and energy efficiency, considering a different type of refactoring. For instance, Banerjee and Roychoudhury [1] present a lightweight refactoring technique to assist app development regarding energy efficiency. Their results show that refactoring the app can reduce the energy consumption up to 29%.

Existing studies investigate the identification of code smells in Android and how they relate to energy efficiency. However, works to identify positive and negative impacts of refactoring on code smells regarding resources usage are premature. Furthermore, there is no study investigating both memory and CPU usage before and after applying the refactoring. In this context, our study investigates how the refactoring of code smells usually common on desktop systems may impact resources usage in Android.

6 Threats to Validity

We based our study on related works to support our research. However, some threats to validity may affect our research findings. We conducted careful filtering of context-aware apps from GitHub. Considering that the exclusion criteria for app selection were applied in an automatic process, we may have discarded compatible apps. In our viewpoint, the selected apps are representative, given that they are well-defined regarding the diversity of use of resources.

We used defaults configuration of JDeodorant to detect code smells and apply refactoring techniques automatically. Besides, another threat is concerning the use of the apps. We decided not to emulate them through Android Studio to achieve more trusted results. To test these apps, we have adopted a rigorous manual test script, whereby all apps have been tested. For example, if the app requires GPS positioning and the user walks with the smartphone, all other apps with the same characteristics have passed by the same route without any interference with the time or distance covered.

7 Conclusion and Future Work

The empirical study reported in this paper evaluated 9 different apps from the context-aware domain. These apps were tested 162 times. In particular, we analyzed 3 types of smells, namely God Class, God Method, and Feature Envy. We aimed to verify the possibility of improving the quality of the source code through refactoring technique and reduce the use of resources, such as memory and CPU.

Our findings point that the refactoring technique, in general, causes an increase in the use of resources. We also observed that the JDeodorant is not able to adequately perform the refactoring in Android since techniques usually adopted in desktop software are the opposite to the best-practices indicated by Android developers site. The tool used in this study was developed with the purpose of detecting the code smells and apply refactoring in Java, the same language used in the tested apps.

We believe that the results of this study will benefit developers by helping them to avoid inappropriate use of refactoring technique in the mobile device. Additionally, we provide two guidelines for developing a new tool able to detect code smells and apply refactoring. As future work, we plan to extend our study to investigate other code smells on Android. We also plan to develop an automated script to test different apps in large-scale studies and implement a tool based on the guidelines uncovered in this study.

8 Acknowledgments

This research was partially supported by Brazilian funding agencies: CAPES, CNPq (Grant 424340/2016-0), FAPEMIG (Grant PPM-00651-17 and APQ-02924-16), and FIP-PUC Minas.

References

[1] Abhijeet Banerjee and Abhik Roychoudhury. Automated refactoring of Android apps to enhance energy-efficiency. In *38th Proc. of the Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft)*, 2016.

[2] Mohamed Boussaa, Wael Kessentini, Marouane Kessentini, Slim Bechikh, and Soukeina Ben Chikha. *Competitive Coevolutionary Code-Smells Detection*. Springer Berlin Heidelberg, 2013.

[3] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy. Investigating the energy impact of Android smells. In *24th Proc. of the Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[4] L. Cruz and R. Abreu. Performance-based guidelines for energy efficient mobile applications. In *4th Proc. of the Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft)*, 2017.

[5] L. Cruz, R. Abreu, and J. N. Rouvignac. Leafactor: Improving energy efficiency of Android apps via automatic refactoring. In *4th Proc. of the Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft)*, 2017.

[6] Quan Chau Dong Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. Mybatrecommender: Automated optimization of energy consumption for android smartphones in software layer. In *13th Proc. of the Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE)*, 2016.

[7] Jacky Estublier. Software configuration management: A roadmap. In *22nd Proc. of the Conf. on The Future of Software Engineering (ICSE)*, 2000.

[8] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mntyl. Code smell detection: Towards a machine learning-based approach. In *29th Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, 2013.

[9] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of Android code smells. In *3th Proc. of the Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft)*, 2016.

[11] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Publishing Company, Incorporated, 2010.

[12] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen. Understanding code smells in Android applications. In *4th Proc. of the Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft)*, 2016.

[13] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. Anti-patterns and the energy efficiency of Android applications. *ArXiv e-prints*, 2, 2016.

[14] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. Lightweight detection of android-specific code smells: The adoctor project. In *24th Proc. of the Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[15] Santiago A. Vidal, Claudia Marcos, and J. Andrés Díaz-Pace. An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23, 2016.

[16] H. Zhu, H. Xiong, Y. Ge, and E. Chen. Discovery of ranking fraud for mobile apps. *IEEE Trans. on Knowledge and Data Engineering*, 27, 2015.