2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)

Evaluating the impact of code smell refactoring on the energy consumption of Android applications

Hina Anwar
Institute of Computer Science
University of Tartu
Tartu, Estonia
hina.anwar@ut.ee

Dietmar Pfahl
Institute of Computer Science
University of Tartu
Tartu, Estonia
dietmar.pfahl@ut.ee

Satish N. Srirama
Institute of Computer Science
University of Tartu
Tartu, Estonia
satish.srirama@ut.ee

Abstract— **Energy consumption of mobile apps is receiving a lot of attention from researchers. Recent studies indicate that energy consumption of mobile devices could be lowered by improving the quality of mobile apps. Frequent refactoring is one way of achieving this goal. We explore the performance and energy impact of several common code refactorings in Android apps. Experimental results indicate that some code smell refactorings positively impact the energy consumption of Android apps. Refactoring of the code smells 'Duplicated code' and 'Type checking' reduce energy consumption by up to 10.8%. Significant reduction in energy consumption, however, does not seem to be directly related to the increase or decrease of execution time. In addition, the energy impact over permutations of code smell refactorings in the selected Android apps was small. When analyzing the order in which refactorings were made across code smell types, it turned out that some permutations resulted in a reduction and some in an increase of energy consumption for the analyzed apps.**

Keywords—Refactoring, Software Maintenance, Code Smell Detection, Code Power Consumption, Code Smell Refactoring.

I. INTRODUCTION

Improving energy consumption of portable devices is a research domain that has captured the attention of the research community for a long time. Among portable devices, mobile phone is the most commonly and widely used. "In 2018, total mobile phone sales were almost 1.9 billion units. In 2019, smartphone sales are on pace to continue to grow, at 5% every year" [1]. According to an online report published in 2010, global mobile usage accounts for approximately one quarter of a percent of global mobile CO_2 emissions [2]. These numbers indicate that as the mobile usage is predicted to grow so will the carbon foot print associated with it. The situation could be improved by reducing the energy consumption of mobile phones. One way of reducing the energy consumption of mobile devices is to improve the software quality of mobile apps. This could be achieved by constant refactoring. Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Code smells are an indicator of a problem in software design and quality that requires refactoring" [3]. Refactoring code smells could impact the energy consumption of mobile apps. We chose open source apps in Android for this investigation because Android holds 87.8% of the world's smartphone market share according to a Gartner report [4]. We assume that the types of code smells identified by Fowler et al. [3] occur in Java code independent of platform, although the frequency with which they occur might be distributed differently [5]. Therefore, we picked the set of code smell types analyzed in our study, as well as their refactorings, from Fowler et al.'s list.

In our study, we investigate the energy impact of code smell refactorings on native Android open source apps. Code smell refactoring in Android apps has been discussed with respect to quality characteristics such as performance and maintainability [6], [7] but the energy impact of code smells has not yet been fully explored [8]–[11]. Studies by Pinto and Kamei [12], Yamashita and Moonen [13] indicate that developers do care about code smells (such as 'Long Method', 'Feature Envy') and conduct refactoring. The research presented in this paper extends previous studies by investigating the energy impact of refactoring five code smell types (first individually per type and then in permutation) of native Android open source apps. We also studied the effect of code smell refactorings on execution time. This study is intended to add to the findings of Verdecchia et al. [14] by exploring the energy impact of code smell refactoring on three native Android apps. We expect that our findings will help developers improve their Android apps not only with regards to maintainability but also in terms of energy efficiency.

II. RELATED WORK

Several empirical studies have investigated the energy impact of code smells in the mobile app domain. Sahin et al. [15] applied six code refactorings from Eclipse IDE refactoring tool to the source code and evaluated their energy consumption. Morales et al. [16] analyzed the energy impact of eight types of anti-patterns in 20 open-source Android apps. Rodriguez et al. [9] evaluated the trade-off between OO design and battery consumption of mobile apps. Verdecchia et al. [14] discussed the energy impact of five code smell refactorings on three ORM-based Java web applications. Their results indicate that in one out of three applications code smell refactoring significantly impacted energy consumption. Reimann et al. [17] published a catalogue of 30 Android code smells different from Fowler et al.'s [3] list of code smells. Castillo et al. [10] analyzed the energy impact of 'God Class' refactoring on two Java applications. Their results indicated that God class refactoring has a negative impact on energy consumption. Tufano et al. [18] and Delchey et al. [19] discussed when and how code smells appear in source code during development. Carette et al. [20] proposed an automated approach called HOT-PEPPER, which enables developers to detect and correct three Android-specific code smells. Di Nucci et al. [21] introduced a new software based tool PETrA, for measuring the energy consumption of Android apps.

In this paper, we investigate the energy impact of code smell refactoring for five common Java code smells in native Android apps because Android has the biggest share in the smartphone market, i.e., 87.8% [4] and 44% mobile app users preferred native mobile apps over desktop and web applications [22]. The work closest to this paper is the work published by Verdecchia et al. [14] in which they measured the energy consumption of five code smell refactorings on web applications. In related studies analyzing code smells in Android apps, the focus was on Android specific code smells

This work is supported by the institutional research grant IUT20- 55 of the Estonian Research Council and the Estonian Center of Excellence in ICT research (EXCITE).

978-1-7281-3421-5/19/$31.00 ©2019 IEEE
DOI 10.1109/SEAA.2019.00021

related to specific mobile resources, and their correlation with performance, maintainability, and sustainability.

III. Research Method

Our research questions are the following:

RQ1: Is there a correlation between code refactoring and energy consumption of Android apps?

RQ2: Is there a correlation between code refactoring and execution time of Android apps?

A. Code Smells

In our study we focus on commonly occurring code smells in Java applications and investigate their energy impact in Android apps. The selected smells are 'Long Method', 'Feature Envy', 'Type Checking', 'Duplicated Code', and 'God Class'. These smells have previously been investigated for their energy impact (cf. Section II) but the results from those studies are limited by the type and number of apps on which these code smells were analyzed.

The 'Long Method' code smell is one of the most commonly occurring smells in Android apps. As defined by Fowler [3], this smell points to a method which has become too long over time and could affect the maintainability of the app. Several refactoring techniques are defined by Fowler to get rid of this smell. In our study, we apply the 'Extract Method' refactoring for this smell. As a result of this refactoring, the 'Long method' is divided into smaller methods which are called inside the original method. The 'Feature Envy' code smell occurs when one class envies the features of another class, i.e., a method in one class sends many 'get method' calls to another class, which indicates that this method could be placed inside the other class whose features it envies [3]. We apply the 'Move method' refactoring for this code smell. The 'Type Checking' code smell occurs when an attribute in a class representing the state is checked for different values. This state attribute is referred to as a type field. The term 'Type Checking' was used by the creator of JDeodorant for this smell [3], [23]. In our study, we use the refactoring 'replace type code with state/strategy'. This refactoring results in new methods and subclasses. The 'Duplicated Code' smell occurs when the same code is used in many places inside an app. This could be intentional or it could be due to the copying of the code or it could be the result of another refactoring. It could be refactored by combining the different code clone structures into one [3]. The 'God Class' code smell occurs when over time many functionalities are added to the same class in a project, making it responsible for a large percentage of the app's architecture. This smell is removed by dividing the large class into smaller classes, each having a unique functionality [3].

B. Code Smell Detection tool and Refactoring

Based on previous studies [24]–[27] that compared different code smell detection and refactoring tools, we chose to use JDeodorant for code smell detection. JDeodorant follows all the refactoring activities stated by Mens et al. [28]. According to the studies mentioned above JDeodorant is very effective in detecting the selected code smells.

C. Selected Apps

We selected from F-Droid open source native Android apps that were more than two years old, had more than 10K downloads, and at least a rating of '4' in the Google play store. The selected apps were picked from the categories 'Tools' and 'Puzzles'. The detection of code smells and the generation of refactoring candidates was done using JDeodorant with Eclipse IDE. Table I summarizes the characteristics of each of the selected apps.

TABLE I. Characteristics of selected Android apps

	Calculator[1]	**Todo-List[2]**	**Openflood[3]**
Age of the project (years)	6.8	3	3.1
Source line of code	7758	6145	1236
Downloads (Google play store)	1,000k+	10k+	10k+
Ratings (Mean of user ratings)	4.5	4	4.6

D. Testing Tool

We used Espresso as it comes built-in with Android Studio and, according to a recent study [29], is very fast and reliable outperforming other tools for testing native Android apps. Since we do not intend to navigate outside the app under test, Espresso is a good choice as it supports white box testing and UI tests can be created easily. The test scripts created in Espresso do not need time delays to function properly. Therefore, the overhead introduced in the execution of the apps when using Espresso is low.

We used our test scripts for energy measurements and also to ensure the correctness of the refactored app code. The test cases[4] included in the test scripts may seem trivial but they were solely defined to ensure that the code containing a smell is actually executed. To validate that the test scripts triggered the execution of code containing a code smell as well as the execution of the corresponding refactored code, execution traces were inspected.

E. Energy Measurement

We used Monsoon power monitor to measure the energy consumption, recording the power measurements at a rate of 5KHz. The time between two readings was 0.0002 seconds. In this experiment, energy consumption corresponds to the total amount of energy used by the mobile device within a period of time. Energy is measured in Joules which is power (watts) times measurement period (seconds). We calculated the energy associated with each reading as E = Power x (0.0002). The total energy consumption corresponds to the sum of the energy associated with each reading. Before starting the experiment, a baseline was recorded to measure the energy consumption of the mobile device in an idle state, which was then subtracted from the actual energy readings during the experiment to filter out the energy used by the app under test. During the experiment, the screen brightness was set to minimum and only essential Android services were run on the phone. The timestamps from the adb logs recording during energy measurement and the CSV files containing energy readings were matched to mark the start and end of an execution run. Each app version was executed 10 times to account for underlying variation in the mobile device.

[1] https://f-droid.org/en/packages/com.xlythe.calculator.material/

[2] https://f-droid.org/packages/org.secuso.privacyfriendlytodolist/

[3] https://f-droid.org/en/packages/com.gunshippenguin.openflood/

[4] https://bitbucket.org/hinaanwar2003/calculator/wiki/Test%20Cases
https://bitbucket.org/hinaanwar2003/todolist/wiki/Test%20Cases
https://bitbucket.org/hinaanwar2003/open_flood_src/wiki/Test%20Cases

F. Test Environment

The test environment consisted of an HP Elite Book, the Monsoon power monitor, and an LG Spirit Y70 Phone having Android 5.0.1 as the operating system with 1GB of RAM, and a 2100mAh battery. The test was controlled from the HP Elite Book using a script that automated the process and runs each app version 10 times. This saved manual effort and ensured that no problems were created during the experiment due to human error. The mobile device was connected via USB cable to the Monsoon power monitor according to the instruction manual of the power monitor [30].

G. Experimental Setup and Execution

In this section, we describe the setup of our experiment to measure the energy consumption of code smell refactorings on Android apps. Since Android studio projects cannot be opened directly inside Eclipse like Java projects, the apps were first modified using an Eclipse plugin in Android studio generating a file structure that helps Eclipse IDE to open the project with Gradle and recognize the Java files. Build path dependencies for projects opened in Eclipse IDE were solved manually by the author. This step was necessary because JDeodorant is provided as a plugin in Eclipse IDE and works only with Java projects. Code smell refactorings were applied in two ways: 1) code smell refactorings per code smell type and 2) code smell refactorings for all code smell types (one type after the other) in various permutations. For single code smell type, we first detected the smells using JDeodorant and then candidate refactorings suggested by JDeodorant were applied. For each suggested refactoring, the test script was executed to ensure that the refactoring could be applied without altering the functionality of the app. If the test script failed for a suggested refactoring, that refactoring was ignored. We applied refactorings for a single code smell type and made new versions of the refactored app. For assessing whether the order of refactoring has an effect, we also made versions in which all code smell types were refactored in various permutations grouped by type. As the total number of all possible permutations of five code smell types was 120, we chose a sample of permutations randomly using Fisher-Yates shuffle [31]. For versions of the app where refactorings of all code smell types were applied in various permutations, we ignored any new candidate refactorings identified as a result of applying refactorings to a previous code smell type. For most code smell types, more refactorings could be applied but we only applied and reported the refactorings for which the test script was successfully executed.

TABLE II. NUMBER OF REFACTORINGS FOR CALCULATOR APP

Refactoring	LM	FE	TC	DC	GC
Single Refactoring	41	1	3	3	6
LM-TC-GC-FE-DC	41	1	2	2	5
FE-LM-TC-DC-GC	41	1	2	2	5
TC-GC-LM-FE-DC	39	1	3	2	5
GC-LM-DC-TC-FE	41	1	2	2	6
FE-DC-TC-GC-LM	40	1	3	3	5
TC-LM-DC-FE-GC	39	1	3	2	5
LM-DC-TC-GC-FE	41	1	2	2	5
LM-TC-DC-FE-GC	41	1	2	2	5
DC-TC-FE-LM-GC	40	1	3	3	5

Tables II, III, and IV show the numbers of applied code smell refactorings per app. The code smell names are abbreviated as LM (Long Method), FE (Feature Envy), TC (Type Checking), DC (Duplicated Code), and GC (God Class). In the first column 'Refactoring', the row 'Single Refactoring' refers to the situation where only code refactorings of one code smell type were applied. The rest of the rows in the first column represent the situation where refactorings of all code smell types were applied one by one in various permutations. For statistical analysis, normality and homogeneity of the energy data were checked before doing the analysis of variance (ANOVA) related to RQ1 and RQ2. For our analyses, α was set to 0.05. We calculated the effect size using Cliff's delta [32].

TABLE III. NUMBER OF REFACTORINGS FOR TODO-LIST APP

Refactoring	LM	FE	TC	DC	GC
Single Refactoring	45	3	3	4	11
LM-TC-GC-FE-DC	45	1	3	2	6
FE-LM-TC-DC-GC	44	3	3	2	6
TC-GC-LM-FE-DC	41	1	3	2	11
GC-LM-DC-TC-FE	43	1	3	2	11
FE-DC-TC-GC-LM	36	3	3	4	11
TC-LM-DC-FE-GC	45	1	3	2	6
LM-DC-TC-GC-FE	45	1	3	2	6
LM-TC-DC-FE-GC	45	1	3	2	6
DC-TC-FE-LM-GC	40	3	3	4	6

TABLE IV. NUMBER OF REFACTORINGS FOR OPENFLOOD APP

Refactoring	LM	FE	TC	DC	GC
Single Refactoring	7	1	0	0	1
LM-GC-FE	7	0	0	0	1
FE-LM-GC	7	1	0	0	1
GC-LM-FE	6	0	0	0	1
FE-GC-LM	6	1	0	0	1
LM-FE-GC	7	1	0	0	1
GC-FE-LM	6	0	0	0	1

IV. RESULTS AND EVALUATION

In this section, we present the results of our investigation regarding the impact of code refactoring on energy consumption and execution time for Android apps.

RQ1: Is there a correlation between code refactoring and energy consumption of Android apps?

An analysis of variance (ANOVA) showed that the effect of code smell refactorings on energy consumption of Android apps was significant in two out of three apps. The null hypothesis related to RQ1 states that the energy consumption between the original version and all possible refactored versions of the app remains the same. Based on the p-values and F-values (cf. Table V), the null hypothesis could be rejected for the 'Calculator' and 'Todo-List' apps (p-values below α=0.05 and large F-values) but not for 'Openflood'.

TABLE V. ANOVA RESULTS FOR ALL APPS

Application	p-Value	F-value
Calculator	0.00005343	7.3503
Todo-List	0.0009859	4.3473
Openflood	0.4351	0.9666

Figures 1 to 3 show the boxplots of Treatment vs. Energy consumption in joules for each of the apps. The treatments are along the x-axis while energy consumption is shown along the y-axis. Treatments refer to the type of code smell refactoring applied (or=Original, LM=Long Method, FE=Feature Envy, TC= Type Checking, GC=God Class, DC=Duplicated Code, ALL=Avg. of all versions where the refactorings of all code smell types were applied in permutations). From figures 1 to 3 we see that for permutations of code smell refactorings, for

all apps, there was no significant difference in energy consumption. For individual code smell refactorings, two out of three apps had a significant difference in energy consumption, but the direction of the effect is not uniform across treatments. Only TC, DC, FE, and LM consistently saving energy in both apps, however, the strength of the effect is not uniform (more details in the discussion section).

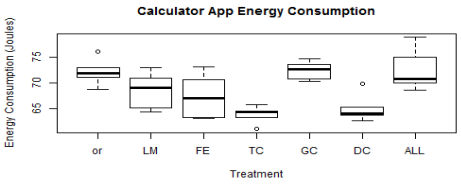

Figure 1: Energy consumption in joules for Calculator app per treatment

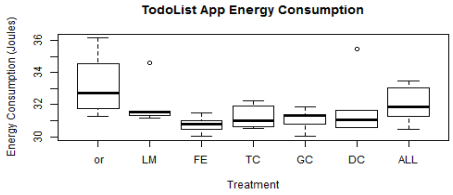

Figure 2: Energy consumption in joules for Todo-List app per treatment

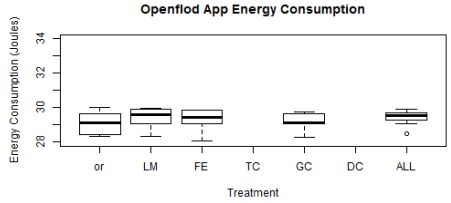

Figure 3: Energy consumption in joules for 'Openflood' app per treatment

RQ2: Is there a correlation between code refactoring and execution time of Android apps?

Based on our measurement we could not find a significant impact of code refactoring on execution time for the selected Android apps. The p-values were 0.2584, 0.113, and 0.384 for the 'Calculator', 'Todo-List', and 'Openflood' apps respectively, which was greater than the alpha value of 0.05. Therefore, the null hypothesis stating that the execution time between the original version and all possible refactored versions of the app remains the same could not be rejected.

V. Discussion

In the following, we take a closer look at how different configurations of code smell refactorings affected the energy consumption of each of the analyzed Android apps. Table VI gives details about the median, standard deviation, percentage change, and effect size for each app after each treatment (we used the same abbreviation as in Figures 1 to 3). We see that for the 'Calculator' app maximum reduction in energy consumption was recorded in app versions where 'Duplicated code' (10.8%) and 'Type checking' (10.5%) code smell refactorings were applied. Verdecchia et al. [14] reported maximum energy reduction in versions where 'Long Method' and 'Feature Envy' code smell refactorings were applied. In our experiment, the number of applied refactoring of both 'Type Checking' and 'Duplicated Code' code smells were smaller as compared to 'Long Method' code smell refactorings but the recorded effect size for energy reduction was larger. For app versions where 'Long Method' code smell refactorings were applied our results were similar to Verdecchia et al. [14].

TABLE VI. Overview of Energy Consmuption results

Application (joules)		Treatments						
		Or	LM	FE	TC	GC	DC	ALL
Calculator	Median	71.9	69.1	67.1	64.3	72.6	64.1	71.6
	SD	2.6	3.6	4.4	1.7	1.8	2.8	3.4
	%	-	-3.8	-6.6	-10.5	+1.0	-10.8	-1.3
	ES	-	L	L	L	N	L	S
Todo-List	Median	32.7	31.5	30.8	31.0	31.3	31.0	31.8
	SD	2.3	1.4	0.5	0.7	0.7	2.0	1.0
	%	-	-3.6	-5.9	-5.2	-4.4	-5.1	-2.6
	ES	-	M	L	L	L	L	M
Openflood	Median	29.1	29.5	29.4	N/A	29.1	N/A	29.5
	SD	0.6	0.5	0.5	-	0.4	-	0.4
	%	-	+1.5	+1.1	N/A	-0.02	N/A	+1.3
	ES	-	S	S	N/A	N	N/A	S

(SD=standard deviation, %= percentage change, ES=effect size, L=large, M= medium, S=small, N=negligible)

Insight 1: Impact of refactoring only a single type of code smell on energy consumption of selected apps was not consistent. However, in two out of three selected apps, where the effect size was medium or large, the energy consumption decreased due to refactoring.

TABLE VII. ENGERGY CONSUMPTION RESULTS WHEN A PERMUTATION OF CODE SMELL REFACTORINGS WAS APPLIED.

Refactoring	Calculator				Todo-List			
	Med	SD	%	ES	Med	SD	%	ES
Original (baseline)	71.9	2.6	-	-	32.7	2.0	-	-
LM-TC-GC-FE-DC	74.8	6.3	+4.0	N	31.3	0.9	-4.3	L
FE-LM-TC-DC-GC	66.8	2.8	-7.0	L	31.4	0.8	-3.9	M
TC-GC-LM-FE-DC	66.8	6.9	-7.1	S	31.3	0.9	-4.1	L
GC-LM-DC-TC-FE	75.8	6.9	+5.4	S	32.3	2.3	-1.1	N
FE-DC-TC-GC-LM	76.6	6.5	+6.9	L	30.4	0.5	-7.0	L
TC-LM-DC-FE-GC	70.9	2.5	-1.3	S	33.8	0.7	+3.4	N
LM-DC-TC-GC-FE	69.4	4.8	-3.4	S	32.2	1.5	-1.3	N
LM-TC-DC-FE-GC	77.1	4.8	+7.2	L	30.9	0.5	-5.3	L
DC-TC-FE-LM-GC	65.1	9.6	-9.3	L	31.9	1.2	-2.4	S

(Med=median, SD= standard deviation, %= percentage change, ES=effect size, L=large, M= medium, S=small, N=negligible)

Table VII shows the median, standard deviation, percentage change, and effect size when the refactorings of all code smell types were applied in permutations. We only analysed the 'Calculator' and 'Todo-List' app because for 'Openflood' app the change in energy consumption was not significant. In the 'Calculator' and 'Todo-List' apps. In 'Calculator' app the permutation 'LM-TC-DC-FE-GC' resulted in a maximum increase of energy consumption up to 7.25%, while the permutation 'DC-TC-FE-LM-GC' resulted in a maximum decrease of energy consumption up to 9.3%. In the 'Todo-List' app no permutation resulted in a significant increase in energy consumption, However, the maximum decrease of energy consumption was in permutation 'FE-DC-TC-GC-LM' (up to 7%). If not used carefully the refactorings for different code smells could cancel out each other's positive effects. To find a clear pattern between a permutation of code smell refactoring types and energy consumption, we need experiments with more permutations on a bigger corpus of Android apps. The results related to RQ2 are contradictory to the results reported by Verdecchia et al. [14]. They reported that observed energy reduction was due to performance-related improvements. Therefore, more experimental

evidence is required to confirm the assumption that a trade-off between execution time and energy consumption exists.

Insight 2: The energy impact of overall permutations of code smell refactorings in the selected Android apps was small. However, specific permutations of code smell refactorings should be used with caution as their energy impact might vary strongly depending on the selected Android app.

Insight 3: Significant reduction in energy consumption of Android apps does not necessarily correlate with a significant reduction or increase of execution time.

VI. Threats to Validity

To ensure that the apps chosen from F-Droid are representative we ensured that minimum conditions were fulfilled (cf. Section III.C). The accuracy of the code smell detection and refactoring tool might affect the accuracy of our results. We selected JDeodorant because it complies with the activities Mens et al. [28] recommended for a good smell detection and refactoring tool. However, using a different tool might produce different results. Power measurements using the Monsoon power monitor were made for the apps plus Android OS related activities. We tried to minimize this threat by using adb logs and matching the timestamps with the energy trace to filter out any unwanted readings. Changing the version of the apps might result in the failure of our test scripts. Activities such as writing test scenarios and checking each candidate refactoring before applying are resource intensive. Therefore, we limited this experiment to three apps. This limits the generalizability of our results.

VII. Conclusion and Future Work

The research presented in this paper extends previous studies by investigating the energy impact of refactoring five code smell types (first individually per type and then in permutations) of native Android open source apps. We also studied the impact of using the code smell refactorings on the execution time of native Android open source apps. Our results indicate that the maximum energy reductions recorded are 10.8% and 10.5% for refactoring code smell 'Duplicated code' and 'Type Checking' respectively. Specific permutations of code smell refactorings should be used with caution, as their energy consumption impact might differ strongly between the selected Android apps. We observe neither significant increase nor decrease of the execution time in selected Android apps.

References

[1] Gartner, "Gartner Says Worldwide End-User Device Spending Set to Increase 7 Percent in 2018; Global Device Shipments Are Forecast to Return to Growth," Gartner, Press Releases, 2018. .

[2] M. Berners-Lee, "What's the carbon footprint of using a mobile phone? The Guardian," 2010.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code. Reading, MA: Addison-Wesley, 1999.

[4] Gartner, "Gartner Says Huawei Secured No. 2 Worldwide Smartphone Vendor Spot, Surpassing Apple in Second Quarter 2018," Gartner, Press Release, 2018.

[5] J. P. dos Reis, F. Brito e Abreu, and G. de F. Carneiro, "Code Smells Incidence: Does It Depend on the Application Domain?," in QUATIC, 2016, pp. 172–177.

[6] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," JSS, vol. 86, no. 10, pp. 2639–2653, Oct. 2013.

[7] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in 3rd Int. symp.- ESEM, 2009, pp. 390–400.

[8] M. Gottschalk, J. Jelschen, and A. Winter, "Saving Energy on Mobile Devices by Refactoring," in EnviroInfo, 2014, pp. 437–444.

[9] A. Rodriguez, M. Longo, and A. Zunino, "Using bad smell-driven code refactorings in mobile applications to reduce battery usage," in ASSE, 2015, p. p.56-68.

[10] R. Pérez-Castillo and M. Piattini, "Analyzing the Harmful Effect of God Class Refactoring on Power Consumption," IEEE Softw., vol. 31, no. 3, pp. 48–54, May 2014.

[11] A. Vetrò, L. Ardito, G. Procaccianti, and M. Morisio, "Definition, implementation and validation of energy code smells: an exploratory study on an embedded system," in ENERGY 2013, 2013, pp. 34–39.

[12] G. H. Pinto and F. Kamei, "What programmers say about refactoring tools?," in ACM - WRT '13, 2013, pp. 33–36.

[13] A. Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," in WCRE, 2013, pp. 242–251.

[14] R. Verdecchia, R. Aparicio Saez, G. Procaccianti, and P. Lago, "Empirical Evaluation of the Energy Impact of Refactoring Code Smells," in 5th ICT4S, 2018, pp. 365–383.

[15] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?," 8th ACM/IEEE Int. Symp. - ESEM '14, pp. 1–10, 2014.

[16] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "EARMO: An Energy-Aware Refactoring Approach for Mobile Apps," IEEE Trans. Softw. Eng., vol. 44, no. 12, pp. 1176–1206, 2018.

[17] J. Reimann and M. Brylski, "A Tool-Supported Quality Smell Catalogue For Android Developers," in workshop MMSM, 2014, 2014, pp. 14–15.

[18] M. Tufano et al., "When and why your code starts to smell bad," in ICSE, 2015, vol. 1, pp. 403–414.

[19] M. Delchev and M. F. Harun, "Investigation of Code Smells in Different Software Domains," in SWC Seminar, 2015.

[20] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of Android smells," in IEEE 24th SANER, 2017, pp. 115–126.

[21] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "PETrA: A Software-Based Tool for Estimating the Energy Profile of Android Applications," in IEEE/ACM 39th ICSE-C, 2017, pp. 3–6.

[22] N. Swanner, "Users Spend More Money in Apps as Mobile Web Use Declines," online survey, 2018. .

[23] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Type-Checking Bad Smells," in 12th CSMR, 2008, pp. 329–331.

[24] D. Verloop, "Code Smells in the Mobile Applications Domain," Delft University of Technology, 2013.

[25] F. A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni, "On experimenting refactoring tools to remove code smells," in Workshop XP '15, 2015.

[26] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten Years of JDeodorant: Lessons Learned from the Hunt for Smells," in IEEE 25th SANER, 2018.

[27] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," JSERD, vol. 5, no. 1, p. 7, Dec. 2017.

[28] T. Mens and T. Tourwé, "A Survey of Software Refactoring," IEEE Trans. Softw. Eng., 2004.

[29] T. Lämsä, "Comparison of GUI testing tools for Android applications," University of Oulu, 2017.

[30] "Mobile Device Power Monitor Manual," 17AD. [Online]. Available: www.msoon.com. [Accessed: 04-Jun-2018].

[31] R. A. Fisher and F. Yates, Statistical tables for biological, agricultural and medical research, 6th ed. Edinburgh: Oliver and Boyd, 1963.

[32] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions.," Psychol. Bull., vol. 114, pp. 494–509, 1993.