Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

On the impact of code smells on the energy consumption of mobile applications

Fabio Palomba*,[a], Dario Di Nucci[b], Annibale Panichella[c], Andy Zaidman[c], Andrea De Lucia[d]

[a] University of Zurich, Binzmuhlestrasse 14, Zurich CH-8050, Switzerland
[b] Vrije Universiteit Brussel, Pleinlaan 2, Elsene 1050, Belgium
[c] Delft University of Technology, Mekelweg 2, CD Delft 2628, The Netherlands
[d] University of Salerno, Via Giovanni Paolo II, 132, Fisciano 84084, Italy

A R T I C L E I N F O

Keywords:
Code smells
Refactoring
Energy consumption
Mobile apps

A B S T R A C T

Context. The demand for green software design is steadily growing 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. **Objective.** Despite the efforts spent so far, only a little knowledge on the influence of code smells, *i.e.,*symptoms of poor design or implementation choices, on the energy consumption of mobile applications is available. **Method.** To provide a wider overview on the relationship between smells and energy efficiency, in this paper 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 are theoretically supposed to be related to non-functional attributes of source code, such as performance and energy consumption. **Results.** The results of the study highlight that methods affected by four code smell types, *i.e.,Internal Setter, Leaking Thread, Member Ignoring Method*, and *Slow Loop*, consume up to 87 times more than methods affected by other code smells. Moreover, we found that refactoring these code smells reduces energy consumption in all of the situations. **Conclusions.** Based on our findings, we argue that more research aimed at designing automatic refactoring approaches and tools for mobile apps is needed.

1. 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 have successfully demonstrated how even software may be at 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 [4–7].

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 in 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 using the 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, little knowledge is available in literature on the potential impact on energy consumption of the so-called *bad code smells* (also named *code smells* or simply *smells*) defined by Reimann et al. [9]. Unlike the traditional smells introduced by Fowler [10], these smells represent a set of bad programming practices in Android mobile applications. While the impact of these smells on energy consumption has been theoretically assumed by Reimann et al. [9], there is a lack of

* Corresponding author.
E-mail addresses: fpalomba@unisa.it, palomba@ifi.uzh.ch (F. Palomba).

https://doi.org/10.1016/j.infsof.2018.08.004
Received 10 January 2018; Received in revised form 22 June 2018; Accepted 9 August 2018
Available online 05 September 2018

studies providing evidence on this impact. Indeed, only the work by Carette et al. [11] initially explored the relationship between smells and energy efficiency. However, they analyzed the behavior of just three code smell types on five mobile apps, finding that the removal of such code smells has a limited effect on energy efficiency (quantified as a 4% improvement of the overall energy consumption).

In this paper, we provide a deeper investigation to determine (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 directly improves the energy efficiency of refactored methods. In particular, our investigation focuses on 9 method-level code smells specifically defined for mobile applications by Reimann et al. [9] in the context of 60 Android apps belonging to the dataset provided by Choudhary et al. [12]. To the best of our knowledge, this is up to date the largest study aimed at practically investigating the actual impact of these code smells on energy consumption and quantifying the extent to which refactoring code smells is beneficial for improving energy efficiency.

To conduct our analyses, we built upon two tools that we previously developed and evaluated, *i.e.,* ADOCTOR and PETRA. The former is a novel *Android-specific* code smell detector that has been evaluated in our prior study [13] using 18 apps and it is very accurate, with a precision of 98% and a recall of 98%. The latter is a software-based tool that estimates the energy profile of mobile applications [14]. It has been evaluated using 54 apps [14] and provides an estimation error within 5% of the actual values measured with a hardware-based tool [5] in 95% of the cases.

Results of our study highlight the existence of four specific *energy-smells*, namely *Internal Setter, Leaking Thread, Member Ignoring Method*, and *Slow Loop*: methods affected by these design flaws consume up to 87 times more energy than methods affected by other code smells. Moreover, we shed light on the usefulness of refactoring as a way of improving energy efficiency by code smell removal. Specifically, we found that it is possible to improve the energy efficiency of source code methods by refactoring code smells.

Structure of the paper. Section 2 reports the design of the empirical study, while Section 3 describes the results achieved. Section 4 discusses the threats that could affect the validity of the results. Section 5 summarizes the related literature in the context of green mining and code smells. Finally, Section 6 concludes the paper.

2. 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 addresses the following three research questions:

- **RQ1:** *To what extent are the considered code smells diffused in the methods of the analyzed applications?*
- **RQ2:** *Do methods affected by code smells have high energy consumption?*
- **RQ3:** *Does the refactoring of code smells positively impact the energy consumption of mobile apps?*

The first research question (**RQ1**) is a preliminary investigation into the diffuseness of code smells in our dataset: this was done with the aim of studying the relevance of the considered problem and the extent to which each code smell type affects source code methods of Android apps. Our **RQ2** represents a deeper investigation into the relationship between the presence of code smells and energy consumption of the affected methods, while **RQ3** aims at assessing the gain provided by refactoring of code smells in terms of energy consumption.

2.1. Context selection

The *context* of the study consists of 60 open source Mobile Android apps publicly available in the F-Droid repository[1]. Specifically, we selected all the apps from the benchmark dataset provided by Choudhary et al. [12], which collects a subset of apps used in previous studies [15–18] having different size and scope and that are still maintained by their own developers. The dataset comprises 2701 classes and 19,504 methods. The complete list of the apps used in this study is available in our online appendix [19].

Table 1 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 catalog defined by Reimann et al. [9]. This catalog 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 catalog, and no empirical evaluation has been directly conducted to verify and measure such an impact.

Among the 30 types of Android-specific design flaws available in the catalog, 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 catalog 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 [9]: 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 [10]) are particularly challenging because objects (instances of a class) can remain in memory during the execution of the app[2] 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.

It is worth observing that we included in our selection the so-called *Internal Getter/Setter*: this smell arises when methods access external fields using getters and setters. While this is usually a good design practice that allows the encapsulation of the internal fields of a class, using getters and setters may lead a mobile app to be less performing as they represent additional function calls [9]. Since ANDROID 2.3, the compiler automatically optimizes *"simple getters that do nothing other than return the field"* [20], while it does not optimize for setters. On the one hand, this means that taking into account this smell still make sense, as internal setters might have an impact on energy efficiency. At the same time, the original definition of the code smell needs to be revised. For this reason, in the context of this paper we considered as smelly methods only those accessing internal fields via setters. In other words, we only considered the *Internal Setter* component of the original smell definition.

2.2. Data extraction

In this section, we provide an overview of the data extraction

[1] https://f-droid.org/

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

Table 1
The Code Smells considered in our study.

Abbreviation	Name	Description	Refactoring
DTWC	Data Transmission Without Compression	A method transmitting a file over a network 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
IS	Internal Setter	Internal fields are set via 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

process to (i) detect code smell instances and (ii) measure the energy consumption of the considered apps.

2.2.1. *Code smell detection*

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 relied on a code smell detector that we previously developed, named ADOCTOR [13]. ADOCTOR extracts structural properties from the source code to detect 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 by Fowler [10]. 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 by identifying the methods using an instance of `HashMap<Integer, Object>`. The complete list of detection rules exploited by ADOCTOR is available in [13]. To evaluate the performance of our tool in detecting smells, we have conducted a case study involving 18 apps in our dataset, finding that ADOCTOR is able to suggest code smells with an average precision of 98% and an average recall of 98%. Thus, the detection process is quite effective. The interested reader can find the publicly available version of the tool as well as more information about its validation in [13].

2.2.2. *Energy consumption estimation*

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 [21]. For this reason, in our study, we relied on PETRA (**P**ower **E**stimation **T**ool fo**r** **A**ndroid), which is publicly available [14]. PETRA is a software-based approach that is 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. PETRA records the time needed to complete the execution of 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.

More formally, as depicted in Listing 1, PETRA's main process is composed of three main blocks: (i) app preprocessing, (ii) energy profile computation, and (iii) output generation. In the following paragraphs, we detail each part independently.

App preprocessing. In the first step, PETRA needs to set the software environment before measuring the energy consumed when executing a mobile app. To this aim, it uses as input an executable version of the app under analysis in the form of an `apk` file. The app is identified by the `apk` location and the name of the app to profile, which correspond to `apkLocation`, and `appName` in Listing 1 respectively. Then, PETRA installs the `apk` on a mobile phone able to run it (*e.g.,*a smartphone having an arbitrary version of the Android operating system) and enables the `debuggable` option. Enabling debugging is mandatory because otherwise the instrumentation and profiling of the app would not be possible.

Energy profile computation. Once the app is properly set up, PETRA exercises the app under consideration using a test case given as input, *i.e.,*`tCase` in Listing 1. This test case can be created with automated tools (*e.g.,*`Monkeyrunner` or `Monkey`) or with manual operations performed by the software engineer. Once the test case is run, the *core* process behind PETRA starts.

For the profiling phase, we leverage the `Project Volta` Android tools[3], which are based on the self-modeling paradigm proposed by Dong and Zhong [22], *i.e.,*the definition of a mobile system that automatically generates its energy model without any external assistance. Such tools are `dmtracedump`[4], `Batterystats`[5], and `Systrace`[6]. Specifically:

- `dmtracedump` provides an alternate way to show trace log files. The files generated by `dmtracedump` are easy to parse and allow the developers to establish precisely, at microseconds granularity, when a method call has been invoked and when it returned. PETRA relies on this component to store the execution traces of the app under analysis. For each method call `dmtracedump` provides the entry and the exit time. The final output is a list of the executed method calls during the run.
- `BatteryStats` is an open source tool of the Android framework able to collect battery data from the device under evaluation. In particular, it is able to show which processes are consuming battery energy and which tasks should be modified to improve battery life. It is executable via the command line. The data collected can be analyzed as a log file or can be converted to an HTML visualization that can be viewed in a browser using `Battery Historian`. PETRA uses the `Batterystats` log to retrieve the active smartphone components and their status in a specific time window. Furthermore, it can provide the information about the device voltage. Given this information, it is then possible to calculate the energy consumed by the smartphone during a time window.
- `Systrace` is a tool that can be used to analyze application performance. It captures and displays the execution time of the active processes of a smartphone, combining data from the Android kernel, *i.e.,*the CPU scheduler, disk activity, and application threads. The data can be viewed as an HTML report that shows the overview of the processes in a given time window. In PETRA, the information provided by `Systrace` is used to capture the frequency of the CPU in a given time window. Considering that CPUs have different consumptions as their frequency varies, this information completes the one provided by `Batterystats` improving the estimations.

[3] https://developer.android.com/about/versions/android-5.0.html#Power
[4] https://developer.android.com/studio/profile/traceview.html
[5] https://developer.android.com/studio/profile/battery-historian.html
[6] https://developer.android.com/studio/profile/systrace-commandline.html

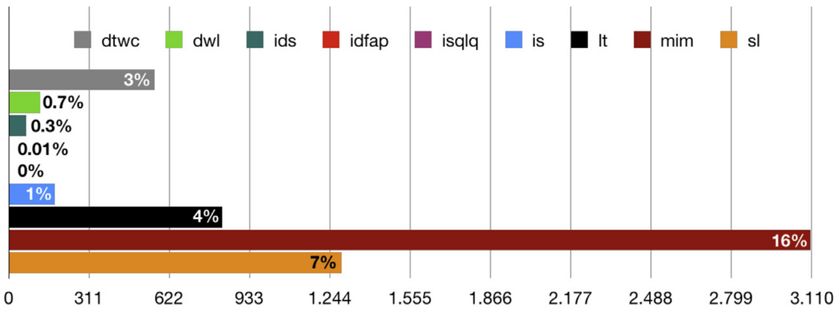

Fig. 1. Diffusness of code smells on the considered dataset.

After gathering the information related to the active components with their status, the CPU frequencies and the method call invocations, the `power profile` file is loaded. The `power profile` values define the current consumption for a component along with an approximation of the battery drain caused by each component over time. For instance, it specifies how many MilliAmperes of current are required to run the CPU at a certain frequency. Every smartphone has its own `power profile`. It is worth noting that each device manufacturer must provide this information and that it can be found in a defined location in the device[7].

Given the previous data, it is possible to compute the energy consumed for every method call invocation. First of all, given a method call invocation and its termination we can calculate the overall time window T_w as the arithmetic difference between the two instants of time when these two events occurred. However, the energy consumed within one single time window is not constant but may change because of a CPU frequency variation or a component state change. Therefore, we divided the time windows into smaller time units, *i.e.,*data frames T_Δ. When the entry to a method is registered, a new time window T_w and a new time frame T_Δ start. Whenever a component changes its state, the existing time frame T_Δ is terminated and a new one (for the new state) is started. When the exit point to a method is registered, then the corresponding time window T_w is terminated as well as the latest time frame T_Δ. In this way, each data frame T_Δ is characterized by coherent component states (*e.g.,*CPU frequency) and by a coherent (constant) energy drain. For example, if the CPU is working at the maximum frequency and none of the components change their state, the time windows T_w will be composed by only one-time frame T_Δ of the same duration, *i.e.,*the difference between the method entry and exit. Therefore, we can calculate the current power intensity at each time frame T_Δ as follows:

$$I_\Delta = \sum_{\forall c \in C} I_{\Delta,c} \tag{1}$$

where C is the set of smartphone hardware components, $I_{\Delta,\ c}$ is the current intensity of the component c within the current time frame T_Δ. For example, 92.6 is the number of MilliAmpere consumed in one second by a Nexus 4 when the CPU frequency is fixed at 384 Mhz.

After calculating the current intensity, it is possible to calculate the energy consumed in a time frame, as follows:

$$J_\Delta = I_\Delta \times V_\Delta \times T_\Delta \tag{2}$$

where J_Δ is the consumed energy in Joule, I_Δ is the current intensity in Ampere, V_Δ is the device voltage in Volt, and T_Δ is the length of the time frame in seconds.

Finally, the energy consumed by a method call can be calculated by summing up the energy consumed in each time frame in which the method call was active:

$$J = \sum_{T_\Delta \in T_w} (I_\Delta \times V_\Delta \times T_\Delta) \tag{3}$$

Output generation. The final output provided by PETRA is a `CSV` file, containing the energy estimation for each method call. More precisely, it provides the signature of each executed method call, along with the consumption in Joule and the execution time in seconds.

Finally, PETRA relies on the `Android Activity Manager`[8], so the `apk` must be enabled for debugging. Furthermore, in order to provide a better estimation, PETRA exercises the app multiple times (`nRuns` in Listing 1). Note that in our experiments `nRuns` is fixed to 10 and that in order to avoid any bias due to multiple runs, at the start of each run the app cache is cleaned and `Batterystats` is reset (lines 4 and 5 in Listing 1).

PETRA is able to provide power estimations similar to those obtained using the power estimation model provided by Android. 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`[9]. To this aim, we used the same phone (*i.e.,*`LG Nexus 4`) and settings used by Linares–Vasquez et al. [5]. In summary, the results show that the estimations produced by PETRA are very close to the actual values, more precisely:

- The mean estimation error achieved using PETRA is 0.04 with respect to the actual value calculated using `Monsoon`.
- The measurement errors are mainly due to a significant use of network capabilities or sensors.
- In 89% of the cases, PETRA produces overestimations, mainly due to the accumulated noise achieved during the estimations. In the remaining cases, the use of sensors and network produces underestimations.

The interested reader can find more information about the validation of the tool in [14]. In the context of this work, we followed a well-defined process already used in previous work [3,5,21,23] to extract the energy profile of the 60 Mobile apps:

- The phone used in the experiment is a factory re-setted `LG Nexus 4` having Android 5.1.1 Lollipop as operating system, equipped with a 1.5 GHz quad-core Snapdragon S4 Pro-processor with 2 GB of RAM, and having a 2100 mAh, 3.8 V battery. The choice of the phone was guided by previous research in the field [3,5,21,23], but also because this particular hardware allows to be connected via a data cable, namely a cable where the USB charging can be disabled[10]. Therefore, no energy is transferred over the cable, enabling more stable measurements. Before starting the experiment, we completely reset the phone to avoid bias in the power measurements. Moreover, to limit noise (i) we disabled all the unnecessary apps and processes

[7] https://source.android.com/devices/tech/power/values.html

[8] https://developer.android.com/studio/command-line/shell.html

[9] http://tinyurl.com/3ys7arm

[10] http://tinyurl.com/jg7q3lf

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 [19], was needed 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 using *Monkey*[11], 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 [12,24], showing that this tool achieves the better compromise between coverage and effort needed for the setup. In the experiment, we used the configuration of Monkey suggested by Choudhary et al. [12]. 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, to not improperly enable/disable smartphone functionality (*e.g.,*WiFi, Bluetooth, GPS), we hid the status bar[12].
- The measurements provided by PETRA were repeated 10 times to have a more reliable estimation of the energy profiles. Each run costs around five minutes since, as reported by Choudhary et al. [12], 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. In our case, the mean can be considered significant because the energy consumption of 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. In particular, to verify the normality of the distribution of the energy consumptions we adopted the following process: (i) we normalized the data of each run in the interval [0,1] using the mix-max algorithm [25] and (ii) we applied the Shapiro–Wilk test [26] with an α threshold for significance set to 0.05. It is important to remark that for this test the null hypothesis represents the normality of the distribution. In our case, the $\rho - value$ assumed value equals to 0.4921 and thus we could not reject the hypothesis that the sample comes from a population which has a normal distribution, meaning that the variance of the distribution is not large and, therefore, the mean operator can be considered.

  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.

2.3. Data analysis

Once we extracted the code smell instances affecting the apps with ADOCTOR, we turned our attention to answer **RQ1**: we verified the diffuseness of each considered code smells in our dataset and we computed the absolute and relative number of methods they affected. In **RQ2**, we aimed at providing a first understanding of the relation between the presence of code smells and energy consumption of the affected methods. Exploiting the energy profiles coming from the output of PETRA, we proceeded as follows: as a first step, we ordered the methods in our dataset by energy consumed to obtain a ranked list having at the top the most consuming methods. Then, we computed how many of those methods were affected by a certain type of code smell. In other words, we assessed how many methods ranked at the top of the list were affected by a code smell, with the aim of understanding the extent to which code smells represent a co-occurring phenomenon with respect to energy consumption. More specifically, given the ranked list we investigated how many methods of the top α% of them were also affected by each of the considered smells. We set α% with values in the set [10, 20, 30, 40, 50]: in this way, we measured the extent of the relation in the first half of the most consuming methods.

Finally, as for **RQ3**, we evaluated whether refactoring operations (originally targeted to remove the smells) are actually useful for reducing the energy consumption of the smelly methods. 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. [9]. In particular, the methods to analyze and refactor have been distributed among two of the authors, who were responsible for refactoring half of the instances each. Each of the involved authors independently refactored the methods assigned to him, by relying on (i) the definitions of refactoring and (ii) the examples provided by Reimann et al. [9]. The output of this phase consisted of the source code where code smells were removed. Then, the two authors involved in this task discussed their activities 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>` [9]. 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 **RQ**) at the end of each refactoring. Once refactored the source code, we repeated the energy measurements. Then, we compared the energy consumption obtained using the smelly version of the app with the energy consumption obtained by its corresponding refactored version. To test the statistical significance of the differences (if any) between such distributions we used the Mann–Whitney test [27]. The results are intended as statistically significant at $\alpha = 0.05$. We estimated the magnitude of the measured differences using Cliff's Delta (or *d*), a non-parametric effect size measure [28] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $0.10 \le |d| < 0.33$, medium for $0.33 \le |d| < 0.474$, and large for $|d| \ge 0.474$ [28].

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 affected by code smells. In particular, given the characteristics of the phone used in the experiments (*i.e.,*2100 mAh and 3.8V battery), the percentage of battery discharge can be computed using the following formula [29]:

$$f(n) = \left(\frac{V \cdot S \cdot I}{V} \cdot \frac{1}{I \cdot S}\right)\% \tag{4}$$

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 Eq. (4) (*i.e.,*$V \cdot S \cdot I$) is exactly the number 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 Eq. (4) is instantiated as follow:

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

where the value of 1000 at the numerator is because the charge is expressed in mAh and not in Ah, and 3600 is used to convert hours to seconds.

[11] http://tinyurl.com/gvnxdd3
[12] http://tinyurl.com/jxkor7a

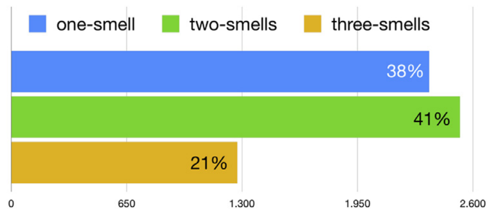

Fig. 2. Diffuseness of code smell co-occurrences on the considered dataset.

3. Analysis of the results

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

3.1. To what extent are the considered code smells diffused in the methods of the analyzed applications?

Fig. 1 shows the code smells diffuseness throughout the entire dataset. In particular, ADOCTOR detected 6155 code smell instances (*i.e.,* ≈ 32% of the total 19,504 methods are smelly). The most frequent ones are: *Member Ignoring Method* (3104 instances, 16% of all the methods), *Slow Loop* (1288 instances, 7% of all the methods), *Leaking Thread* (828 instances, 4% of all the methods), and *Data Transmission Without Compression* (564 instances, 3% of the smelly methods) code smells. At the same time, the least diffused smells are *Inefficient Data Format And Parser* and *Inefficient SQL Query*, with 3 and 0 instances, respectively. As a consequence, we can firstly conclude that the earlier cited smells are those having the highest relevance in our dataset, meaning that they are the ones that appear more frequently in mobile apps. For the sake of our study, these results had a major outcome: we excluded the *Inefficient SQL Query* smell from the analysis. At the same time, despite its extremely low diffuseness we still kept the *Inefficient Data Format And Parser* into consideration: it might theoretically happen that even few instances of such code smell have a strong impact on energy efficiency. Thus, we kept evaluating the extent to which this smell contributes to energy consumption. The detailed results about the distribution of the studied smells over the 60 apps are reported in our online appendix [19].

While our first analysis targeted the diffuseness of the single code smell types, it is important to note that methods in our dataset may be affected by multiple smell types. Several studies in the past [30–32] have shown that smell co-occurrences might amplify the negative effects on the source code. For this reason, we took into account the phenomenon of code smell co-occurrences. Fig. 2 shows the results of this analysis. It is worth noting that we did not find any method affected by more than three code smell types at the same time. Looking at our findings, it is interesting to note that 62% of the methods are affected by more than one smell: this confirms the recent findings by Palomba et al. [32] on the high relevance of the co-occurrence phenomenon. Further analyses on the interaction between different smells revealed that most of the times there are four specific smells that co-occur together, namely *Leaking Thread, Member Ignoring Method, Slow Loop*, and *Internal Setter*. In particular, in 84% of the cases, the methods having three smells are affected by 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 Setter* and *Slow Loop* in 8%). Interestingly, the *Data Transmission Without Compression* smell, despite its diffuseness, generally tends to arise alone: this indicates that its high diffuseness does not necessarily imply a high co-occurrence with other smells.

As a matter of fact, the results of this first research question revealed that some code smell types tend to occur and co-occur more frequently: more detailed analysis of the impact of such smells (if any) on energy consumption is presented in the next section.

Summary for RQ1. Overall, four of the code smells analyzed tend to occur more frequently: they are the *Leaking Thread, Member Ignoring Method, Slow Loop*, and *Data Transmission Without Compression* ones. The latter three, together with the *Internal Setter*, are those co-occurring more frequently.

3.2. Do methods affected by code smells have high energy consumption?

As explained in Section 2, to investigate the impact of code smells on energy consumption, we firstly ordered the methods in our dataset by energy consumed, with the aim of assessing how many of those ranked at the top of the list were affected by a code smell. Overall, we observed that in the top 50% of the list (*i.e.,*9752 methods), 3120 methods were smelly (32%), while 2773 smelly methods (47%) were in the top 30% (*i.e.,*5851 methods) and 1835 (94%) were in the top 10% (*i.e.,*1950 methods). These results somehow suggest a relation between code smells and energy consumption. A representative example can be found in the `a2dpvolume` project[13], an app able to automatically adjust the media volume when the phone is connected to Bluetooth devices. The method `onCreate` of the `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 J on average (*i.e.,*it was the 58th most consuming method in our experiment).

The fine-grained findings of our analysis are reported in Fig. 3: specifically, the figure shows the percentage of the top-α% (with $\alpha = 10$, 20, 30, 40, 50), energy consuming methods that were also affected by each of the code smells in our dataset. The plot highlights that four code smells tend to frequently occur in the most consuming methods, namely *Member Ignoring Method, Slow Loop, Leaking Thread*, and *Internal Setter*.

On the other hand, instances of *Data Transmission Without Compression, Durable Wakelock, Inefficient Data Structure*, and *Inefficient Data Format and Parser* appeared in just the 5%, 4%, 3%, 0.001%, respectively, of the top-10% of the most consuming methods. Nevertheless, it is important to note that such percentages might be biased by the low diffuseness of these smells. For this reason, we also analyzed how many of the methods affected by a given smell are included in the top-α% (with $\alpha = 10$, 20, 30, 40, 50) most consuming methods, with respect to the total number of methods affected by that smell. The results are graphically shown in Fig. 4. As it is possible to see, we discovered that in the top-10% of the list appeared (i) 67% of all the *Internal Setter* instances (*i.e.,*119 out of the total 178 instances), (ii) 62% of the *Durable Wakelock* ones (*i.e.,*75/122), (iii) 81% of all the *Inefficient Data Structure* instances (55/68), and (iv) 66% of *Inefficient Data Format and Parser* ones (2/3). In other words, we can conclude that despite their low diffuseness, most of the instances of these smells appear in the top-10% of the most consuming methods, indicating that they potentially have an effect on energy consumption. The next research question aims at investigating further the causality between the presence of code smells and energy consumption of the affected methods. We analyzed all the smells, independently of their diffuseness.

As a final point of discussion, it is worth analyzing the relevance of code smell co-occurrences. Fig. 5 shows that methods affected by two smells simultaneously are those more frequently appearing at the top of the list of the most consuming methods.

[13] https://play.google.com/store/apps/details?id=a2dp.Vol

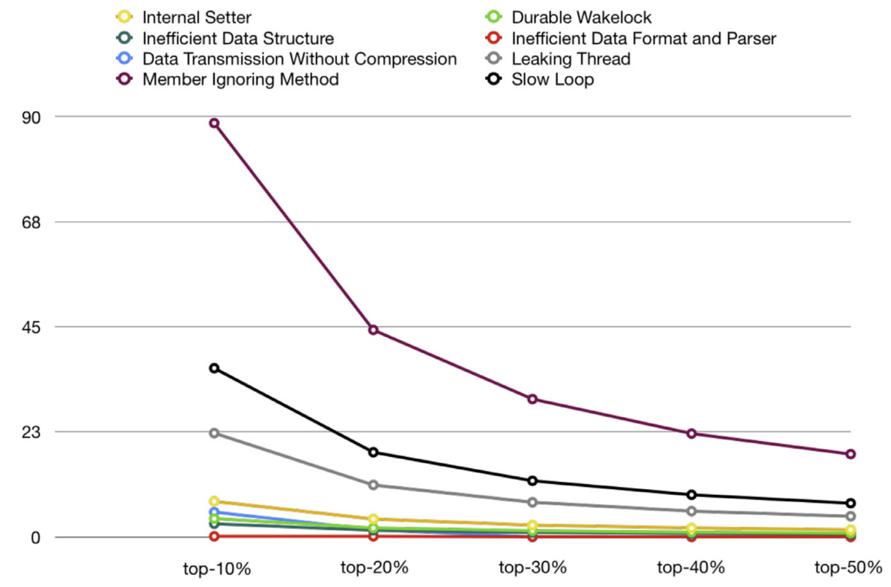

Fig. 3. Percentage of most consuming methods also affected by a code smell.

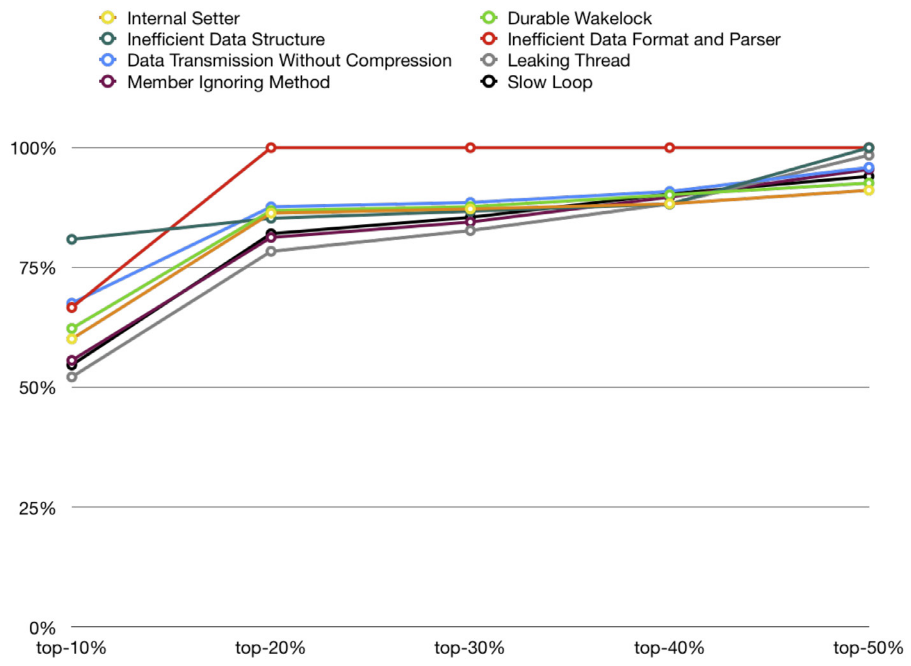

Fig. 4. Percentage of code smells appearing in the top-10%, top-20%, top-30%, top-40%, and top-50% most consuming smells.

Summary for RQ2. 94% of the most consuming methods in our dataset were affected by at least one code smell: this might indicate the existence of a strong relationship between the considered code smells and the energy consumption of methods.

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

The results of the previous research question pointed out that most of the analyzed code smells have a kind of relation with the energy consumption of the affected methods. However, this is not enough to show the actual impact of code smells. Our final research question is aimed at assessing the actual impact of code smells on energy consumption. To this aim, we evaluated to what extent the refactoring of the considered smells has an effect on the reduction of the energy consumption of the smelly methods. Specifically, we manually refactored the 2354 smelly methods affected by only one of the smells considered. Note that in this research question we have not considered the methods affected by more smells since we are interested in understanding the effect of the single refactoring operations on the energy consumption of such methods. The refactoring phase required approximatively 450 man-hours.

Table 2 reports the statistics on the energy consumption of the methods affected by *Data Transmission Without Compression, Durable Wakelock, Inefficient Data Structure*, and *Inefficient Data Format and Parser*, respectively, before and after the refactoring. As it is possible to observe, the refactoring operations applied do not have any effect on the resulting energy consumption. On the one hand, it is important to

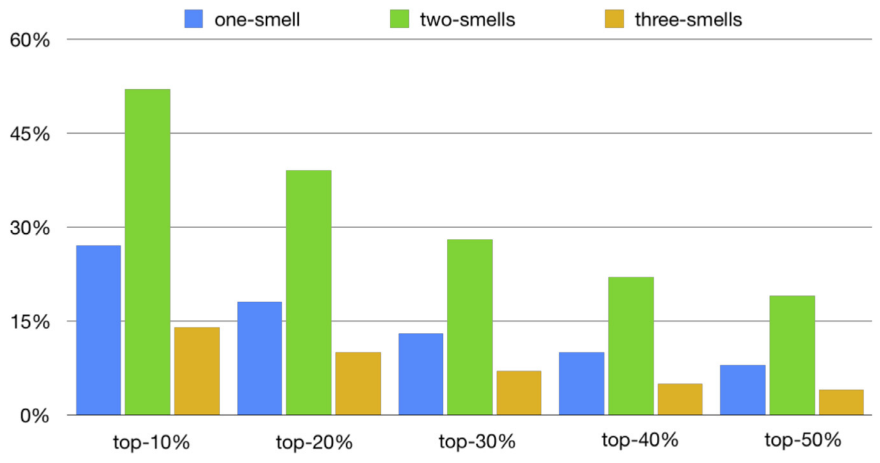

Fig. 5. Percentage of most consuming methods that are affected by one or more smells.

```
computeEnergyConsumption(apkLocation, appName, tCase, nRuns){
installApp(apkLocation);
for (run=0; run<nRuns; run++) {
clearAppCache(appName);
resetBatteryStats();
startProfiler();
exerciseApp(appName, tCase);
stopProfiler();
collectBatteryStatsData();
collectSysTraceData();
collectDMTraceDumpData();
loadPowerProfile();
for each method call in trace file {
computeCallEnergyConsumption();
}
saveResults();
stopApp(appName);
}
uninstallApp(apkLocation)
}
```

Listing 1. `PETrA` workflow.

Table 2
Energy consumed by methods before and after the application of refactorings. “R-” prefix stands for Refactored.

Smell Type	Min	1st Qu.	Median	Mean	3rd Qu.	Max
DTWC	0.004	0.006	0.008	0.010	0.013	0.018
R-DTWC	0.004	0.006	0.008	0.010	0.013	0.018
DWL	0.001	0.001	0.001	0.001	0.002	0.003
R-DWL	0.001	0.001	0.001	0.001	0.002	0.003
IDS	0.002	0.002	0.003	0.003	0.003	0.004
R-IDS	0.002	0.002	0.003	0.003	0.003	0.004
IDFAP	0.001	0.001	0.001	0.001	0.001	0.002
R-IDFAP	0.001	0.001	0.001	0.001	0.001	0.002

point out that our study was conducted at method-level, while the negative impact of certain code smell types might arise at a higher level of granularity. For instance, the high energy consumption of the *Inefficient Data Structure* smell was shown by Hasan et al. [7] when the workload of an `Hashmap<Integer, Object>` increases (*i.e.,*when many insertions or iterations are done over the data structure): this indicates that some smells may have a negative influence under stressing conditions observable at app-level, while others (those with the highest co-occurrences with the most energy consuming methods in our study) immediately impact energy consumption of methods. Further investigations on the role of code smells at different levels of granularity would be desirable. On the other hand, some previous studies (such as the one by Hasan et al. [7]) were conducted in the context of larger and more complex applications, like Java libraries: the low relation between certain smells and energy consumption observed in our study might also be due to the fact that we analyzed mobile apps, *i.e.,*significantly smaller software systems as compared to libraries. In this context, we observed that some types of issues are generally poorly diffused and, therefore, not particularly relevant. For instance, *Inefficient Data Structure* instances represent the 0.01% of the analyzed methods, while *Inefficient Data Format And Parser* only the 0.001%: this makes their impact notably marginal while reinforcing the idea to deeper investigate the effect of such smells at a higher level of granularity.

A similar discussion can be held with respect to the other code smells considered in the study: *Durable Wakelock* and *Data Transmission Without Compression.* Methods acquiring a wakelock without releasing it might be problematic from an energy perspective due to a long execution of an app, as well as methods transmitting a file over a network without compression may create energy leaks depending on the size of the transmitted file. In other words, some smells might be more prone to cause energy leaks in different situations of those explored in this paper. This further confirms the need for additional studies on the impact of code smells. At the same time, we believe that our results represent valuable findings since they actually show how the negative impact of some code smells (i) is not visible at method-level and (ii) should be assessed in different contexts.

At the same time, Table 3 reports the statistics on the energy

Table 3
Energy consumed by methods before and after the application of refactorings. "R-" prefix stands for Refactored.

Smell Type	Min	1st Qu.	Median	Mean	3rd Qu.	Max
IS	0.076	0.076	0.082	0.083	0.092	0.092
R-IS	0.001	0.001	0.009	0.016	0.024	0.024
LT	0.01	0.02	0.03	0.077	0.013	0.77
R-LT	0.001	0.001	0.003	0.009	0.009	0.019
MIM	0.001	0.002	0.004	0.04	0.028	0.981
R-MIM	0.0001	0.002	0.018	0.02	0.019	0.038
SL	0.001	0.006	0.0119	0.056	0.026	0.929
R-SL	0.001	0.004	0.0114	0.010	0.014	0.018

consumption of methods affected by *Internal Setter, Leaking Thread, Member Ignoring Method*, and *Slow Loop*, respectively, before and after the application of the corresponding refactoring operations. In all the comparisons between smelly and refactored versions a recurring pattern can be observed: when the code smells affecting the methods are removed, the energy consumption of such methods decrease. For methods affected by *Internal Setter* we can observe that the associated solution, namely the *Direct Field Access* refactoring [9], reverses the negative effect of the smell by reducing the energy consumption by almost 27 times with respect to the smelly methods (the median energy consumption of smelly methods is 0.082 J, while the median of the distribution of the refactored methods is 0.003 J). A representative example is the method `AcalDateTime.applyLocalTimeZone` of the `aCal` app[14], 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`. By directly using the fields reporting the actual zone and the current time, the method consumes, on average, 0.012 J, namely 7 times less than the non-refactored version (0.094 J). 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 0.080J when methods are affected by the smell, compared to 0.003J when the methods are refactored. Hence, the *Introduce Run Check Variable* refactoring [9] helps in reducing the energy consumption 26 times 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 **RQ2**. From an average of 0.770 J previously obtained, we observed that the energy consumed by the refactored version is 0.010 J, namely 77 times lower. The differences are statistically significant (p-value < 0.001) with a medium effect size ($d = 0.35$).

The *Introduce Static Method* refactoring [9] needed to remove the *Member Ignoring Method* smell turned out to be strongly impacting the energy efficiency of source code methods. On average, the energy consumption of methods changes from a mean of 0.070–0.008 J, leading to the definition of a method nine times more efficient. The differences are statistically significant (p-value < 0.001) with a medium effect size ($d = 0.46$). An example appeared in the `Alarm Klock` app[15]. 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. Once refactored, we observed a strong improvement of its energy efficiency (in the ten runs the consumption decreases from 0.870 J to 0.030 J, namely 29 times lower).

The discussion about the *Slow Loop* smell is similar to one of the other energy smells. The differences are statistically significant (p-value < 0.005), yet, with small effect size (d=0.12). As an example, the `onOptionsItemSelected` method, belonging to the `BlinkenlightsBattery` class of the `Battery Circle` app[16] is responsible for analyzing all the possible configuration options provided as input by the user, using a slow version of the `for` loop. When refactored, the method passed from a consumption of 0.870 J to 0.010 J (87 times less), *i.e.,*it completely changed its energy profile.

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 (*i.e.,*modifying external method calls in a way they call a static method). 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 4. 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 RQ3. 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 previously affected by *Internal Setter, Leaking Thread, Member Ignoring Method*, and *Slow Loop*, consume up to 87 times less than methods affected by smells. Based on these results, we observe that refactoring is a powerful activity that should be applied by mobile developers.

4. 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 the ADOCTOR tool to detect candidate code smell instances. We are aware that our results can be affected by the presence of false positives and false negatives. However, the performance of the tool has been evaluated on 18 apps considered in the study, finding that ADOCTOR has a precision of 98% and a recall of 98%. 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 [19]) are in line with those obtained in our paper, confirming all our findings.

On the other hand, we measured energy consumption using our tool PETRA. As briefly explained in Section 2, we empirically evaluated the accuracy of the approach on 54 mobile apps comparing the power estimation of the tool with the oracle provided by Linares–Vasquez [5]. The validation revealed that in 95% 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 [14]. Moreover, in case of differences between the estimations provided by PETRA and the hardware-based tool, these would be consistent for both methods affected by smells and refactored methods, so that the error would not invalidate our findings. We aggregated the results given by PETRA using the mean operator. As highlighted in Section 2, in our case, the mean can be considered significant because the energy consumption of each exercised method tends to remain similar over the ten runs and, therefore, the distribution of the

[14] http://tinyurl.com/8hg67fk
[15] http://tinyurl.com/ngzkv3v
[16] http://tinyurl.com/o33ms7d

Table 4
Average battery discharge of methods before and after refactored.

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

energy consumption of each method does not contain outliers. Despite this, to be more confident about our findings, we repeated the experiment by aggregating the energy consumption using the sum, *i.e.,*the final output was a unique value representing the sum of the energy consumption of the methods exercised during the ten runs. The results achieved from this analysis are available in our online appendix [19] and similar to those reported in Section 3. Furthermore, it is worth noting that to measure the consumption of each method we analyzed the energy consumed by the application within a certain time frame. Although this is clearly an approximation (*e.g.,*the presence of more threads running simultaneously might bias the measurements), the same procedure has been performed in several previous research papers [33–37], which, however, also include additional approximations due to tail energy usage (that arise when certain hardware components are optimistically kept active by the operating system, even during idle periods): in this sense, we believe that our measurement process is still more precise than previous work and thus we are confident about the results provided in this paper.

To execute the apps, we automatically generated test cases using Monkey. Despite the possible limitations of this tool, it is worth considering that a previous study [12] demonstrated that Monkey was one of the best alternatives on the set of apps considered in our empirical study, as it was the tool able to perform best in terms of coverage. Moreover, it is worth noting that this tool does not deal with preliminary authentication steps of an application like, for example, the login phase. Nevertheless, this did not represent a threat in our case since all the applications taken into account did not have any preliminary step to access the real functionalities. Thus, we could entirely test the involved apps. To collect the energy consumption for each method, we had to run the apps in debugging mode. It is possible that due to this process, some optimization phases were prevented. However, on the one hand, we followed a well-defined process already used in previous work [3,5,21,23] and on the other hand the official Android documentation does not provide any information on this topic. Despite this, we acknowledge the possible threat.

All the experiments were performed on an `LG Nexus 4`, equipped with Android 5.1.1. It is worth considering that this choice was guided by previous research in the field [3,5,21,23]. However, it is possible that different results could be achieved on different smartphones and Android versions.

We cannot exclude that in the relation between code smells and energy consumption other factors may have played a role: however, we strengthen the observations made in **RQ3** by explicitly performing a study on how refactoring of those smells affect the investigated phenomenon. Another observation is that some methods might tend to be more smelly because of application design, *e.g.,*some design constraints enforcing developers in introducing code smells. While it would be worth analyzing the extent to which code smell introduction is due to such design constraints, our work aims at targeting the Android-specific code smells that impact more on energy efficiency as well as the impact of refactoring on the resulting energy consumption.

To study the effect of refactoring on energy efficiency (**RQ3**), we manually refactored the source code. The procedure involved two of the authors who carefully followed the guidelines provided by Reimann et al. [9]. 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. While we cannot exclude imprecision and some degree of subjectiveness (mitigated by the discussion) in the way we refactor the source code, it is important to note that we re-executed the same test cases generated by Monkey to (i) double-check the validity of the refactoring operations applied and (ii) control that the external behavior of the refactored methods was not changed after the refactoring. As a result, all tests passed, confirming that the refactoring of code smells did not change the external behavior of the app.

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. Another aspect to discuss is related to the granularity of the measurements conducted, which might have influenced our observations. We worked at the method level as our goal was to assess the relation between code smells and energy consumption of methods: nevertheless, this does not exclude that other code smells at class- or app-level might still have an impact when considering the overall app consumption. For example, the *Inefficient Data Structure* might still be highly consuming in cases where several operations (*e.g.,*frequent insertions/iterations/removals of elements) are performed over a `Hashmap<Integer, Object>` [7]. Thus, it might be worth to spend future research effort on studying the impact of higher level code smells on the energy consumption.

Threats to *internal validity* concern factors that could influence our observations. We are aware that in principle we cannot claim a direct cause-effect relationship 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 (**RQ3**). 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 both smelly and non-smelly methods having different (i) size, (ii) complexity, and (iii) level of test coverage. We selected these three confounding factors for the following reasons: size can impact energy consumption since longer methods might execute more code, thus leading to consume more energy; more complex methods can have more complex programming constructs that might require more energy consumption; as we executed test cases, the level of coverage might impact the number of executed statements and, therefore, a higher/lower coverage might influence the energy consumed by a certain method. In details, we performed the following operations to control for the three confounding factors:

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 of 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 of 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 of the methods having a size larger than the third quartile of the distribution of all the method sizes, *i.e.,large* size;
2. We computed the energy consumption for each method belonging to the three groups, to investigate whether larger methods consume more than smaller methods.

The experiment has been repeated considering the McCabe cyclomatic complexity [38] and the coverage obtained by the test cases ran over the methods analyzed [39] as measures to split methods in *small, medium*, and *large* sets. We reported the achieved results in our online appendix [19], however, we observed that such characteristics are not strongly related with the energy consumption of the methods: specifically, only 13% of the largest methods are in the top-10% of the most consuming methods. Similarly, 11% of the most complex ones are in the top-10% and 6% of the methods having the highest coverage are on the list of the most consuming smells. These results indicate that such confounding factors do not have a relevant impact on energy efficiency. We are aware that other factors might have influenced our findings: for instance, methods belonging to `Activity` classes of Android apps might consume more by design, as they are called more frequently. Nevertheless, our main goal was to study the effect of code smells and the results obtained by refactoring smelly instances make us confident enough the causation between their removal and the improvement of energy efficiency of methods. However, further investigations on the role of the additional factors on energy consumption would be worthwhile.

Finally, regarding the generalization of our findings (*external validity*), we evaluated the impact of nine 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.

5. Related work

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

5.1. About code smells and refactoring

The traditional code smells defined by Fowler [10] have been widely studied in the past. Several studies demonstrated their negative effects on program comprehension [31], change- and fault-proneness [40,41], and maintainability [42,43]. At the same time, several approaches and tools, relying on different sources of information, have been proposed to automatically detect [44–47], and fix them through the application of refactoring operations [48,49].

Traditional code smells have also been studied in the context of mobile apps by Mannan et al. [50], who demonstrated that, despite the different nature of mobile applications, the variety and density of code smells is similar. In the same context, Morales et al. [51] studied the effect of several anti-patterns [10,52] with respect to the energy efficiency of mobile apps. They analyzed 59 Android apps and found that some refactorings have a positive effect on the energy efficiency of mobile apps, while applying others has a negative effect. Similarly, Gjoshevski and Schweighofer [53] analyzed 30 Android apps using 140 Lint rules showing that the size of the apps does not have an impact on technical debt. A more detailed overview about code smells and refactoring can be found in [54] and [55].

As for the Android-specific code smells defined by Reimann et al. [9], there is little knowledge about them. Indeed, while the authors of the catalog assumed the existence of a relationship between the presence of code smells and non-functional attributes of source code, they never empirically assessed it [9]. The unique investigation on the impact of three code smells on the performance of Android applications has been carried out by Hetch et al. [56], who found some positive correlations between the studied smells and the decreasing performance in term of delayed frames and CPU usage. Finally, Morales et al. [57] proposed EARMO, a refactoring tool that, besides code quality, takes into account the energy consumption when refactoring code smells detected in mobile apps. It is important to note that the authors mostly considered the code smells proposed by Fowler [10], while our work aims at understanding the impact of a large variety of Android-specific code smells on energy efficiency as well as the role of refactoring on the performance improvement of mobile apps.

5.2. 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 [21] 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 power estimations.

On top of estimation tools, researchers have studied ways to predict the energy consumption using empirical data [33,34] or dynamic analysis [35,36], 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. Procaccianti et al. [37] provided evidence on the beneficial effect of using *green* practices such as query optimization in MySQL Server and the usage of the `sleep` instruction in the Apache web server. Sahin et al. [8] studied the influence of code obfuscation on energy consumption, finding that the magnitude of such impacts is unlikely to impact end users. Sahin et al. [4] also studied the role of design patterns, highlighting that some patterns (*e.g.,*the *Decorator* pattern) negatively impact the energy efficiency of an application. Similar results have been found by Noureddine et al. [59].

Sahin et al. [6] studied the effect of the refactoring operations defined by Fowler [10] on energy consumption. Specifically, they evaluated the behavior of six 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. [60] 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 from the presence of design flaws. In contrast, our study analyzes the impact of code smells specifically defined for Mobile applications [9] on energy consumption as well as the influence of refactoring operations aimed at removing them from the source code.

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%. Along the same lines, other researchers focused their attention on the behavior of sorting algorithms [61], lock-free data structures [62], GUI optimizations [63], API usage of Android apps [5], providing findings on how to efficiently use different programming structures and algorithms.

Cruz et al. [64] performed an analysis of eight best practices for improving the energy consumed by Android apps. Through their empirical study on six popular apps, they observed that it is possible to extend the battery life of the device applying energy-aware practices. Our work is complementary to the one by Cruz et al. [64]: rather than analyzing the effect of energy-aware best practices, we studied the effect of refactoring code smells on energy consumption of methods.

Kim [65] introduced a complementary set of performance-enhancing best practices for Android programming with respect to those proposed by Reimann et al. [9]. He evaluated these practices on the CPU time of the apps showing that applying them it is possible to develop cpu-efficient mobile apps.

Li et al. [66] conducted a small-scale empirical investigation—involving four code snippets—into some programming practices believed to be energy-saving, such as the strategies for invoking methods, accessing fields, and setting the length of arrays. The results of this study confirmed the expectations, showing that such practices help in reducing the energy consumption of mobile apps. While some of the practices considered by Li et al. [66] are similar to the code smells we considered (*e.g.,*the way developers access fields), it is worth noting the set of analyzed programming practices is quite small. Moreover, their study did not quantify how much energy can be saved by removing the analyzed poor programming practices. Furthermore, they focused their attention only on a few code snippets, rather than considering a large variety of mobile apps.

Finally, the studies proposed by Hecht et al. [67] and Carette et al. [11] have the same purpose as the one proposed in this paper, since both are aimed at investigating the impact of three Android-specific code smells on the energy consumption of mobile apps. Specifically, Carette et al. [11] considered the behavior of the *Internal Getter/Setter, Inefficient Data Structure*, and *Member Ignoring Method* smells on a set of five mobile applications, and measured the energy consumption before and after their removal. Their results are strongly different from those reported by us. Indeed, the authors found that the removal of single code smells increase the energy efficiency up to 3.86%, while the correction of all the code smells can reduce the energy consumption of mobile apps by up to 4.83%. On the one hand, this is due to the fact that their results are affected by the presence of the *Inefficient Data Structure* smell, that in this paper we show to be poorly diffused in mobile apps. On the other hand, we exploited a much larger set of apps, being able to better characterize the behavior of the *Internal Setter* and *Member Ignoring Method* smells. Moreover, the voltmeter used in their experimentation to measure the energy consumption works at a frequency of only 10Hz; as demonstrated by Saborido et al. [68], such a frequency is too low to observe the actual consumption of methods, thus possibly producing unreliable results. In other words, our work is complementary with this work, considering that the former performs an analysis at app level granularity, while our analysis is at method level granularity. For this reason, it is possible that some effects of code smells on energy consumption are not visible in their evaluation, but are visible in ours, and vice versa.

Afterward, Habci et al. [69] extended the Paprika tool proposed by Hecht et al. [67] for analyzing the effect of code smells on iOS apps developed in Objective-C or Swift. The result of their empirical study shows that both the apps developed in Objective-C and Swift tend to contain the same proportions of code smells. However, iOS apps seem to be less code smells prone with respect to Android apps.

6. Conclusion

This paper presented a large-scale empirical investigation taking into account the role of nine *Android-specific* code smells [9] on the energy consumption of mobile apps. Starting from an analysis aimed at studying the relevance of each code smell type in the context of mobile apps (**RQ1**), we then further investigated the relation between code smells and energy consumption (**RQ2**). Finally, we evaluated whether refactoring operations applied to remove code smells help in substantially improving the efficiency of mobile applications (**RQ3**). The achieved results provide valuable findings and insights for the research community. Summarizing, the paper provided the following contributions:

1. A large-scale empirical study involving 60 Android apps aimed at assessing the extent to which nine method-level code smells impact energy efficiency and whether refactoring operations are able to fix energy leaks.
2. A comprehensive replication package [19], including all the raw data and scripts used for the empirical study.

The results achieved provide two main lessons for both the research community and tool vendors:

Lesson 1. *Some code smells have a strong impact on the energy efficiency of source code methods.* Specifically, methods affected by four particular smell types that frequently co-occur, *i.e.,Leaking Thread, Member Ignoring Method, Slow Loop*, and *Internal Setter*, have an energy consumption up to 87 times higher than other smelly methods. This result highlights 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 refactoring represents a powerful technique to reduce the energy consumption of methods. 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.,*Fowler's smells [10]) as well as code smell co-occurrences on energy efficiency of methods or apps. Furthermore, we plan to investigate whether certain smells have an effect on energy consumption when this is computed at a higher granularity (*e.g.,*overall app consumption). Finally, we plan to test the effect of refactoring in an *in-vivo* experiment with end-users, with the aim of assessing whether they can actually perceive the energy optimization of used apps after refactoring.

References

[1] A. Hindle, Green mining: a methodology of relating software change and configuration to power consumption, Empir. Softw. Eng. 20 (2) (2015) 374–409, https://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, R. Govindan, Estimating mobile application energy consumption using program analysis, Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 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, K. Winbladh, Initial explorations on design pattern energy usage, Proceedings of the First International Workshop on Green and Sustainable Software, GREENS '12, IEEE Press, Piscataway, NJ, USA, 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, D. Poshyvanyk, Mining energy-greedy api usage patterns in android apps: An empirical study, Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, ACM, New York, NY, USA, 2014, pp. 2–11, https://doi.org/10.1145/2597073.2597085.

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

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

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

[9] J. Reimann, M. Brylski, U. Amann, A Tool-Supported Quality Smell Catalogue for Android Developers, (2014).

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

[11] A. Carette, M.A.A. Younes, G. Hecht, N. Moha, R. Rouvoy, Investigating the energy impact of android smells, 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), (2017), pp. 115–126, https://doi.org/10.1109/SANER.2017.7884614.

[12] S.R. Choudhary, A. Gorla, A. Orso, Automated test input generation for android: are we there yet? Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 429–440, https://doi.org/10.1109/ASE.2015.89.

[13] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, Lightweight detection of android-specific code smells: the adoctor project, 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), (2017), pp. 487–491, https://doi.org/10.1109/SANER.2017.7884659.

[14] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, A. De Lucia, Software-based energy profiling of android apps: Simple, efficient and reliable? 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), (2017), pp. 103–114, https://doi.org/10.1109/SANER.2017.7884613.

[15] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, A.M. Memon, Using GUI ripping for automated testing of android applications, Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, ACM, New York, NY, USA, 2012, pp. 258–261, https://doi.org/10.1145/2351676.2351717.

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

[17] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: An input generation system for

android apps, Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 224–234, https://doi.org/10.1145/2491411.2491450.

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

[19] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, On the Impact of Code Smells on the Energy Consumption of Mobile Applications - Replication Package, 2017. https://dx.doi.org/10.6084/m9.gshare.3759489.

[20] Android, Android Lint Documentation. https://sites.google.com/a/android.com/tools/tips/lint-checks.

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

[22] M. Dong, L. Zhong, Self-constructive high-rate system energy modeling for battery-powered mobile systems, Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, ACM, 2011, pp. 335–348.

[23] D. Li, S. Hao, J. Gui, W.G.J. Halfond, An empirical study of the energy consumption of android applications, Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, (2014), pp. 121–130, https://doi.org/10.1109/ICSME.2014.34.

[24] K. Mao, M. Harman, Y. Jia, Sapienz: multi-objective automated testing for android applications, Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, ACM, New York, NY, USA, 2016, pp. 94–105, https://doi.org/10.1145/2931037.2931054.

[25] L. Al Shalabi, Z. Shaaban, Normalization as a preprocessing engine for data mining and the approach of preference matrix, Dependability of Computer Systems, 2006. DepCos-RELCOMEX'06. International Conference on, IEEE, 2006, pp. 207–214.

[26] S.S. Shapiro, M.B. Wilk, An analysis of variance test for normality (complete samples), Biometrika 52 (3/4) (1965) 591–611.

[27] W.J. Conover, Practical Nonparametric Statistics, third ed., Wiley, 1998.

[28] R.J. Grissom, J.J. Kim, Effect Sizes for Research: A Broad Practical Approach, second ed., Lawrence Earlbaum Associates, 2005.

[29] R.H. Romer, Energy : An Introduction to Physics, Freeman, San Francisco, 1976.

[30] A.F. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects? 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23–28, 2012, IEEE Computer Society, 2012, pp. 306–315.

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

[32] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, A large-scale empirical study on the lifecycle of code smell co-occurrences, Inf. Softw. Technol. 99 (2018) 1–10.

[33] N. Amsel, B. Tomlinson, Green tracker: a tool for estimating the energy consumption of software, CHI '10 Extended Abstracts on Human Factors in Computing Systems, CHI EA '10, ACM, New York, NY, USA, 2010, pp. 3337–3342, https://doi.org/10.1145/1753846.1753981.

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

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

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

[37] G. Procaccianti, H. Fernández, P. Lago, Empirical evaluation of two best practices for energy-efficient software development, J. Syst. Softw. 117 (C) (2016) 185–198, https://doi.org/10.1016/j.jss.2016.02.035.

[38] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320, https://doi.org/10.1109/TSE.1976.233837.

[39] J.C. Miller, C.J. Maloney, Systematic mistake analysis of digital computer programs, Commun. ACM 6 (2) (1963) 58–63, https://doi.org/10.1145/366246.366248.

[40] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empir. Softw. Eng. 17 (3) (2012) 243–275.

[41] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Empir. Softw. Eng. (2017) 1–34.

[42] D.I.K. Sjøberg, A.F. Yamashita, B.C.D. Anda, A. Mockus, T. Dybå, Quantifying the effect of code smells on maintenance effort, IEEE Trans. Softw. Eng. 39 (8) (2013) 1144–1156.

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

[44] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F.L. Meur, Decor: a method for the specification and detection of code and design smells, IEEE Trans. Softw. Eng. 36 (2010) 20–36.

[45] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, Mining version histories for detecting code smells, Softw. Eng. IEEE Trans. 41 (5) (2015) 462–489, https://doi.org/10.1109/TSE.2014.2372760.

[46] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, A. Zaidman, A textual-based technique for smell detection, 2016 IEEE 24th International Conference on Program Comprehension (ICPC), (2016), pp. 1–10, https://doi.org/10.1109/ICPC.2016.7503704.

[47] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, A. De Lucia, The scent of a smell: an extensive comparison between textual and structural smells, IEEE Trans. Softw. Eng.. *To Appear*.

[48] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367.

[49] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, Methodbook: recommending move method refactorings via relational topic models, IEEE Trans. Softw. Eng. 40 (7) (2014) 671–694, https://doi.org/10.1109/TSE.2013.60.

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

[51] R. Morales, R. Saborido, F. Khomh, F. Chicano, G. Antoniol, Anti-patterns and the energy efficiency of android applications, arXiv preprint arXiv:1610.05711 (2016).

[52] G. Hecht, N. Moha, R. Rouvoy, An empirical study of the performance impacts of android code smells, Proceedings of the International Conference on Mobile Software Engineering and Systems, ACM, 2016, pp. 59–69.

[53] M. Gjoshevski, T. Schweighofer, Small scale analysis of source code quality with regard to native android mobile applications. SQAMIA, (2015), pp. 9–16.

[54] F. Palomba, A. De Lucia, G. Bavota, R. Oliveto, Anti-pattern detection: methods, challenges, and open issues, Adv. Comput. 95 (2015) 201–238.

[55] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, Recommending Refactoring Operations in Large Software Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 387–419. 10.1007/978-3-642-45135-5_15.

[56] G. Hecht, N. Moha, R. Rouvoy, An empirical study of the performance impacts of android code smells, Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, ACM, New York, NY, USA, 2016, pp. 59–69, https://doi.org/10.1145/2897073.2897100.

[57] R. Morales, R. Saborido, F. Khomh, F. Chicano, G. Antoniol, Earmo: an energy-aware refactoring approach for mobile apps, IEEE Trans. Softw. Eng. (2017).

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

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

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

[61] C. Bunse, H. HÃ¶pfner, E. Mansour, S. Roychoudhury, Exploring the energy consumption of data sorting algorithms in embedded and mobile environments, 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware, (2009), pp. 600–607, https://doi.org/10.1109/MDM.2009.103.

[62] N. Hunt, P.S. Sandhu, L. Ceze, Characterizing the performance and energy efficiency of lock-free data structures, 2011 15th Workshop on Interaction between Compilers and Computer Architectures, (2011), pp. 63–70, https://doi.org/10.1109/INTERACT.2011.13.

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

[64] L. Cruz, R. Abreu, Performance-based guidelines for energy efficient mobile applications, Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, IEEE Press, 2017, pp. 46–57.

[65] D.K. Kim, Towards performance-enhancing programming for android application development. Int. J. Contents 13 (4) (2017).

[66] D. Li, W.G.J. Halfond, An investigation into energy-saving programming practices for android smartphone app development, Proceedings of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014, ACM, New York, NY, USA, 2014, pp. 46–53, https://doi.org/10.1145/2593743.2593750.

[67] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, L. Duchien, Tracking the software quality of android applications along their evolution (t), Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, 2015, pp. 236–247.

[68] R. Saborido, V. Arnaoudova, G. Beltrame, F. Khomh, G. Antoniol, On the Impact of Sampling Frequency on Software Energy Measurements, Technical Report, PeerJ PrePrints, 2015.

[69] S. Habchi, G. Hecht, R. Rouvoy, N. Moha, Code smells in iOS apps: how do they compare to android? Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, IEEE Press, 2017, pp. 110–121.