On the Impact of Code Smells on the Energy Consumption of Mobile Applications

Fabio Palomba[1], Dario Di Nucci[1], Annibale Panichella[2], Andy Zaidman[2], Andrea De Lucia[1]
[1]University of Salerno, Italy — [2]Delft University of Technology, The Netherlands
fpalomba@unisa.it, ddinucci@unisa.it, a.panichella@tudelft.nl, a.e.zaidman@tudelft.nl, adelucia@unisa.it

Abstract—**The demand of green software design is steadily higher especially in the context of mobile devices, where the computation is often limited by battery life. Previous studies found how wrong programming solutions have a strong impact on the energy consumption. However, the research community has not considered yet the influence that code smells, widely considered symptoms of poor design or implementation choices, have on the energy efficiency of mobile apps. To fill this gap, we conducted a large-scale empirical study on the influence of 9 Android-specific code smells on the energy consumption of 60 Android apps. In particular, we focus our attention on the design flaws that have been theoretically supposed to be related to non-functional attributes of source code, such as performance and energy consumption. The results of the study highlight that methods affected by code smells consume up to 385% more energy than smell-free methods. A fine-grained analysis reveals that the higher energy consumption is mainly due to four specific smell types, *i.e., Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*. Moreover, we found that refactoring these code smells reduces energy consumption by up to 900%.**

Keywords-**Code Smells; Refactoring; Energy Consumption; Mobile Apps**

I. Introduction

Energy efficiency is becoming a major issue in modern software engineering, as applications performing their activities need to preserve battery life. Although the problem is mainly concerned with hardware efficiency, in the recent past researchers successfully demonstrated how even software may be the root of energy leaks [1]. The problem is even more evident in the context of mobile applications (*a.k.a.*, "apps"), where billions of customers rely on smartphones every day for social and emergency connectivity [2].

Green mining is the branch of software engineering responsible for the identification of factors causing energy leaks, as well as for the definition of practical solutions to deal with them. In this context, recent research has ranged from the definition of approaches to measure the power profile of mobile apps [1], [3] to the analysis of the impact of programming solutions on the energy consumption [4], [5], [6], [7].

Recent advances in the latter category of studies revealed that wrong choices made by programmers during the development tend to negatively influence the energy usage of mobile apps. For instance, Sahin *et al.* [4] highlighted the existence of design patterns that negatively impact the power efficiency, as well as the role of code obfuscation on the phenomenon [8]. Linares Vasquez *et al.* [5] studied the API usage of Android apps and their relationship with energetic characteristics of apps. More recently, Hasan *et al.* [7] investigated the impact of the Java Collections, finding that wrong type of data structure can decrease the energy efficiency by up to 300%.

Although several important research steps have been made and despite the ever increasing number of empirical studies aimed at understanding the reasons behind the presence of energy leaks in the source code, we observe that the research community has not considered yet the potential impact on energy consumption of so-called *bad code smells* [9] (also named *code smells* or simply *smells*) defined by Reimann *et al.* [10] for Android mobile applications.

In this paper we fill this gap by studying (i) to what extent code smells, affecting source code methods of mobile applications, influence energy efficiency and (ii) whether refactoring operations applied to remove them are able to reverse the negative effects of code smells on energy consumption. In particular, our investigation focuses on 9 method-level code smells specifically defined for mobile applications by Reimann *et al.* [10] in the context of 60 Android apps belonging to the dataset provided by Choudhary *et al.* [11]. While Reimann *et al.* theoretically supposed the existence of a relationship between these code smells and non-functional attributes of source code (*e.g.,* energy consumption), to the best of our knowledge this is the first study aimed at practically investigating the actual impact of such code smells on energy consumption and quantifying the extent to which refactoring code smells is beneficial for improving energy efficiency.

Due to the absence of publicly available tools to address these research questions, we developed and evaluated (i) a novel *Android-specific* code smell detector coined as `aDoctor` and (ii) a software-based tool named `PETrA` (**P**ower **E**stimation **T**ool fo**r** **A**ndroid) to estimate the energy profile of mobile applications. The code smell detector has been evaluated on 18 apps, showing a precision of 93% and a recall of 97%, while `PETrA` has been evaluated in the context of 23 apps showing that in 91% of the cases our tool provides an estimation error within 5% of the actual values measured with a hardware-based tool [5]. It is worth noting that we made both the tools and experimental data concerning their evaluations publicly available [12].

Results of the study highlight that methods affected by code smells consume up to 385% more energy than methods not affected by any smell. A *fine-grained* analysis reveals the existence of four specific *energy-smells*, namely *Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*. Finally, we also shed light on the usefulness of refactoring as a way for improving energy efficiency by code smell removal. Specifically, we found that it is possible to improve the energy efficiency of source code methods by up to 900% through the refactoring of code smells.

Summarizing, the contributions of this paper are:

1) A large scale empirical study involving 60 Android apps aimed at assessing the extent to which 9 method-level code smells impact energy efficiency and whether refactoring operations are able to fix energy leaks.
2) A novel *Android-specific* code smell detector and its evaluation, coined as `aDoctor`.
3) A software-based power estimation tool for Android applications, named as `PETrA`.
4) A comprehensive replication package [12], including all the raw data, tools (`aDoctor` and `PETrA`), and scripts used to answer our research questions.

Structure of the paper. Section II reports the design of the empirical study, while Section III describes the results achieved. Section IV discusses the threats that could affect the validity of the results. Section V summarizes the related literature in the context of green mining and code smells. Finally, Section VI concludes the paper.

II. Empirical Study Definition and Design

The *goal* of the study is to analyze the source code of mobile apps with the *purpose* of investigating whether (i) the presence of code smells influences energy consumption, and (ii) the removal of such design flaws through refactoring actually reduces the energy consumption of mobile applications. More specifically, the study aims at addressing the following three research questions:

- **RQ$_1$:** *What is the impact of code smells on the energy consumption of mobile apps?*
- **RQ$_2$:** *Which code smells have higher negative impact on the energy consumption of mobile apps?*
- **RQ$_3$:** *Does the refactoring of code smells positively impact the energy consumption of mobile apps?*

A. Context Selection

The *context* of the study consists of 60 open source Mobile Android apps publicly available in the F-Droid repository [13]. Specifically, we selected all the apps from the benchmark dataset provided by Choudhary *et al.* [11], which collects a subset of apps used in previous studies [14], [15], [16], [17] having different size and scope and that are still maintained by their own developers. The complete list of the apps used in this study is available in our online appendix [12].

Table I reports the set of code smells investigated in the study, together with a brief explanation and the corresponding refactoring operations. In particular, we analyzed the behavior of 9 *Android-specific* code smells extracted from the catalogue defined by Reimann *et al.* [10]. Such a catalogue reports a set of poor design/implementation choices applied by Android developers that are believed to impact non-functional attributes of mobile apps, such as software quality, user experience, performance, and energy consumption. However, it is important to point out that the actual impact of the defined smells on non-functional attributes has only been conjectured by the authors of the catalogue, while no empirically evaluation has been directly conducted to verify and measure such an impact.

Among the 30 types of Android-specific design flaws available in the catalogue, we selected only 9 code smells for three main reasons. First of all, we selected the design flaws that directly affect the source code, while the catalogue also includes problems related to poor user interface design choices, *e.g., Nested Layout*. Secondly, we included the code smells supposed to be directly connected with the energy consumption of the app rather than the ones related to violations of other non-functional aspects, such as data security and privacy. For instance, we have not considered the *Public Data* code smell, which appears when private data is stored in a location publicly accessible by other applications [10]: even if this problem affects the source code of an app, it does not seem directly connected with its energy consumption. Finally, we focused on method-level code smells only, since for them we can isolate the energy consumption for each single method execution. On the other hand, the analysis of class-level code smells (*e.g.,* most of the Fowler's smells [9]) are particularly challenging because objects (instances of a class) can remain in memory during the execution of the app[1] and, hence, isolating their behavior is more difficult. Therefore, while the analysis of other class-level code smells could be worthwhile, it requires specialized tools and methodologies able to adequately deal with them. It is, therefore, part of our future research agenda.

B. Data Extraction and Analysis

To answer our research questions, we first needed to identify the instances of the 9 code smells considered in the study. A manual detection would have been prohibitively expensive because of the number of both code smell types and mobile apps involved in the study. For this reason, we developed a novel code smell detector, coined `aDoctor`, which is publicly available in our online appendix [12]. `aDoctor` exploits the structural properties extractable from the source code for detecting instances of all the considered smells. It is important to notice that the design flaws defined for Android apps are often easier to identify when compared to the traditional smells described in [9]. As an example, the *Inefficient Data Structure* smell is based on the fact that the mapping from an integer to an object is slow: for this reason, the smell is strongly related to the use of an `HashMap<Integer, Object>` as data structure and, therefore, easily detectable automatically. To evaluate the performance of our tool in detecting smells, we have manually validated a subset of the smell instances

[1]https://developer.android.com/reference/android/app/Activity.html

TABLE I: The Code Smells considered in our Study

Abbreviation	Name	Description	Refactoring
DTWC	Data Transmission Without Compression	A method transmitting a file over a network infrastructure without compressing it	Add Data Compression
DWL	Durable Wakelock	A method acquiring a wakelock and not releasing it	Aquire WakeLock with Timeout
IDS	Inefficient Data Structure	A method using an Hashmap<Integer, Object>	Use Efficient Data Structure
ISQLQ	Inefficient SQL Query	A method using a SQL query over a JDBC connection to a remote server	Use JSON query
IDFAP	Inefficient Data Format And Parser	A method using a TreeParser	Use Efficient Data Parser and Format
IGS	Internal Getter/Setter	Internal fields are accessed via getters and setters	Direct Field Access
LT	Leaking Thread	A method using a thread that will never be stopped	Introduce Run Check Variable
MIM	Member-Ignoring Method	Non-static methods that don't access any property	Introduce Static Method
SL	Slow Loop	A slow version of a for-loop is used	Enhanced For-Loop

detected on 18 apps, for a total of 1,927 instances manually inspected. This manual validation has been performed by two authors independently, and cases of disagreement were discussed. Such a (stratified) sample is deemed to be statistically significant for a 95% confidence level and $\pm 10\%$ confidence interval [18]. The results of the manual validation indicated a mean precision and recall of 93% and 97% respectively. The results achieved in this analysis as well as the set of validated code smells are available in our replication package [12]. This step required approximately 160 man-hours.

The second step consisted of deriving the energy consumption profile of Android apps. Despite the fact that a number of tools have been proposed to perform such measurements, these are not available [3] or require hardware equipment and a strong experience in the set up of the test bed [19]. For this reason, in our study we relied on `PETrA` (**P**ower **E**stimation **T**ool fo**r** **A**ndroid), which is publicly available [20]. `PETrA` is a software-based approach able to estimate the energy consumed by each executed method. More in detail, the tool instruments the methods of the app under analysis, and runs a set of test cases received as input. During test execution, `PETrA` records the time needed to complete the execution for each exercised method along with the estimation of the joules consumed by the app during the time between the entry and the exit of the monitored method. Our tool is built on top of `Project Volta` Android tools, such as `dmtracedump` [21], `Systrace` [22], and `Batterystats` [23]. For this reason, `PETrA` is able to provide power estimations similar to those obtained using the power estimation model provided by Android. The differences in the estimations can be explained by the fact that currently `PETrA` does not consider the energy consumed by (i) mobile radio, (ii) sensors (*i.e.,* motion, environmental, and position sensors), and (iii) changes in the WiFi signal. However, we empirically evaluated the performances of our tool, comparing its estimations with a publicly available oracle [5] reporting the actual consumptions provided by a hardware-based tool, *i.e.,* `Monsoon` [24]. To this aim, we used the same phone (*i.e.,* a `LG Nexus 4`) and settings used by Linares-Vasquez *et al.* [5]. The results show that in 91% of the cases the estimation error of `PETrA` is within 5% of the actual values, while in the remaining 9% of the cases the estimation error is within 50% of the actual values. The interested reader can find more information about the validation of the tool in [12]. In the context of this work, we followed a well-defined process already used in previous work[3], [5], [19], [25] to extract the energy profile of the 60 Mobile apps:

- The phone used in the experiment is a factory resetted `LG Nexus 4` having Android 5.1.1 Lollipop as operating system, and equipped with 1.5 GHz quad-core Snapdragon S4 Pro processor with 2 GB of RAM, and having a 2100 mAh, 3.8V battery. The choice of the phone is guided by previous research in the field [3], [5], [19], [25] but also because this particular hardware allow to be connected via a data cable, namely a cable where the USB charging can be disabled [26]. Therefore, no energy is transferred over the cable, allowing more stable measurements. Before starting the experiment, we completely re-set the phone in order to avoid bias in the power measurements. Moreover, to limit noise (i) we disabled all the unnecessary apps and processes running on the phone to avoid race conditions, (ii) we did not insert any sim card to avoid asynchronous events, such as incoming messages or calls, and (iii) to avoid energy measurements by sensors and WiFi signal changes we held the phone steady. This setup, available in our online appendix [12], was needed in order to allow `PETrA` to work in an adequate test environment.
- As for the test cases to give as input to `PETrA`, we automatically generated them by using *Monkey* [27], a tool belonging to the Android SDK that produces pseudo-random streams of user events (*i.e.,* clicks, touches, gestures). The choice of Monkey has been guided by recent results [11], [28] showing that this tool achieves the better compromise between coverage and effort needed for the set up. In the experiment, we used the configuration of Monkey suggested by Choudhary *et al.* [11]. Since Monkey may produce events which have the effect of testing external parts of the app under test (*e.g.,* a click may open the status bar), we properly configured Monkey to focus only on the app under analysis. Moreover, in order to not improperly enable/disable smartphone functionality (*e.g.,* WiFi, bluetooth, GPS), we hid the status bar [29].
- The measurements provided by `PETrA` were repeated 10 times in order to have a more reliable estimation of the energy profiles. Each run costs around five minutes since, as reported by Choudhary *et al.* [11], this is the time needed by Monkey to achieve code coverage convergence. The results achieved after 10 runs (*i.e.,* the joules consumed by the methods in each run) were aggregated using the mean operator. Thus, the final output consisted of a unique value representing the average energy consumed by the methods exercised during the test

execution. Overall, the data extraction process (*i.e.,* the smell detection and the extraction of the energy profiles of 60 apps) took eight weeks.

Once we extracted the code smell instances affecting the apps with `aDoctor` and the energy profiles using `PETrA`, we answered **RQ**$_1$ by comparing the energy consumed by methods with and without code smells. We used the boxplots to provide a graphical comparison between the distributions of the energy consumed by the two groups of methods, i.e., with and without Android-specific smells. To test the statistical significance of the differences (if any) between such distributions we used the Mann-Whitney test [30]. The results are intended as statistically significant at $\alpha = 0.05$. We estimated the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure [31] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [31]. Note that the energy consumption of methods might be influenced by other factors such as size, complexity, or level of coverage achieved by the test cases. However, we carefully excluded this hypothesis by performing an additional analysis on the effect of these source code characteristics on the energy consumed by each method (see Section IV for further details).

In the context of **RQ**$_1$ we do not discriminate the specific type of smell affecting a method (*i.e.,* a method is considered smelly if it contains any type of code smell). A fine grained analysis of the impact of the different smell types on energy consumption is investigated in **RQ**$_2$.

To this aim, we split the set of smelly methods identified in the context of the previous research question in 9 subsets, one subset for each code smell type. Then, we compared the distributions of energy consumed by such subsets in order to find statistically significant differences (if any). Note that since our goal is to evaluate the impact of single code smell types, the subsets built in this stage do not contain methods affected by more than one code smell type.

Finally, the goal of **RQ**$_3$ was to evaluate whether refactoring operations (originally targeted to remove the smells) are actually useful for reducing the energy consumption of the smelly methods. In this stage, we focused our attention only on the code smells identified as strongly related to the energy consumption in the previous research question. To perform this analysis, we manually analyzed the source code of the methods involved in the design problem and performed refactoring operations according to the guidelines provided by Reimann *et al.* [10]. In particular, the set of methods to analyze and refactor (1,344) have been distributed among two of the authors, who were responsible for refactoring 672 instances each. Each of the involved authors independently refactored the methods assigned to them, by relying on (i) the definitions of refactoring and (ii) the examples provided by Reimann *et al.* [10]. This step required approximatively 120 man-hours. The output of this phase consisted of the source code where code smells was removed. Then, the two authors involved in the task discussed their activities in order to double-check the consistency of their individual refactoring applications. It is worth remarking that these types of smells can be removed by applying simple program transformations that do not impact the external behavior of the source code. For instance, the previously mentioned *Inefficient Data Structure* can be refactored by replacing the `HashMap<Integer, Object>` with a `SparseArray<Bitmap>` [10]. To be confident that the process did not alter the behavior of the app under analysis, we also re-executed the test cases generated by Monkey (and used to answer our previous RQs) at the end of each refactoring. Once refactored the source code, we repeated the energy measurements using the same design as for **RQ**$_1$. Then, we compared the energy consumption obtained using the smelly version of the app with the energy consumption obtained by its corresponding refactored version. Also in this case, we used boxplots and statistical tests for significance (Mann-Whitney test) and effect size (Cliff's Delta).

To have a practical view of the results achieved in the study, we also computed the percentage of the battery charge consumed by methods affected and not by code smells. In particular, given the characteristics of the phone used in the experiments (*i.e.,* 2,100 mAh and 3.8V battery), the percentage of battery discharge can be computed using the following formula [32]:

$$f(n) = \left(\frac{V \cdot S \cdot I}{V} \cdot \frac{1}{I \cdot S}\right)\% \qquad (1)$$

where V is the voltage, I represents the current intensity, and S the time. Our measures are performed by considering the joules consumed by a method. Formally, a joule represents the work required to move an electric charge of one coulomb through an electrical potential difference of one volt ($V \cdot C$). Since a coulomb is the charge transported by a constant current of one ampere in one second ($I \cdot S$), the numerator of the first part of Equation 1 (*i.e.,* $V \cdot S \cdot I$) is exactly the amount of joules consumed by a method. Hence, a method consuming 0.01 J will consume $3 \cdot 10^{-5}\%$ of the total battery charge because Equation 1 is instantiated as follow:

$$\left(\frac{0.01}{3.8} \cdot \frac{1,000}{2,100 \cdot 3,600}\right)\% = 3 \cdot 10^{-5}\% \qquad (2)$$

where the value of 1,000 at the numerator is because the charge is expressed in mAh and not in Ah, and 3,600 is used to convert hours to seconds.

III. Analysis of the Results

In this section we describe the results achieved to answer our three research questions.

A. What is the impact of code smells on the energy consumption of mobile apps?

In the entire dataset, `aDoctor` detected 6,196 code smell instances. The most frequent ones are: *Member Ignoring Method* (3,104 instances), *Slow Loop* (1,288 instances), and *Data Transmission Without Compression* (564 instances)

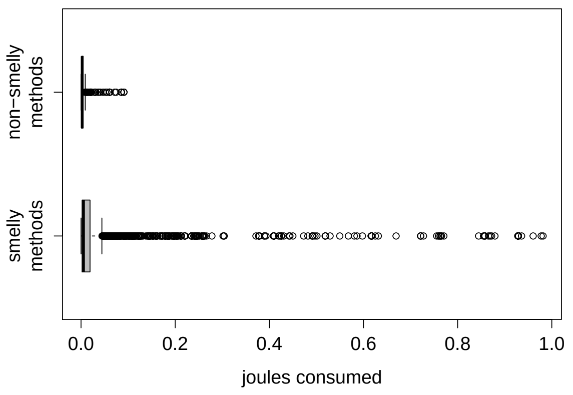

Fig. 1: Energy Consumption of methods affected and not by code smells

smells. The detailed results about the distribution of the studied smells over the 60 apps are reported in our online appendix [12]. Figure 1 depicts the boxplot reporting the distribution of the energy consumption of smelly and non-smelly methods. Note that for sake of space limitation, we aggregate the results of all the 60 apps. As we can see, the difference in the energy consumed by the methods in the two sets is quite large and, therefore, it seems there exists a relationship between the presence of smells and the energy consumption of methods. Indeed, the median energy consumption of methods affected by code smells ($0.0053J$) is almost 3 times higher with respect to the median energy consumption of the other methods ($0.002J$). More importantly, the mean energy consumption of smelly methods is 25 times higher than the one of smell-free methods ($0.05J$ vs $0.007J$). In general, from the boxplot we observe that the methods not affected by any design flaw have an extremely low energy variability, unlike those affected by smells. A representative example can be found in the `a2dpvolume` project [33], an app able to automatically adjusts the media volume when the phone is connected to Bluetooth devices. The method `onCreate` of the `a2dp.Vol.AppChooser` class creates a thread without closing it and, therefore, it is affected by a *Leaking Thread* smell. In ten runs, `PETrA` estimates its energy consumption around 0.77 joules on average, namely 385% more than the median consumption of the non-smelly methods (110% more when considering the mean). The results observed above are also confirmed by the Mann-Whitney and Cliff tests, which reveal that the differences between the two sets of methods are statistically significant (p-value<0.001) with a large effect size (d=0.61).

While the analysis carried out until now clearly highlighted a trend in terms of energy consumption for smelly and non-smelly methods, it is important to note that a smelly method may be affected by multiple smell types. For this reason, we performed an additional analysis to verify how the energy consumption of methods varies when considering methods affected by zero, one, two, and three code smell types. Note

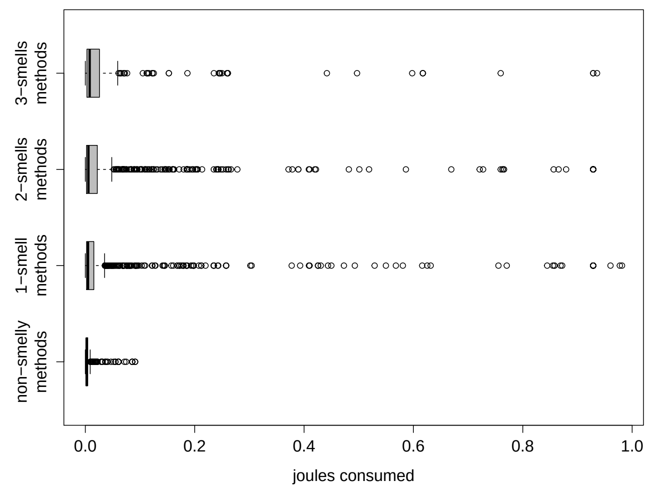

Fig. 2: Energy Consumption of methods having one, two, and three code smell types

TABLE II: Battery Discharge of Methods Affecting by Different Number of Code Smells

Number of Smells	% of Battery Discharge
0-smells	$6.9 \cdot 10^{-8}\%$
1-smell	$1.3 \cdot 10^{-6}\%$
2-smells	$1.9 \cdot 10^{-6}\%$
3-smells	$2.7 \cdot 10^{-6}\%$

that we have not found cases where more than three smells co-occurred in the same method. Figure 2 reports the results of this analysis. As shown, the higher the number of code smells in a method the higher the median energy consumption. In particular, we found that the median energy consumed by methods affected by three smell types is $0.008J$, which is twice the energy consumed by methods affected by only one smell ($0.004J$), and 39% times higher than methods affected by two smells ($0.0057J$). The differences are statistically significant (p-value<0.001 in both the cases) with a small effect size (d=0.14 and d=0.11, respectively).

From a practical point of view, these results lead to the battery discharge percentages shown in Table II. We can observe that each single execution of a method affected by a smell discharges the battery of at least $1.3 \cdot 10^{-6}\%$, while methods not affected by design flaws normally consume $6.9 \cdot 10^{-8}\%$ of the battery (*i.e.,* the battery discharges 19 times faster by executing smelly methods). Moreover, methods affected by more than one smell have a higher impact on the battery discharge, up to $2.7 \cdot 10^{-6}\%$ which means 40 times faster battery discharges than non-smelly methods.

Further analyses on the interaction between different smells revealed that most of the times there are four specific smells that co-occur together in methods having higher energy consumption, namely *Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*. Indeed, in 84% of the cases the methods having three smells are affected by

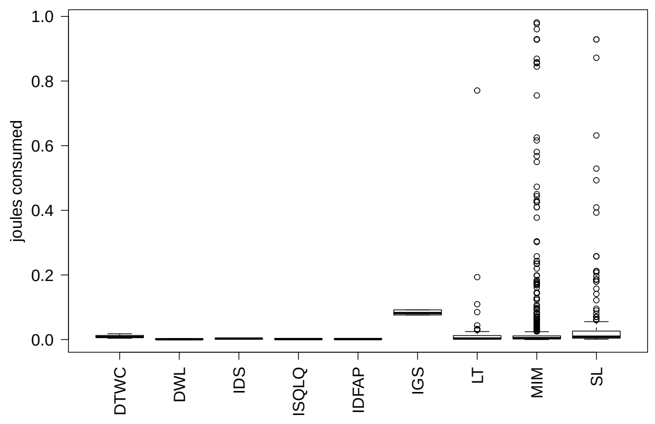

Fig. 3: Energy Consumption of methods affected by single code smell types

TABLE III: Battery Discharge of Different Types of Code Smells

Smell Type	% of Battery Discharge
DTWC	$8.7 \cdot 10^{-8}\%$
DWL	$7.7 \cdot 10^{-8}\%$
IDS	$7.7 \cdot 10^{-8}\%$
ISQLQ	$7.3 \cdot 10^{-8}\%$
IDFAP	$7.4 \cdot 10^{-8}\%$
IGS	$2.8 \cdot 10^{-6}\%$
LT	$1.1 \cdot 10^{-6}\%$
MIM	$1.4 \cdot 10^{-6}\%$
SL	$2.1 \cdot 10^{-6}\%$

a combination of these four specific design flaws. Moreover, the frequent co-occurrence of such smells is also visible when analyzing methods affected by two smells (*Member Ignoring Method* and *Slow Loop* co-occur in 33% of the methods, *Leaking Thread* and *Member Ignoring Method* in 28%, *Leaking Thread* and *Slow Loop* in 11%, *Internal Getter and Setter* and *Slow Loop* in 8%). It is important to note that methods where other types of smells co-occur tend to be more efficient than methods affected by a combination of the four smells mentioned above. From a practical point of view, this result tells us that not all the code smells influence the phenomenon, but rather particular smell types and their combinations can be a cause of energy leaks.

Summary for RQ$_1$. Overall, from our *coarse-grained* preliminary analysis we can conclude that the presence of code smells can result in a strong increment of the energy consumption (up to 385% with respect to smell-free methods, with a battery discharge of up to 40 times faster). Moreover, we found that methods affected by more smells consume more than methods not affected by design flaws. However, we observed four code smells, *i.e., Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*, that generally co-occur together and that may be the cause of energy leaks.

B. Which code smells have higher negative impact on the energy consumption of mobile apps?

From **RQ$_1$** we learnt that (i) methods affected by smells consume more than non-smelly methods, and (ii) methods having a specific combination of smell types tend to be less efficient. To have a *fine-grained* view of the impact of single code smell types on the energy consumption, we further analyzed the methods affected by each code smell. Figure 3 reports the boxplots comparing the distribution of the joules consumed by the different smell types considered in this study on the apps in our dataset. The first thing that leaps to the eye is that the four code smells co-occurring more frequently are those with higher impact on the energy consumption of the methods, *i.e., Internal Getter and Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*. This aspect confirms our conjecture that the energy consumption is influenced by specific types of design flaws. The claim is also supported by the percentage of battery discharged by methods affected by different smell types (Table III), where it is possible to see how methods affected by these four code smells have a much higher discharge rate than other methods (*e.g.,* each execution of a method affected by *Slow Loop* discharges the battery 30 times faster than non-smelly methods). As for the other smells, despite some of them being highly diffused (*e.g., Data Transmission Without Compression*) there is no clear relationship between their presence and energy consumption. In the following we report the detailed results, discussing each of the *most impacting* smells independently.

Internal Getter and Setter. According to the set of best practices for Android programming provided by Google [34], direct field access is up to 7 times faster than invoking a virtual method. During our experiments, we clearly experienced differences in methods affected and not affected by this smell. Indeed, we observed an evident increment of the energy consumption when a method performs a call to getters and setters. On average, such methods consume 40 times more energy than free smell methods ($0.08J$ vs $0.002J$). A representative example is the method `acaltime.AcalDateTime.applyLocalTimeZone` of the `aCal` app [35], which sets the local time zone by reading information about the actual zone using the method `getUTCInstance`, and then sets a new time using the method `setTimeZone`. The energy consumption of this method is, on average, 46% higher than non-smelly methods. The example is generalizable to the other methods affected by this smell and, indeed, the differences between the two distributions (smelly vs non smelly methods) are statistically significant (p-value<0.005) with a large effect size (d=0.97).

Leaking Thread. In Android programming a thread is a garbage collector (GC) root. The GC does not collect the root objects and, therefore, if a thread is not adequately stopped it can remain in memory for all the execution of the application, causing an abuse of the memory of the app. As an indirect consequence, the methods involved in this smell have a higher energy consumption with respect to the others. This smell implies a consumption of energy 15 times higher with respect to smell-free methods ($0.03J$ vs $0.002J$ considering the median). Other than the already mentioned class `a2dp.Vol.AppChooser` (see Section III-A), we found several other cases where this type of smell is involved in energy leaks. For instance, the method `onResume` of the `mileage.FillupListActivity` class contained in the `Mileage` app [36] opens a thread in order to model the behavior of the application when it is restarted. The missing closure of the thread implies an energy consumption more than 100% higher than classes not affected by any smell in the app. When applying the Mann-Whitney test, we compared the distributions of energy consumed by methods involved in *Leaking Thread* and methods that correctly close threads, finding the differences to be statistically significant (p-value<0.005) with a medium effect size (d=0.35).

Member Ignoring Method. This smell arises when a method does not access any field of the class it belongs to. It should be made `static` in order to speed up the invocations of such method [10]. In our context, we observed that in cases where the method is called several times the increment of energy consumed is 20 times higher with respect to non-smelly methods ($0.04J$ consumed on average in each call compared to the $0.002J$ consumed by non-smelly methods). A significant example appeared in the `Alarm Klock` app [37]. The method `onItemClick` of the `alarmclock.ActivityAlarmClock` class is responsible for capturing the taps of the user on the screen when using the app. This method is therefore called in action several times during each app execution. We observed an energy consumption 244% higher compared with other methods. We also observed a similar trend for all the other instances of this type of smell. Indeed, the Mann-Whitney test revealed that the differences are statistically significant (p-value<0.005) and the Cliff test reported a medium effect size (d=0.34).

Slow Loop. Methods affected by this smell iterate over collections using the standard version of the `for` loop, while the `for-each` loop can provide a better efficiency [10]. Our results show that methods involved in this type of smell, on average, consume almost 30 times more energy than non-smelly methods ($0.06J$ vs $0.002J$). As an example, the `onOptionsItemSelected` method, belonging to the `battery.BlinkenlightsBattery` class of the `Battery Circle` app [38] is responsible for analyzing all the possible configuration options provided as input by the user. In our experiment, we found the energy consumption of this method 80% higher than the other non-smelly methods of the same app. Also in this case, the differences between the two distributions (smelly vs non smelly methods) are significant (p-value<0.005) with a small effect size (d=0.25).

Other smells. Besides the smells mentioned above, we observed that other smell types do not have a relevant impact on the energy consumption of mobile applications. The result is quite surprising since previous research by Hecht *et al.* [39] showed that other smells (*e.g., IDFAP*) negatively impact on properties such as CPU usage, that are supposed to be somehow related to energy consumption. In our experiment, we found that methods affected by other types of smells (not discussed above) do not statistically differ from non-smelly methods in terms of energy consumption as confirmed by the Mann-Whitney test (p-values always higher than 0.27). Among all other design flaws, it is worth discussing the case of *Inefficient Data Structure*: recent findings by Hasan *et al.* [7] demonstrated how the application of a wrong type of data structure increases the energy consumption up to 300%. This seems to be in contrast with respect to our findings. However, it is important to note that Hasan *et al.* have conducted their experiments in the context of larger and more complex applications, such as Java libraries, while in the context of mobile apps we observed that this type of problem is generally poorly diffused (*i.e.,* only 0.6% of the methods analyzed are affected by this smell) and, therefore, not particularly relevant.

Summary for RQ$_2$. Our analysis reveals that there exist four *energy-smells*, *i.e., Leaking Thread*, *Member Ignoring Method*, *Slow Loop*, and *Internal Getter and Setter*, which significantly impact the energy consumption of methods in a Mobile app.

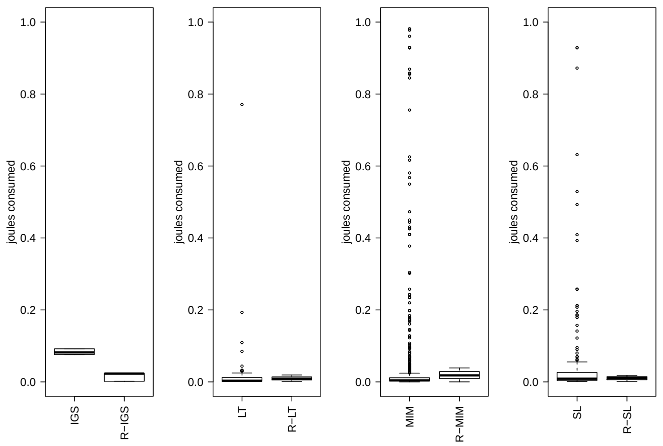

Fig. 4: Energy consumed by methods before and after the application of refactorings. 'R-' prefix stays for Refactored.

C. *Does the refactoring of code smells positively impact the energy consumption of mobile apps?*

The results of the previous research questions pointed out the existence of four specific energy smells for which it is worth to further investigate whether refactoring can effectively alleviate the corresponding energy leaks. To this aim, we manually refactored 1,344 instances of the four energy smells identified in the previous research question. Note that to study the effect of single refactorings applied to remove a given code smell, also in this case we consider the methods affected by only one smell.

Figure 4 reports the energy consumption of methods affected by *Internal Getter and Setter*, *Leaking Thread*, *Member*

Ignoring Method, and *Slow Loop*, respectively, before and after the application of the refactorings. In all pairs of box plots a recurring pattern can be observed: when the code smells affecting the methods are removed, the energy consumption of such methods decrease significatively.

For methods affected by *Internal Getter and Setter* we can observe that the associated solution, namely the *Direct Field Access* refactoring [10], reverses the negative effect of the smell by reducing the energy consumption by almost 9 times with respect to the smelly methods (the median energy consumption of smelly methods is 0.08 joules, while the median of the distribution of the refactored methods is $0.009J$). Another important observation is that the energy consumed by these methods after the refactoring is comparable with the energy consumed by methods that were not affected by any smell. This means that in this case the refactoring is able to remove the negative effects of the smell. The result is even more evident when considering the case of the previously mentioned `applyLocalTimeZone` method of the `aCal` app. By directly using the fields reporting actual zone and current time, the method consumes, on average, 0.012 joules, namely 683% less than the non-refactored version ($0.094J$). The magnitude of the differences between smelly and refactored methods is large (d=0.67) and statistically significant (p-value$<$0.005).

Similar results can be observed for the *Leaking Thread* smell, with the median energy consumption equal to $0.03J$ when methods are affected by the smell, compared to $0.003J$ when the methods are refactored. Hence, the usefulness of the *Introduce Run Check Variable* refactoring [10] is quantifiable in reducing 900% of the energy consumption of methods previously affected by a *Leaking Thread*. Also in this case, we discuss the differences between the smelly and the refactored version of the `AppChooser.onCreate` method of the `a2dpvolume` class, mentioned in the context of $\mathbf{RQ}_1$. From an average of 0.77 joules previously obtained, we observed that the energy consumed by the refactored version is $0.01J$, namely 77 times lower. The differences are statistically significant (p-value$<$0.001) with a medium effect size (d=0.35). Furthermore, it is important to note that the refactored method consumes considerably less than the other non-smelly methods.

The *Introduce Static Method* refactoring [10] needed to remove the *Member Ignoring Method* smell turned out to be strongly impacting on the energy efficiency of source code methods. Indeed, the boxplot shows that all the ouliers present in the smelly distribution disappear after the removal of the smells. On average, the energy consumption of methods change from a mean of $0.04J$ to $0.02J$, leading to an improvement of exactly 100%. Although the differences are statistically significant (p-value$<$0.001) with a medium effect size (d=0.46), it is worth considering that in this case the median energy consumption of the distribution of the refactored methods is higher than the one related to the smelly version of methods ($0.018J$ vs $0.004J$). This result appears to be not in line with the discussion up until now. However, this is just a reflection of the large number of outliers previously affecting the distribution. Indeed, analyzing more in depth this aspect, we observed that most of the outliers once refactored have an energy consumption that is slightly higher than the median of the non-smelly methods: as a consequence, the resulting distribution is more condensed and the statistical descriptors are higher than before. One example is the `ActivityAlarmClock.onItemClick` method of the `Alarm Klock` app, described before as a method consuming 244% more than non-smelly methods. Once refactored, we observed a strong improvement of its energy efficiency (in the ten runs the consumption decreases from 0.97 to 0.03 joules, namely 33 times lower), leading to an average consumption slightly higher than non-smelly methods.

The discussion about the *Slow Loop* smell is close to the one of the other energy smells. Indeed, the application of the *Enhanced For-Loop* refactoring improves, on average, the energy efficiency of 180% with respect to energy consumed by the smelly methods. Also in this case, all the outliers disappear after applying the refactoring and generally the distribution is much less scattered than before. The differences are statistically significant (p-value$<$0.005), yet, with a small effect size (d=0.12). Finally, it is worth observing that the consumption of refactored methods remains higher than the consumption of non-smelly methods ($0.0114J$ vs $0.002J$). While the motivations behind this aspect may be related to several other factors that are not the object of this study, in our context it is important to point out the huge benefit provided by the refactoring to the energy efficiency of such methods. The example discussed in $\mathbf{RQ}_2$ about the method `BlinkenlightsBattery.onOptionsItemSelected` of the `Battery Circle` app is significant in order to understand the effect of refactoring *Slow Loop* instances. Indeed, once refactored the method passed from a consumption of 0.87 joules to $0.01J$ (-870%), *i.e.,* it completely changed its energy profile becoming a green method.

TABLE IV: Battery Discharge of Methods Before and After Refactored

Smell Type	% of Battery Discharge Before	% of Battery Discharge After
IGS	$2.8 \cdot 10^{-6}\%$	$3.1 \cdot 10^{-7}\%$
LT	$1.1 \cdot 10^{-6}\%$	$1.1 \cdot 10^{-7}\%$
MIM	$1.4 \cdot 10^{-6}\%$	$6.2 \cdot 10^{-7}\%$
SL	$2.1 \cdot 10^{-6}\%$	$3.9 \cdot 10^{-7}\%$

To broaden the scope of the discussion, the quantity of source code that needs to be refactored to remove the code smells is small. For instance, the *Introduce Static Method* only requires the addition of the keyword `static` to the signature of the method, together with other small changes to adapt method calls over all the project. However, we observed that such *small changes in the source code result in a big change in the energy consumed* by the methods involved. The results are confirmed by the analysis of the percentage of battery discharge of methods before and after being refactored, as reported in Table IV. As we can see, all the types of refactoring result in a significantly lower energy drain. In our opinion, this is a key result since it reveals the actual cost-effectiveness of refactoring of *Android-specific* code smells.

Summary for $\mathbf{RQ}_3$. Refactoring code smells has a key role in improving the energy efficiency of source code methods. On average, we found that the refactored versions of methods consume almost 300% less than methods affected by smells. We also found several cases where refactoring helps in reducing the energy consumption up to 870%. Based on these results, we observe that refactoring is a cost-effective activity that should be applied by mobile developers.

IV. Threats to Validity

The main threats related to the relationship between theory and observation (*construct validity*) are due to imprecisions in the measurements we performed. Above all, we relied on a tool we built and made publicly available [12] to detect candidate code smell instances. Nevertheless, we are aware that our results can be affected by the presence of false positives and false negatives. However, we manually evaluated the performances of the tool on a statistically significant subset of the apps considered in the study, finding that `aDoctor` has a precision of 93% and a recall of 97%. These results allow us to be confident about the code smell instances found over all the considered apps. In addition, we replicated all the analysis performed to answer our research questions by just considering the 18 apps where the smells have been validated. The results achieved in this analysis (available in our replication package [12]) are inline with those obtained in our paper, confirming all our findings.

On the other hand, we measure energy consumption using our tool `PETrA`. As briefly explained in Section II, we empirically evaluated the accuracy of the approach on 23 mobile apps comparing the power estimation of the tool with the oracle provided by Linares-Vasquez *et al.* [5]. The validation revealed that in 91% of the cases the estimations of our tool are within 5% of the actual values. Therefore, we believe that the data provided by the tool are consistent and close enough to the actual energy consumption. Moreover, we aggregated the results given by `PETrA` using the mean operator. In our case, the mean can be considered significant because the energy consumption of the each exercised method tends to remain similar over the 10 runs and, therefore, the distribution of the energy consumption of each method does not contain outliers.

To study the effect of refactoring on energy efficiency ($\mathbf{RQ}_3$), we manually refactored the source code. The procedure involved two of the authors who carefully followed the guidelines provided by Reimann *et al.* [10]. At the end of the first stage of refactoring, the authors involved in the task opened a discussion aimed at double checking the refactoring operations individually performed. Still, we cannot exclude imprecision and some degree of subjectiveness (mitigated by the discussion) in the way we refactor the source code.

Threats related to the relationship between the treatment and the outcome (*conclusion validity*) are represented by the analysis methods exploited in our study. We discuss our results by presenting descriptive statistics and using proper statistical tests in order to assess the significance and the magnitude of our findings. In addition, the practical relevance of the differences observed in terms of energy consumption is highlighted by effect size measures.

Threats to *internal validity* concern factors that could influence our observations. We are aware that in principle we cannot claim a direct cause-effect relation between the presence of code smells and the energy consumption of methods. However, on the one hand the impact of code smells is firstly demonstrated by the fact that we observed a strong improvement of the energy efficiency when such smells were refactored ($\mathbf{RQ}_3$). On the other hand, we performed an additional analysis to verify whether other factors could have influenced our results. Specifically, we re-run our analysis by considering the energy consumption of methods having different (i) size, (ii) complexity, and (iii) level of test coverage. In particular:

1) We grouped together methods with similar size by considering their distribution in terms of size. Specifically, we computed the distribution of the lines of code of methods. This step results in the construction of (i) the group composed by all the methods having a size lower than the first quartile of the distribution of all the size of the methods, *i.e., small* size; (ii) the group composed by all the methods having a size between the first and the third quartile of the distribution, *i.e., medium* size; and (iii) the group composed by the methods having a smell and having a size larger than the third quartile of the distribution of all the size of the methods, *i.e., large* size;
2) We computed the energy consumption for each method belonging to the three groups, in order to investigate whether larger methods consume more with respect to smaller methods.

The experiment has been repeated considering the McCabe cyclomatic complexity [40] and the coverage obtained by the test cases ran over the methods analyzed [41] as measures to split methods in *small*, *medium*, and *large* sets. We reported the achieved results in our online appendix [12]. We observed that such characteristics are not strongly correlated with the energy consumption of the methods.

Finally, regarding the generalization of our findings (*external validity*) we evaluated the impact of 9 code smell types on the energy consumption of 60 mobile applications. However, further studies aiming at replicating our work on larger datasets are desirable and part of our future agenda.

V. Related Work

This section discusses the related literature about code smells and energy consumption.

A. About Code Smells and Refactoring

The traditional code smells defined by Fowler [9] have been widely studied in the past. Several studies demonstrated their negative effects on program comprehension [42], change- and fault-proneness [43], and maintainability [44], [45]. At the same time, several approaches and tools, relying on different sources of information, have been proposed to automatically detect [46], [47], [48], and fix them through the application of refactoring operations [49], [50]. Traditional code smells have also been studied in the context of Mobile apps by Mannan *et al.* [51], who demonstrate that, despite the different nature of mobile applications, the variety and density of code smells is similar. A more detailed overview about code smells and refactoring can be found in [52] and [53].

As for the Android-specific code smells defined by Reimann *et al.* [10], there is a little knowledge about them. Indeed, while the authors of the catalogue assumed the existence of a relationship between the presence of code smells and non-functional attributes of source code, they never empirically assessed it [10]. The unique investigation on the impact of three code smells on the performance of Android applications has been carried out by Hetch *et al.* [39], who found some positive correlations between the studied smells and the decreasing performance in term of delayed frames and CPU usage. To the best of our knowledge, our paper represents the first large scale empirical investigation about the role that code smells have on the energy usage of mobile applications.

B. About Energy Consumption

Themes related to energy consumption are becoming ever more relevant for the research community due to the large diffusion of smartphones. In recent years several hardware and software tools to estimate the energy consumption have been proposed, such as GreenMiner [19] or eLens [3]. Unfortunately, these tools are not publicly available. Our solution has been the construction of `PETrA`, a new software-based energy estimator based on reliable Android tools, having high accuracy in the power estimations.

On top of estimation tools, researchers have studied ways to predict the energy consumption using empirical data [54], [55] or dynamic analysis [56], [57], to study how changes across software versions affect energy consumption [1], or to estimate the energy consumed by single lines of code [58]. At the same time, several empirical investigations have been carried out. Sahin *et al.* [8] studied the influence of code obfuscation on energy consumption, finding that the magnitudes of such impacts are unlikely to impact end users. Sahin *et al.* also studied the role of design patterns [4], highlighting the existence of some patterns (*e.g.,* the *Decorator* pattern) negatively impacting the energy efficiency of an application. Similar results have been found by Noureddine and Rajan [59].

Hasan *et al.* [7] investigated the impact of the Collections type used by developers, demonstrating how the application of the wrong type of data structure can increase the energy consumption by up to 300%. Aling the same lines, other researchers focused their attention on the behavior of sorting algorithms [60], lock-free data structures [61], GUI optimizations [62], API usage of Android apps [5], providing findings on how to efficiently use different programming structures and algorithms.

The studies by Sahin *et al.* [6] and Park *et al.* [63] are closest to the one proposed in this paper. Sahin *et al.* [6] studied the effect of the refactorings defined by Fowler [9] on energy consumption. Specifically, they evaluated the behavior of 6 types of refactoring, finding that some of them, such as *Extract Local Variable* and *Introduce Parameter Object*, can both increase or decrease the amount of energy used by an application [6]. On the other hand, Park *et al.* [63] experimented with 63 different refactorings and propose a set of 33 energy-efficient refactorings. It is worth noticing that these papers analyzed the effect of refactoring independently of the presence of design flaws. In contrast, our study analyzes the impact of code smells specifically defined for Mobile applications [10] on energy consumption as well as the influence of refactoring operations aimed at removing them from the source code.

VI. Conclusion

This paper presents a large scale empirical investigation taking into account the role of 9 *Android-specific* code smells [10] on the energy consumption of Mobile apps. Starting from a *coarse-grained* analysis in which we analyze whether source code methods affected by smells have higher energy consumption than non-smelly methods, we then refine our analyses in order to investigate which are the specific code smell types that influence more the phenomenon. Finally, we evaluated whether refactoring operations applied to remove code smells help in substantially improving the efficiency of Mobile applications. The achieved results provide valuable findings and insights for the research community.

Lesson 1. *Code smells have a strong impact on the energy efficiency of source code methods.* Specifically, methods affected by smells consume up to 385% more with respect to smell-free methods. Even if the interaction between them results in lower efficiency, we found four particular smell types that frequently co-occur and that impact energy consumption more than others, *i.e., Leaking Thread*, *Member Ignoring Method*, *Slow Loop*, and *Internal Getter and Setter*. These aspects highlight the importance of investing (1) in studying more in depth the dynamics behind *Android-specific* code smells and (2) in developing tools that prevent their introduction.

Lesson 2. *Refactoring code smells is a key activity to improve energy efficiency.* We found that the energy consumption of refactored methods is reduced by up to 900% with respect to smelly methods. Therefore, we empirically demonstrated how small changes applied to remove code smells result in applications that are much more efficient in terms of energy consumption. Approaches and tools able to support Mobile developers in automatically refactoring the source code represent a *must* for future research in the field.

These lessons represent the main input for our future research agenda on this topic, mainly focused on designing and developing a new generation of code quality-checkers and refactoring tools, other than corroborating our results by studying the impact of other smells (*e.g.,* the Fowler's smells [9]) on energy efficiency.

References

[1] A. Hindle, "Green mining: A methodology of relating software change and configuration to power consumption," *Empirical Softw. Engg.*, vol. 20, no. 2, pp. 374–409, Apr. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9276-6

[2] A. Smith, "Smartphone ownership," 2013. [Online]. Available: http://pewinternet.org/Reports/2013/Smartphone-Ownership-2013/Findings.aspx

[3] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 92–101. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486801

[4] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *Proceedings of the First International Workshop on Green and Sustainable Software*, ser. GREENS '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 55–61. [Online]. Available: http://dl.acm.org/citation.cfm?id=2663779.2663789

[5] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597085

[6] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14. New York, NY, USA: ACM, 2014, pp. 36:1–36:10. [Online]. Available: http://doi.acm.org/10.1145/2652524.2652538

[7] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 225–236. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884869

[8] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause, "How does code obfuscation impact energy usage?" in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 131–140. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.35

[9] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[10] J. Reimann, M. Brylski, and U. Amann, "A tool-supported quality smell catalogue for android developers," 2014.

[11] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 429–440. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.89

[12] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. (2016) On the impact of code smells on the energy consumption of mobile applications - replication package. [Online]. Available: https://dx.doi.org/10.6084/m9.figshare.3759489

[13] F-Droid, "F-droid repository. https://f-droid.org."

[14] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[15] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[16] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[17] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 623–640. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509552

[18] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures.*, second edition ed. Chapman & Hall/CRC, 2000.

[19] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "Greenminer: A hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597097

[20] D. Di Nucci, F. Palomba, A. Panichella, A. Zaidman, and A. De Lucia, "Petra. https://github.com/dardin88/PETrA."

[21] Android Tools, "dmtracedump. http://tinyurl.com/hqmspxl."

[22] ——, "Systrace. http://tinyurl.com/zpcshng."

[23] ——, "Battery stats. http://tinyurl.com/jkdntt6."

[24] Power Monitor, "Monsoon powermonitor. http://tinyurl.com/3ys7arm."

[25] D. Li, S. Hao, J. Gui, and W. G. J. Halfond, "An empirical study of the energy consumption of android applications," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, Sept 2014, pp. 121–130.

[26] Android Stackexchange, "Usb charging. http://tinyurl.com/jg7q3lf."

[27] Android Tools, "Monkey. http://tinyurl.com/gvnxdd3."

[28] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 94–105. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931054

[29] Simiasque, "Simiasque. http://tinyurl.com/jxkor7a."

[30] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[31] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[32] R. H. Romer, *Energy : an introduction to physics*. San Francisco: Freeman, 1976.

[33] a2dp volume, "a2dp volume app. https://play.google.com/store/apps/details?id=a2dp.Vol."

[34] Android Developers, "Android best practices. http://tinyurl.com/j9wucev."

[35] aCal, "acal app. http://tinyurl.com/8hg67fk."

[36] Mileage, "Mileage app. http://tinyurl.com/cw3uttu."

[37] Alarm Klock, "Alarm klock app. http://tinyurl.com/ngzkv3v."

[38] Battery Circle, "Battery circle app. http://tinyurl.com/o33ms7d."

[39] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the 3rd IEEE/ACM International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft16, 2016.

[40] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[41] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Commun. ACM*, vol. 6, no. 2, pp. 58–63, Feb. 1963. [Online]. Available: http://doi.acm.org/10.1145/366246.366248

[42] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, 2011, pp. 181–190.

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

[44] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1144–1156, 2013.

[45] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 403–414. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818805

[46] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, 2010.

[47] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *Software Engineering, IEEE Transactions on*, vol. 41, no. 5, pp. 462–489, May 2015.

[48] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "A textual-based technique for smell detection," in *Proceedings of the 24th International Conference on Program Comprehension (ICPC 2016)*. Austin, USA: IEEE, 2016, p. to appear.

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

[50] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, July 2014.

[51] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: ACM, 2016, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/2897073.2897094

[52] F. Palomba, D. A., G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues," *Advances in Computers*, vol. 95, pp. 201–238, 2015.

[53] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," pp. 387–419, 2014.

[54] N. Amsel and B. Tomlinson, "Green tracker: A tool for estimating the energy consumption of software," in *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '10. New York, NY, USA: ACM, 2010, pp. 3337–3342. [Online]. Available: http://doi.acm.org/10.1145/1753846.1753981

[55] K. Aggarwal, C. Zhang, J. C. Campbell, A. Hindle, and E. Stroulia, "The power of system call traces: Predicting the software energy consumption impact of changes," in *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '14. Riverton, NJ, USA: IBM Corp., 2014, pp. 219–233. [Online]. Available: http://dl.acm.org/citation.cfm?id=2735522.2735546

[56] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/1878961.1878982

[57] A. Gupta, T. Zimmermann, C. Bird, N. Nagappan, T. Bhat, and S. Emran, "Mining energy traces to aid in software development: An empirical case study," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14. New York, NY, USA: ACM, 2014, pp. 40:1–40:8. [Online]. Available: http://doi.acm.org/10.1145/2652524.2652578

[58] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 78–89. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483780

[59] A. Noureddine and A. Rajan, "Optimising energy consumption of design patterns," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 623–626. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819120

[60] C. Bunse, H. Hpfner, E. Mansour, and S. Roychoudhury, "Exploring the energy consumption of data sorting algorithms in embedded and mobile environments," in *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, May 2009, pp. 600–607.

[61] N. Hunt, P. S. Sandhu, and L. Ceze, "Characterizing the performance and energy efficiency of lock-free data structures," in *2011 15th Workshop on Interaction between Compilers and Computer Architectures*, Feb 2011, pp. 63–70.

[62] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of guis in android apps: A multi-objective approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786847

[63] J.-J. Park, J.-E. Hong, and S.-H. Lee, "Investigation for software power consumption of code refactoring techniques," in *Proceedings of the Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering*, ser. SEKE '14, 2014.