EARMO: An Energy-Aware Refactoring Approach for Mobile Apps

Journal:	*Transactions on Software Engineering*
Manuscript ID	Draft
Manuscript Type:	Journal First
Keywords:	K.6.3.b Software maintenance < K.6.3 Software Management < K.6 Management of Computing and Information Systems < K Computing Mil, J.9 Mobile Applications < J Computer Applications, code design, I.2.m.c Evolutionary computing and genetic algorithms < I.2.m Miscellaneous < I.2 Artificial Intelligence < I Computing Methodol

EARMO: An Energy-Aware Refactoring Approach for Mobile Apps

Rodrigo Morales, *Member, IEEE,* Rubén Saborido, *Member, IEEE,* Foutse Khomh, *Member, IEEE,* Francisco Chicano, *Member, IEEE,* and Giuliano Antoniol, *Senior Member, IEEE*

Abstract—The boom in mobile apps has changed the traditional landscape of software development by introducing new challenges due to the limited resources of mobile devices, e.g., memory, CPU, network bandwidth and battery. The energy consumption of mobile apps is nowadays a hot topic and researchers are actively investigating the role of coding practices on energy efficiency. Recent studies suggest that design quality can conflict with energy efficiency. Therefore, it is important to take into account energy efficiency when evolving the design of a mobile app. The research community has proposed approaches to detect and remove anti-patterns (i.e., poor solutions to design and implementation problems) in software systems but, to the best of our knowledge, none of these approaches have included anti-patterns that are specific to mobile apps and–or considered the energy efficiency of apps. In this paper, we fill this gap in the literature by analyzing the impact of eight type of anti-patterns on a testbed of 20 android apps extracted from F-Droid. First, we (1) analyze the impact of anti-patterns in mobile apps with respect to energy efficiency; then (2) we study the impact of different types of anti-patterns on energy efficiency; and (3) propose EARMO, a novel anti-pattern correction approach that accounts for energy efficiency when refactoring mobile anti-patterns. Next, (4) we evaluate EARMO using three multiobjective search-based algorithms. Our obtained results show that EARMO is able to remove a median of 84% of anti-patterns, while saving up to 34% of energy. Finally, (5) we conducted a qualitative study with developers of our studied apps, to assess the refactoring recommendations made by EARMO. Developers found 68% of refactorings suggested by EARMO to be very relevant.

Index Terms—Software maintenance; Refactoring; Anti-patterns; Mobile apps; Energy consumption; Search-based Software Engineering

1 Introduction

During the last five years, and with the exponential growth of the market of mobile apps [1], software engineers have witnessed a radical change in the landscape of software development. From a design point of view, new challenges have been introduced in the development of mobile apps such as the constraints related to internal resources, *e.g.,* CPU, memory, and battery; as well as external resources, *e.g.,* internet access. Moreover, traditional desired quality attributes, such as functionality and reliability, have been overshadowed by subjective visual attributes, *i.e.,* "flashiness" [2].

Mobile apps play a central role in our life today. We use them almost anywhere, at any time and for everything; *e.g.,* to check our emails, to browse the Internet, and even to access critical services such as banking and health monitoring. Hence, their reliability and quality is critical. Similar to traditional desktop applications, mobile apps age as a consequence of changes in their functionality, bug-fixing, and introduction of new features, which sometimes lead to the deterioration of the initial design [3]. This phenomenon known as *software decay* [4] is manifested in the form of design flaws or anti-patterns. An example of anti-pattern is the *Lazy class*, which occurs when a class does too little, *i.e.,* has few responsibilities in an app. A *Lazy class* typically is comprised of methods with low complexity and is the result of speculation in the design and-or implementation stage. Another common anti-pattern is the *Blob, a.k.a., God class*, which is a large and complex class that centralizes most of the responsibilities of an app, while using the rest of the classes merely as data holders. A *Blob class* has low cohesion, and hinders software maintenance, making code hard to reuse and understand. Resource management is critical for mobile apps. Developers should avoid anti-patterns that cause battery drain. An example of such anti-pattern is *Binding resources too early class* [5]. This anti-pattern occurs when a class *switches on* energy-intensive components of a mobile device (*e.g.,* Wi-fi, GPS) when they cannot interact with the user. Another example is the use of *private getters and setters* to access class attributes in a class, instead of accessing directly the attributes. The Android documentation [6] strongly recommends to avoid this anti-pattern as virtual method calls are up to seven times more expensive than using direct field access [6].

Previous studies have pointed out the negative impact of anti-patterns on change-proneness [7], fault-proneness [8], and maintenance effort [9]. In the context of mobile apps, Hecht et al. [10] found that anti-patterns are prevalent along the evolution of mobile apps. They also confirmed the observation made by Chatzigeorgiou and Manakos [11] that anti-patterns tend to remain in systems through several releases, unless a major change is performed on the system.

Recently, researchers and practitioners have proposed approaches and tools to detect [12], [13] and correct [14] anti-patterns. However, these approaches only focus on object-oriented anti-patterns and do not consider mobile

- *R. Morales, R. Saborido, F. Khomh, and G. Antoniol are with Polytechnique Montéal, Québec, Canada. E-mail: {rodrigo.morales, ruben.saborido-infantes, foutse.khomh}@polymtl.ca, antoniol@ieee.org.*
- *F. Chicano is with University of Málaga, Málaga, Spain. E-mail: chicano@uma.es.*

Manuscript received September 2016.

development concerns. One critical concern of mobile apps development is improving energy efficiency, due to the short life-time of mobile device's batteries. Some research studies have shown that behavior-preserving code transformations (*i.e., refactorings*) that are applied to remove anti-patterns can impact the energy efficiency of a program [15], [16], [17]. Hecht et al. [18] observed an improvement in the user interface and memory performance of mobile apps when correcting Android anti-patterns, like *private getters and setters*, *HashMap usage* and *member ignoring method*, confirming the need of refactoring approaches that support mobile app developers.

Despite these works on anti-patterns and energy consumption, to the best of our knowledge, there is no refactoring approach to support mobile developers towards the correction of anti-patterns. In this paper, we aim to fill this gap by proposing an automated refactoring approach that can remove anti-patterns in mobile apps while controlling for energy efficiency. To achieve this goal, we first study the impact of eight well-known Object-oriented (OO) and Android specific (extracted from Android Performance guidelines [6]) anti-patterns on energy efficiency. We use a testbed of 20 open-source android apps extracted from the F-Droid marketplace, an Android app repository. After analyzing the impact on energy efficiency of anti-patterns, we propose a novel multiobjective refactoring approach called *EARMO* (Energy-Aware Refactoring approach for MObile apps), that leverages information about the energy cost of anti-patterns, to detect and correct anti-patterns in mobile apps, while improving energy efficiency.

To assess the effectiveness of EARMO, we implement it using three different metaheuristics and apply it to our testbed of Android apps. Results show that EARMO can remove a median of 84% anti-patterns, while saving up to 34% of energy in the resulting app. We also conduct a qualitative study with developers of the studied apps, to assess the refactoring recommendations made by EARMO. These developers found 68% of refactorings suggested by EARMO to be very relevant and accepted our refactored code.

The primary contributions of this work can be summarized as follows:

1) We perform an empirical study of the impact of anti-patterns on the energy efficiency of mobile apps. We also propose a methodology for a correct measurement of the energy consumption of mobile apps, and compare it with a state-of-the-art approach. Our obtained results provide evidence to support the claim that developer's design choices can improve/decrease the energy efficiency of mobile apps.
2) The paper presents a novel automated refactoring approach to improve the design quality of mobile apps, while controlling for energy efficiency. The proposed technique provides developers the best trade-off between these two conflicted objectives, design quality and energy.
3) We evaluate the effectiveness of the proposed approach using three different multiobjective metaheuristics.
4) We perform a manual evaluation of the refactoring recommendations of EARMO for 5 randomly-selected apps, by selecting the solutions with the highest energy performance from the Pareto Front. Then, for each app we implemented the recommendations of EARMO, and build a new version to compare their energy efficiency with the original ones. We report energy efficiency improvements in all cases.
5) We evaluate the usefulness of the solutions proposed by EARMO from the perspective of mobile developers through a qualitative study.
6) We perform the evaluation of the design quality of the refactored apps using a widely-used Quality Model (QMOOD) [19].

The remainder of this paper is organized as follows: Section 2 provides some background information on refactoring, energy measurement of mobile apps, and multiobjective optimization. Section 3 presents a preliminary study regarding the impact of anti-patterns on energy efficiency. In Section 4, we present our automated approach for refactoring mobile apps while Section 5 describes the experimental setting for evaluating the proposed approach and present and discuss the results obtained from our experiments. In Section 6, we discuss the threats to the validity of our study, while in Section 7 we relate our work to the state of the art. Finally, we present our conclusions and highlight directions for future work in Section 8.

2 Background

This section presents an overview of the main concepts used in this paper.

2.1 Refactoring

Refactoring, a software maintenance activity that transforms the structure of a code without altering its behavior [20], is widely used by software maintainers to counteract the effects of design decay due to the continuous addition of new functionalities or the introduction of poor design choices, *i.e.,* anti-patterns, in the past [3]. The process of refactoring requires the identification of places where code should be refactored (*e.g.,* anti-patterns). Developers also have to determine which kind of refactoring operations can be applied to the identified locations. This step is cumbersome, as different anti-patterns can have different impact on the software design. Moreover, some refactoring operations can be conflicting, hence, finding the best combination of refactorings is not a trivial task. More formally, if k is the number of available refactorings, then, the number of possible solutions (NS) is given by $NS = (k!)^k$ [21], which results in a large space of possible solutions to be explored exhaustively. Therefore, researchers have reformulated the problem of automated-refactoring as a combinatorial optimization problem and proposed different techniques to solve it. The techniques range from single-objective approaches using local-search metaheuristics, *e.g.,* hill climbing, and simulated annealing [22], [23], to evolutionary techniques like genetic algorithm, and multiobjective approaches: *e.g.,* NSGA-II and MOGA [21], [24], [25], [26], MOCell, NSGA-II and SPEA2 [27].

Recent works [16], [28] have provided empirical evidence that software design plays also an important role

in the energy consumption of mobile devices; *i.e.,* high-level design decisions during development and maintenance tasks impact the energy efficiency of mobile apps. More specifically, these research works have studied the effect of applying refactorings to a set of software systems; comparing the energy difference between the original and refactored code.

In this research, we propose an automated-refactoring approach for refactoring mobile apps while controlling for energy efficiency. We target two categories of anti-patterns: (i) anti-patterns that stem from common Object-oriented design pitfalls [29], [30] (*i.e.,* Blob, Lazy Class, Long-parameter list, Refused Bequest, and Speculative Generality) and (ii) anti-patterns that affect resource usages as discussed by Gottschalk [28] and in the Android documentation [6], [28] (*i.e.,* Binding Resources too early, HashMap usage, and Private getters and setters). We believe that these anti-patterns occur often and could impact the energy efficiency of mobile apps. In the following subsections, we explain how we measure and include energy consumption in our proposed approach.

2.2 Energy measurement of mobile apps

Energy efficiency, a critical concern for mobile and embedded devices, has been typically targeted from the point of view of hardware and lower-architecture layers by the research community. Energy is defined as the capacity of doing work while power is the rate of doing work or the rate of using energy. In our case, the amount of total energy used by a device within a period of time is the energy consumption. *Energy* (*E*) is measured in *joules* (*J*) while *power* (*P*) is measured in *watts* (*W*). Energy is equal to power times the time period T in seconds. Therefore, $E = P \cdot T$. For instance, if a task uses two watts of power for five seconds it consumes 10 joules of energy.

One of the most used energy hardware profilers is the *Monsoon Power Monitor*[1]. It provides a power measurement solution for any single lithium (Li) powered mobile device rated at 4.5 volts (maximum three amps) or lower. It samples the energy consumption of the connected device at a frequency of $5\,kHz$, therefore a measure is taken each 0.2 milliseconds.

In this work energy consumption is measured using a more precise environment. Specifically we use a digital oscilloscope *TiePie Handyscope HS5* which offers the *LibTiePie SDK*, a cross platform library for using *TiePie* engineering USB oscilloscopes through third party software. We use this device because it allows to measure using higher frequencies than the *Monsoon Power Monitor*. The mobile phone is powered by a power supply and, between both, we connect, in series, a *uCurrent*[2] device, which is a precision current adapter for multimeters converting the input current in a proportional output voltage (V_{out}). The input current (I) is calculated by the *uCurrent* device and, therefore, $I = V_{out}$. Knowing I and the voltage supplied by the power supply (V_{sup}), we use the *Ohm's Law* to calculate the power usage (P) as $P = V_{sup} \cdot I$. The resolution is set up to 16 bits and the frequency to $125\,kHz$, therefore a measure is taken each eight microseconds. We calculate the energy associated to each sample as $E = P \cdot T = P \cdot (8 \cdot 10^{-6})s$. Where P is the power of the smart-phone and T is the period sampling in seconds. The total energy consumption is the sum of the energy associated to each sample.

In our experiments, we used a LG Nexus 4 Android phone equipped with a quad-core CPU, a 4.7-inch screen and running the Android Lollipop operating system (version 5.1.1, Build number LMY47V). We believe that this phone is a good representative of the current generation of Android mobile phones because more than three million have been sold since its release in 2013[3], and the latest version of Android studio includes a virtual device image of it for debugging.

We connect the phone to an external power supplier which is connected to the phone's motherboard, thus we avoid any kind of interference with the phone battery in our measurements. The diagram of the connection is shown in Fig. 1. Note that although we use an external power supplier, the battery has to be connected to the phone to work. Hence, we do not connect the positive pole of the battery with the phone.

To transfer and receive data from the phone to the computer, we use a USB cable, and to avoid interference in our measurements as a result of the USB charging function, we wrote an application to disable it. This application is free and it is available for download in the *Play Store*[4].

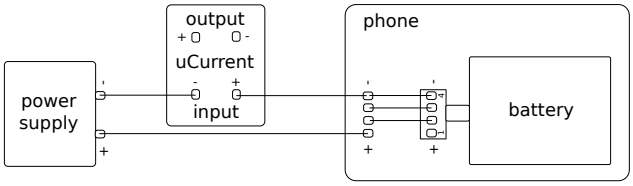

Figure 1: Connection between power supply and the Nexus 4 phone.

2.3 Multiobjective optimization

Optimization problems with more than one objective do not have single solutions because the objectives are usually in conflict. Consequently, the goal is to find solutions that represent a good compromise between all objectives without degrading any of them. These solutions are called non-dominated, in the sense that there are no solutions which are better with respect to one of the objective functions without achieving a worse value in at least another one.

More formally, let y_1 and y_2 be two solutions, for a multiobjective maximization problem, and $f_i, i \in 1 \ldots n$ the set of objectives. The solution y_1 dominates y_2 if: $\forall i, f_i(y_2) \leq f_i(y_1)$, and $\exists j | f_j(y_2) < f_j(y_1)$.

The set of all non-dominated solutions is called the Pareto Front. Very often, the search of the Pareto Front is NP-hard [31], hence researchers focus on finding an approximation set or reference front (RF) as close as possible to the Pareto Front.

1. https://www.msoon.com/LabEquipment/PowerMonitor/
2. http://www.eevblog.com/projects/ucurrent/
3. https://goo.gl/6guUpf
4. https://goo.gl/wyUcdD

As our aim is to improve the design quality of mobile apps, while controlling for energy efficiency, we consider each one of these criteria as a separate objective to fulfill.

In this work we use Evolutionary Multiobjective Optimization (EMO) algorithms, a family of metaheuristics techniques that are known to perform well handling multi-objective optimization problems [32]. To assess the effectiveness of our proposed automated-refactoring approach, we conduct a case study with three different EMO algorithms and compare their results in terms of performance, using two well-known performance indicators, to provide developers with information about the benefits and limitations of these different alternatives. In the following, we describe the metaheuristics techniques used in this paper, and in Section 4 we explain how we adapt them to find the best compromise between design quality and energy efficiency dimensions.

The *Non-dominated sorting genetic algorithm* (NSGA-II) [33] proceeds by evolving a new population from an initial population, applying variation operators like crossover and mutation. Then, it merges the candidate solutions from both populations and sort them according to their rank, extracting the best candidates to create the next generation. If there is a conflict when selecting individuals with the same ranking, the conflict is solved using a measure of density in the neighborhood, *a.k.a.*, crowding distance.

The *Strength Pareto Evolutionary Algorithm 2* (SPEA2) [34] uses variation operators to evolve a population, like NSGAII, but with the addition of an *external archive*. The archive is a set of non-dominated solutions, and it is updated during the iteration process to maintain the characteristics of the non-dominated front. In SPEA2, each solution is assigned a fitness value that is the sum of its strength fitness plus a density estimation.

The *Multiobjective Cellular Genetic Algorithm (MOCell)* is a cellular algorithm [35], that includes an external archive like SPEA2 to store the non-dominated solutions found during the search process. It uses the crowding distance of NSGA-II to maintain the diversity in the Pareto front. Note that the version used in this paper is an *asynchronous* version of MOCell called aMOCell4 [36]. The selection consists in taking individuals from the neighborhood of the current solution (cells) and selecting another one randomly from the archive. After applying the variation operators, the new offspring is compared with the current solution and replaces the current solution if both are non-dominated, otherwise the worst individual in the neighborhood will be replaced by the offspring.

3 Preliminary Study

The main goal of this paper is to propose a novel approach to improve the design of mobile applications while controlling for energy efficiency. To achieve this goal, the first step is to measure the impact of anti-patterns (*i.e.*, poor design choices) on energy efficiency. Understanding if anti-patterns affect the energy efficiency of mobile apps is important for researchers and practitioners interested in improving the design of apps through refactoring. Specifically, if anti-patterns do not significantly impact energy consumption, then it is not necessary to control for energy efficiency during a refactoring process. Whereas, if anti-patterns significantly affect energy consumption, developers and practitioners should be equipped with refactoring approaches that control for energy efficiency during the refactoring process, in order to prevent a deterioration of the energy efficiency of apps.

We formulate the research questions of this preliminary study as follows:

(PQ1) What is the relation between anti-patterns and energy efficiency?
The rationale behind this question is to determine if the energy efficiency of mobile apps with anti-patterns differs from the energy efficiency of apps without anti-patterns. We test the following null hypothesis: H_{0_1}*: there is no difference between the energy efficiency of apps containing anti-patterns and apps without anti-patterns*.

(PQ2) What is the relation between anti-pattern types and energy efficiency?
In this research question, we analyze whether certain types of anti-patterns lead to more energy consumption than others. We test the following null hypothesis: H_{0_2}*: there is no difference between the energy efficiency of apps containing different types of anti-patterns.*

3.1 Design of the Preliminary Study

As mentioned earlier, we consider two categories of anti-patterns: (i) *Object-oriented (OO)* anti-patterns [29], [30], and (ii) *Android anti-patterns (AA)* defined by [6], [28]. Table 1 presents the details of these anti-patterns. We select these anti-patterns because they have been found in mobile apps [10], [18], and they are well defined in the literature with recommended steps to remove them [6], [28], [29], [30].

Some of the refactorings applied to remove the aforementioned anti-patterns have been previously evaluated in terms of energy consumption using software estimation approaches. For example, *Binding resources too early* was evaluated by Gottschalk [28] and Park et al. [16] evaluated the refactorings proposed by Fowler. For Android anti-patterns like *HashMap usage*, and *private getters and setters*, there is no energy-consumption evaluation that we are aware of, however, they have been reported to decrease memory performance in previous works [18]. We believe that these anti-patterns occur often in mobile apps and could impact their energy efficiency.

To study the impact of the anti-patterns, we randomly downloaded 59 android apps from F-droid, an open-source Android app repository[7]. These apps come from five different categories (Games, Science and Education, Sports and health, Navigation, and Multimedia). To select the apps used in our study, we set the following criteria: more than one class, with at least one instance of any of the anti-patterns studied. Because we physically measure the energy consumption of the apps on a real phone, we validate that the candidate app compiles and run in the phone employed in this study. After discarding the apps that do not respect

5. https://developer.android.com/reference/android/support/v4/util/ArrayMap.html
6. https://source.android.com/devices/tech/dalvik/
7. https://f-droid.org/

Table 1: List of studied Anti-patterns.

Name	Description	Refactoring(s) strategy
Object-oriented anti-patterns		
Blob (BL) [29]	A large class that absorbs most of the functionality of the system with very low cohesion between its constituents.	*Move method (MM)*. Move the methods that does not seem to fit in the Blob class abstraction to more appropriate classes [23].
Lazy Class (LC) [30]	Small classes with low complexity that do not justify their existence in the system.	*Inline class (IC)*. Move the attributes and methods of the LC to another class in the system.
Long-parameter list (LP) [30]	A class with one or more methods having a long list of parameters.	*Introduce parameter object (IPO)*. Extract a new class with the long list of parameters and replace the method signature.
Refused Bequest (RB) [30]	A subclass uses only a very limited functionality of the parent class.	*Replace inheritance with delegation (RIWD)*. Remove the inheritance from the RB class and replace it with delegation through using an object instance of the parent class.
Speculative Generality (SG) [30]	There is an abstract class created to anticipate further features, but it is only extended by one class adding extra complexity to the design.	*Collapse hierarchy (CH)*. Move the attributes and methods of the child class to the parent and remove the *abstract* modifier.
Mobile anti-patterns		
Binding Resources too early (BE) [28]	Refers to the initialization of high-energy-consumption components of the device, *e.g.*, GPS, Wi-Fi before they can be used.	*Move resource request to visible method (MRM)*. Move the method calls that initialize the devices to a suitable Android event. For example, move method call for `requestlocationUpdates`, which starts GPS device, after the device is visible to the app/user (`OnResume` method).
HashMap usage (HMU) [18]	From API 19, Android platform provides *ArrayMap*[5] which is an enhanced version of the standard *Java HashMap* data structure in terms of memory usage. According to Android documentation, it can effectively reduce the growth of the size of these arrays when used in maps holding up to hundreds of items.	*Replace HashMap with ArrayMap (RHA)*. Import ArrayMap and replace HashMap declarations with ArrayMap data structure.
Private getters and setters (PGS) [6], [18]	Refers to the use of private getters and setters to access a field inside a class decreasing the performance of the app because of simple inlining of Android virtual machine [6] that translates this call to a virtual method called, which is up to seven times slower than direct field access.	*Inline private getters and setters (IGS)*. Inline the private methods and replace the method calls with direct field access.

the selection criteria, we end-up with a dataset of 20 apps. Table 2 shows the selected apps.

3.2 Data Extraction

The data extraction process is comprised of the following steps, which are summarized in Fig. 2.

1) **Extraction of android apps.** We wrote a script to download the apps from *F-droid* repository. This script provides us with the name of the app, the link to the source code, Android API version, and the number of Java files. We use the API version to discriminate apps that are not compatible with our phone, and the number of java files to filter apps with only one class. After filtering the apps, we import the source code in Eclipse (for the older versions) or Android Studio and ensure that they can be compiled and executed.
2) **Detection of anti-patterns and refactoring candidates**. The detection and generation of refactoring candidates is performed using our previous automated approach *ReCon* [37]. We use ReCon's current implementation of object-oriented anti-patterns and add two new OO anti-patterns (*Blob* and *Refused bequest*); we also add four Android anti-patterns based on the guidelines defined by Gottschalk [28], and the *Android documentation* [6]. ReCon supports two modes, root-canal- and floss-refactoring. We use the root-canal mode as we are interested in improving the complete design of the studied apps.
3) **Generation of scenarios**. For each app we define a typical usage scenario interacting with each application under study using the Android application *HiroMacro*[8]. This software allows us to generate scripts containing touch and move events, imitating a real user interacting with the app on the phone, to be executed several times without introducing variations in execution time due to user fatigue, or skillfulness. To automatize the measurement of the studied apps we convert the defined scenarios (*HiroMacro* scripts) to *Monkeyrunner* format. Thus, the collected actions can be played automatically from a script using the *Monkeyrunner*[9] Android tool.
4) **Refactoring of mobile apps**.
We use *Android Studio* and Eclipse refactoring-tool-support for applying the refactorings suggested by *ReCon*. For the cases where there is no tool support, we applied the refactorings manually into the source code. Currently, there is no tool support for refactoring *Binding resources too early* and *Hashmap usage*. To be sure that the refactored code is executed in the scenario, we set breakpoints and validate that the debugger stops on it. We also check that the refactored methods appeared in the execution trace. To activate the generation of execution trace file, we use the methods provided in *Android Debug Class*[10], for both original and refactored versions. The trace file contains information about all the methods executed with respect to time, that we use in the next step.
5) **Measurement of energy consumption**. As we mention in Section 2, we measure energy consumption of mobile apps using a precise digital oscilloscope *TiePie Handyscope HS5* which allows us to measure using high frequencies.
In our experiments each app is run 30 times to get median results and, for each run, the app is uninstalled after its usage and the cache is cleaned. A description of the followed steps is given in Algorithm 1, which has been implemented as a *python* script. As it is described, all apps are executed before a new run is started. Thus, we aim to avoid that cache memory on the phone stores information related to the app run that can cause to run faster after some executions. In addition, before the experiments, the screen brightness is set to the

8. https://play.google.com/store/apps/details?id=com.prohiro.macro
9. http://developer.android.com/tools/help/monkeyrunner_concepts.html
10. https://developer.android.com/reference/android/os/Debug.html

Table 2: Apps used to conduct the preliminary study.

App	Version	LOC	Category	Description
blackjacktrainer	0.1	3783	Games	Learning BlackJack
calculator	5.1.1	13985	Science & Education	Make calculations
gltron	1.1.2	12074	Games	3D lightbike racing game
kindmind	1.0.0	6555	Sports & Health	Be aware of sad feelings and unmet needs
matrixcalc	1.5	2416	Science & Education	Matrix calculator
monsterhunter	1.0.4	27368	Games	Reference for Monster Hunter 3 game
mylocation	1.2.1	1146	Navigation	Share your location
oddscalculator	1.2	2226	Games	Bulgarian card game odds calculator
prism	1.2	4277	Science & Education	Demonstrates the basics of ray diagrams
quicksnap	1.0.1	18487	Multimedia	Basic camera app
SASAbus	0.2.3	9349	Navigation	Bus schedule for South Tyrol
scrabble	1.2	3165	Games	Scrabble in french
soundmanager	2.1.0	5307	Multimedia	Volume level scheduler
speedometer	1	139	Navigation	Simple Speedometer
stk	0.3	4493	Games	A 3D open-source arcade racer
sudowars	1.1	22837	Games	Multiplayer sudoku
swjournal	1.5	5955	Sports & Health	Track your workouts
tapsoffire	1.0.5	19920	Games	Guitar game
vitoshadm	1.1	567	Games	Helps you to make decisions
words	1.6	7125	Science & Education	Helps to study vocabulary for IELTS exam

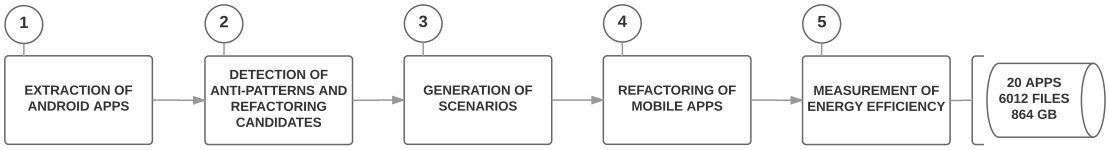

Figure 2: Data extraction process.

minimum value and the phone is set to keep the screen on. In order to avoid any kind of interferences during the measurements, only the essential Android services are run on the phone (for example, we deactivate WiFi if the app does not require it to be correctly executed, etc.).

When the oscilloscope is started it begins to store in memory energy measurements which are written to a *Comma Separated Values* (CSV) file when the scenario associated to the app finishes. In addition to energy, the generated file contains a timestamp for each sample. Once Algorithm 1 finishes, we have two files for each app and run: the energy trace and the execution trace. Using the existing timestamp in energy traces and the starting and ending time of methods calls in execution traces, energy consumption is calculated for each method called and this information is saved in a new CSV file for each app and run. From these files, we filtered out method names that does not belong to the *namespace* of the app. For example, for the Android *calculator* app, the main activity is located in the package `com.android2.calculator3`, and we only consider the methods included in this package as they correspond to the source code that we analyze to generate refactoring opportunities. This is done to reduce the noise of OS native processes running in the background, and third-party services. Finally, the median and average energy consumption of each app over the 30 runs is calculated.

Algorithm 1: Steps to collect energy consumption.

```
forall runs do
    forall apps do
        Install app (using adb).
        Start oscilloscope to measure energy.
        Run app (using adb).
        Play scenario (using Monkeyrunner).
        Stop oscilloscope.
        Download the execution trace file (using adb).
        Stop app (using adb).
        Clean app files (using adb).
        Uninstall app (using adb).
    end
end
```

3.3 Data Analysis and Discussion

In the following we describe the dependent and independent variables of this preliminary study, and the statistical procedures used to address each research question. For all statistical tests, we assume a significance level of 5%. In total we collected 864 GB of data from which 391 GB correspond to energy traces, 329 GB to execution traces. The amount of data generated from computing the energy consumption of methods calls using these traces is 144 GB.

(PQ1): What is the relation between anti-patterns and energy efficiency?

For **PQ1**, the *dependent variable* is the energy consumption for each app version (original, refactored). The *independent variable* is the existence of any of the anti-patterns studied, and it is true for the original design of the apps we studied, and false otherwise. We statistically compare the energy consumption between the original and refactored design

using a non-parametric test, Mann-Whitney U test. Because we do not know beforehand if the energy consumption will be higher in one direction or in the other, we perform a two-tailed test. For estimating the magnitude of the differences of means between original and refactored designs, we use the non-parametric effect size measure Cliff's d, which indicates the magnitude of the effect size of the treatment on the dependent variable. The effect size is small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

(PQ2): What is the relation between anti-pattern types and energy efficiency?
For **PQ2**, we follow the same methodology as **PQ1**. For each type of anti-pattern, we have three different apps containing an instance of the anti-pattern. We refactor these apps to obtain versions without the anti-pattern. We measure the energy consumption of the original and refactored versions of the apps 30 times to obtain the values of the *dependent variable*. The *independent variable* is the existence of the type of anti-pattern.

3.4 Results of the Preliminary Study

In Figure 3 we present the distribution of energy consumption for apps participating in anti-patterns AP and their refactored version NAP. We observe that removing anti-patterns in an app can sometimes have a negative impact on the energy efficiency of the app (see the results of *kindmind*, *matrixcalc*, *monsterhunter*). In the 18 remaining apps, the energy consumption is lower in apps without anti-patterns compare to apps with anti-patterns. This result suggests that developers should be careful when removing anti-patterns to improve the design quality of their apps as the operation can have an undesirable effect on energy efficiency (*e.g.,* it's the case for *kindmind*, *matrixcalc*, *monsterhunter*). This finding is consistent with a previous finding by Sahin et al. [15], that refactoring do not always lead to an improvement of the energy efficiency.

In the studied apps we corrected 24 anti-patterns in total. In seven cases (*i.e.,* 30%) we obtained a statistically significant difference between the energy consumption of the original and refactored versions of the apps, with Cliff's δ effect sizes ranging from small to large. Specifically, we obtained three apps with large effect size: *speedometer*, *gltron*, and *soundmanager* (2 type of anti-patterns); two with medium effect size: *oddscalculator*, *words*; and one with small effect size, *vitoshadm*. Therefore we reject H_{0_1} for these seven apps.

*Overall, our results suggest that different types of anti-patterns may impact the energy efficiency of apps differently. Our next research question (i.e., **PQ2**) investigates this hypothesis in more details.*

To answer **PQ2**, on the impact of different types of anti-patterns on energy efficiency, we present in Figure 4 the distribution of the energy consumption for each anti-pattern studied, and in Table 3 the results of the Mann-Whitney U test and Cliff's δ effect sizes.

Regarding object-oriented (OO) anti-patterns. In the first plot (position 1, 1 corresponding to *blackJacktrainer*) of Figure 4, we have the original version (ORI), and a refactored version when we remove a *Lazy class* instance (LC). We observe that the median is slightly higher for the original code in comparison with the refactored version. This trend holds for *tapsoffire* (4, 3) and *soundmanager* (3, 3) respectively, with the former one having statistically significance and large magnitude (ES). In the case of *Refused Bequest* (RB), two out of three apps show that removing the anti-pattern saves energy, and the difference is statistically significant for *vitoshadm*. A similar trend is observed for the *Blob*; two out of three apps report a decrease in energy consumption after removing the *Blob*, though the differences are not statistically significant.

Concerning *Long Parameter list* (LP), and *Speculative Generality* (SG), both report a negative impact in energy efficiency after refactoring. While for LP, all the apps point toward more energy consumption, in the case of SG, the energy consumption is increased in two out of three apps after refactoring. We explain the result obtained for LP by the fact that the creation of a new object (*i.e.,* the parameter object that contains the long list of parameters) adds to some extent more memory usage. For SG we do not have a plausible explanation for this trend. For both anti-patterns, the obtained differences in energy consumption is not statistically significant, hence we cannot conclude that these two anti-patterns always increase or decrease energy consumption.

Regarding Android anti-patterns. For *HashMap usage* (HMU) and *Private getters and setters* (PGS), we obtained statistically significant results for two apps. For *Binding Resources too early* (BE), the result is statistically significant for one app. In all cases, apps that contained these anti-patterns consumed more energy than their refactored versions that did not contained the anti-patterns. This finding is consistent with the recommendation of previous works (*i.e.,* [5], [6]) that advise to remove HMU, PGS, and BE

Table 3: Statistical tests for the difference in energy consumption of apps containing different types of anti-patterns. Mann—Whitney U Test and Cliff'S δ Effect Size (ES).

Application	Type	$p-$value	ES	ES Magnitude
mylocation	BE	0.57	0.03	small
SASAbus	BE	0.23	-0.13	small
speedometer	BE	**<0.05**	**-0.97**	**large**
calculator	BL	0.58	-0.13	small
quicksnap	BL	0.95	-0.03	small
swjournal	BL	0.23	-0.23	small
calculator	HMU	0.43	-0.1	small
gltron	HMU	**<0.05**	**-0.7**	**large**
oddscalculator	HMU	**<0.05**	**-0.34**	**medium**
soundmanager	IGS	**<0.05**	**-0.63**	**large**
sudowars	IGS	0.64	0.04	small
words	IGS	**<0.05**	**-0.44**	**medium**
blackjacktrainer	LC	0.4	-0.12	small
soundmanager	LC	**<0.05**	**-0.53**	**large**
tapsoffire	LC	0.36	-0.22	small
kindmind	LP	0.3	0.16	small
monsterhunter	LP	0.34	0.1	small
stk	LP	0.5	0.02	small
prism	RB	0.09	0.17	small
scrabble	RB	0.98	-0.04	small
vitoshadm	RB	**<0.05**	**-0.29**	**small**
matrixcalc	SG	0.49	0.09	small
prism	SG	0.72	0.03	small
quicksnap	SG	0.49	0.04	small

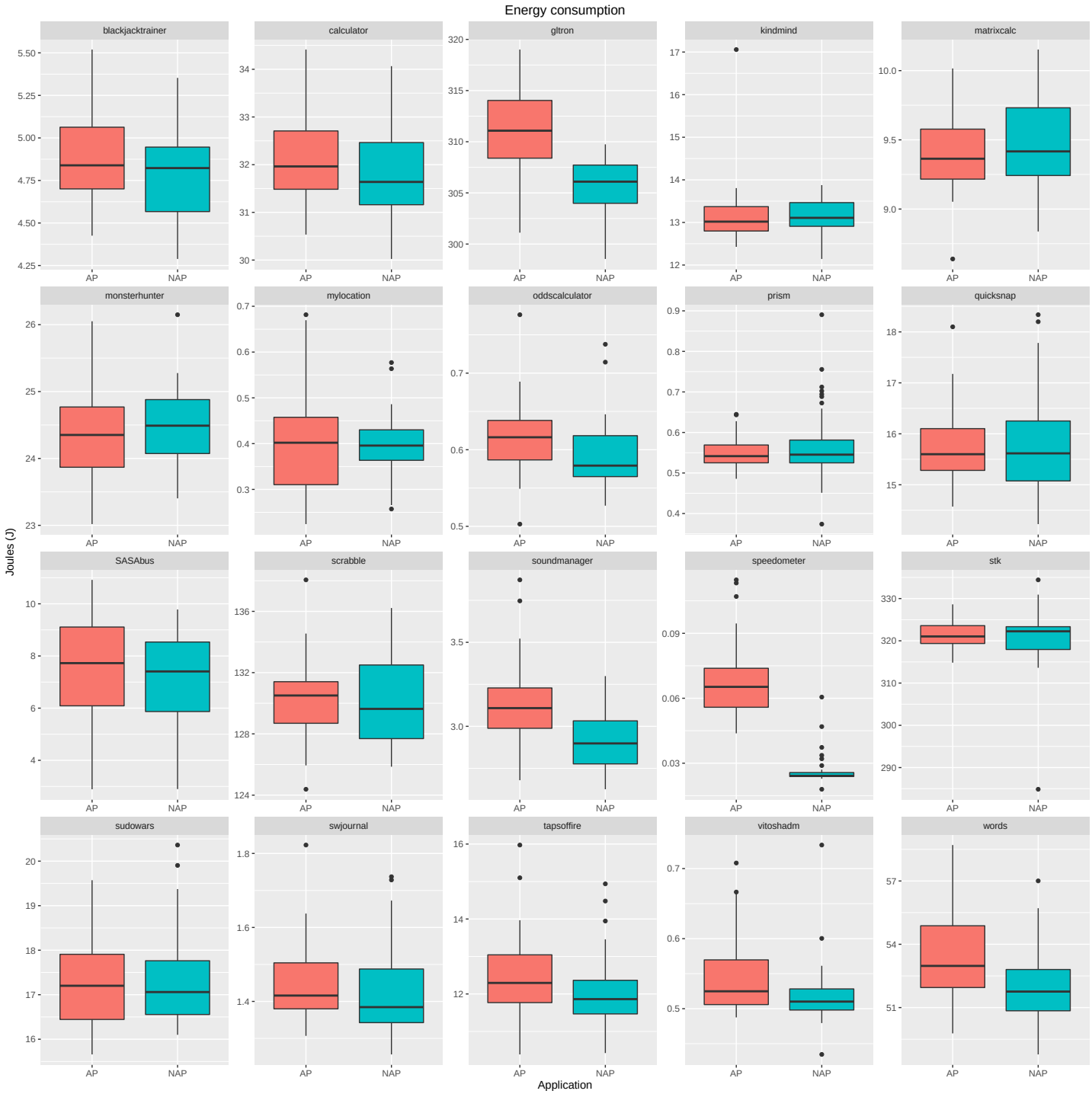

Figure 3: Energy consumption of apps with (AP) and without anti-patterns (NAP)

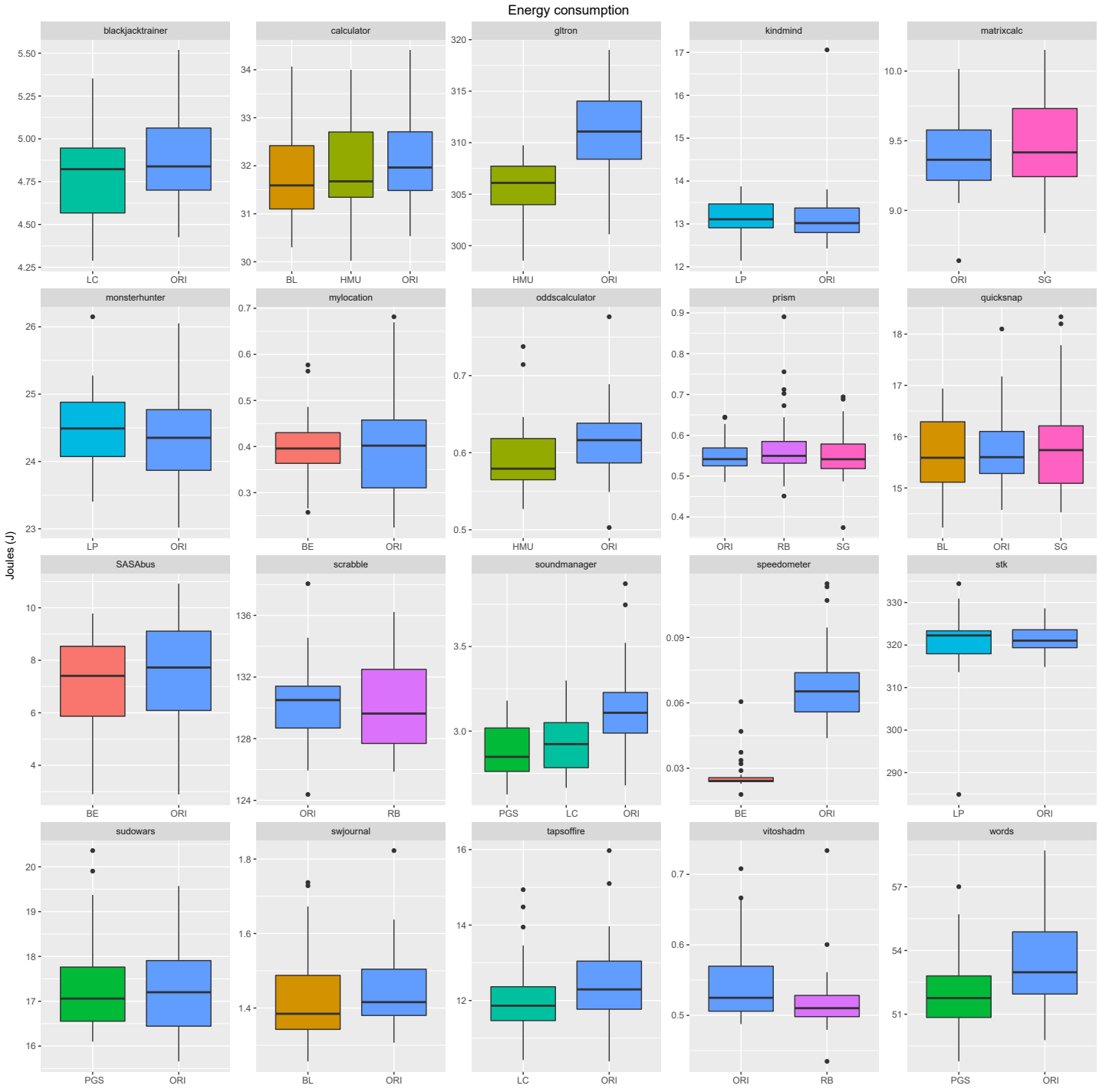

Figure 4: Energy consumption of apps after removing different types of anti-patterns

from Android apps, because of their negative effects on energy efficiency. Note that the amount of energy saved is influenced by the context in which the application runs. For example, *SASAbus* is a bus schedule application, and every time we launch the app it downloads the latest bus schedule, consuming a considerable amount of data and energy. As a result, the gain in energy for relocating the call method that starts the GPS device is negligible in comparison to the overall scenario. *Mylocation* is a simpler app, that only provides the coordinated position of mobile user. This app optimizes the use of the GPS device by disabling several parameters, like altitude and speed. It also sets the precision to *coarse* (approximate location[11]), and the power requirements to *low*. For this app, we observe a consistent improvement when the anti-pattern is removed, but in a small amount. On the other hand, we have *speedometer*, which is a simple app as well, that measures user's speed, but using *high precision mode*. *High precision mode* uses GPS and internet data at the same time to estimate location with high accuracy. In *speedometer*, we observe a high reduction in energy consumption when the anti-pattern is corrected, in comparison with the previous two apps.

In summary, removing ***Lazy class***, ***Refused Bequest***, ***Blob***, ***Binding Resources too early***, ***Private getters and setters***, *and* ***Hashmap usage*** *anti-patterns can improve the energy efficiency of an Android app (with the removal of the last three anti-patterns providing the biggest savings), while removing* ***Long Parameter list***, *and* ***Speculative Generality*** *anti-patterns can deteriorate the energy efficiency of the app.*

The impact of different types of anti-patterns on the energy consumption of mobile apps is not the same. Hence, we reject H_{02}.

4 Energy-Aware Automated Refactoring of Mobile Apps

After determining in Section 3 that the occurrence of anti-patterns impacts the energy consumption of mobile apps, we leverage this knowledge to propose an approach to improve the design quality of mobile apps, while controlling for energy efficiency. Our proposed approach is based on a search-based process where we generate refactoring sequences to improve the design of an app. This process involves evaluating several sequences of refactoring iteratively and the resultant design in terms of design quality and energy efficiency. Due to the impossibility to measure in real-time the impact of a refactoring sequence on the energy efficiency, because that will require to apply the refactoring in the code, compile, generate the *APK* and download it into the phone, every time the search-based process requires to evaluate a solution. We defined a strategy to estimate the impact of each refactoring operation on energy consumption, based on the results obtained in our preliminary study Section 3 as follows.

11. https://developer.android.com/guide/topics/location/strategies.html

1) We compute the energy consumption EC of an app a in our testbed using the following formulation.
$$EC(a) = \sum_{m \in M} EC(a_m) \quad (1)$$
Where M is the set of methods in a.
2) We obtain the median (med) of the 30 independent runs for $EC(a^K)$ and $EC(a^{ORI})$, where a^{ORI} is the original version of the app and a^K is the refactored version obtained by applying the refactoring operation type K.
3) The energy consumption coefficient of a refactoring K for an app a is obtained by computing the following expression.
$$\delta EC(a^K) = \frac{med(EC(a^K)) - med(EC(a^{ORI}))}{med(EC(a^{ORI}))} \quad (2)$$
If this value is negative, it means that in general, the refactored version consumes less energy. On the contrary, if this value is positive, it means that the refactored version consumes more energy than the original version.
4) We estimate the final energy coefficient of refactoring K, $\delta EC(K)$ by computing the following expression.
$$\delta EC(K) = med(\delta EC(a^k)); \forall a^k \in A^k \quad (3)$$
Where A is the set of apps where refactoring K was applied.

In the following, we describe the key components of our proposed approach *EARMO*, for the correction of anti-patterns while controlling for energy consumption.

EARMO overview

EARMO is comprised of four steps, depicted in Algorithm 2. The first step consists in estimating the energy consumption of an app, running a defined scenario. In the second step, we build an abstract representation of the mobile app's design, *i.e., code meta-model*. In the third step, the code meta-model is visited to search for anti-pattern occurrences. Once the list of anti-patterns is generated, the proposed approach determines a set of refactoring opportunities based on a series of pre- and post-conditions extracted from the anti-patterns literature [5], [6], [29], [30]. In the final step, a multiobjective search-based approach is run to find the best sequence of refactorings that can be legally applied to the code, from the refactoring opportunities list generated in the previous step. The solutions produced by the proposed approach meet two conflicting objectives: 1) remove a maximum number of anti-patterns in the system, and 2) improve the energy efficiency of the code design. In the following, we describe in detail each of these steps.

Step 1: Energy consumption measurement

We start by measuring the energy consumption of an app based on a user-defined scenario. The scenario can be input as a script written in *Hiromacro* as described in Section 3.2. Developers can measure the energy consumption of their apps by setting an energy estimation environment similar to the one presented in Section 3, or using a dedicated hardware-based energy measurement tool like GreenMiner [38]

Algorithm 2: EARMO Approach

Input : App to refactor (App), scenario (scen)
Output: Non-dominated refactoring sequences
Pseudocode EARMO (*Mobile app*)
  E_0 = Energy consumption measurement (App, scenario)
  /* We estimate the energy consumption of an app to estimate the energy improvement during our search-based approach */
  AM=Code meta-model generation (App)
  /* From the source code generate a light-weight representation of the code */
  MA=Code meta-model assessment (AM)
  /* 1. Detect anti-patterns in the system and generate a map of classes that contain anti-patterns */
  /* 2. Generate a list of refactoring operations to correct anti-patterns */
  Generation of optimal set of refactoring sequences (*AM, MA,* E_0)
  /* This is a generic template of the EARMO algorithm that finds the optimal set of refactoring sequences */
  return
Procedure Generation of an optimal set of refactoring sequences (*AM, MA,* E_0)
  $AM' = AM$
  $MA' = MA$
  $P_0 = GenerateInitialPopulation(MA')$
  $X_0 = \varnothing$
  /* X is the set of non-dominated solutions */
  /* Evaluation of P_0 */
  for *all* $S_i \in P_0$ **do**
    /* S_i is a refactoring sequence */
    $apply\_refactorings(AM', S_i)$
    $compute\_Quality(AM', S_i)$
    $compute\_Energy\_Consumption(AM', S_i, E_0)$
  end for
  /* Update the set of non-dominated solutions found in this first sampling */
  $X_0 = Update(X_0, P_0)$
  $t = 0$
  while *not* $StoppingCriterion$ **do**
    $t = t + 1$
    $P_t = Variation\_Operators(P_{t-1})$
    for *all* $S_i \in P_t$ **do**
      $apply\_refactorings(AM', S_i)$
      $compute\_Quality(AM', S_i)$
      $compute\_Energy\_Consumption(AM', S_i, E_0)$
    end for
    $X_t = Update(X_t, P_t)$
  end while
  $best\_solution = X$
  return best_solution

Once we estimated the energy consumption (E_0), we store this value and use it in the last step to evaluate the energy efficiency of a candidate refactoring solution during the search-based process.

Step 2: Code meta-model generation

In this step we generate a light-weight representation (a meta-model) of a mobile app, using static code analysis techniques, with the aim of evolving the current design into an improved version in terms of design quality and energy efficiency. This code meta-model contains information about the entities (classes, methods, and attributes) and how they interact with each other. It is used to detect anti-patterns, apply refactoring sequences and evaluate them.

Step 3: Code meta-model assessment

In this step we assess the quality of the code-meta model by (1) identifying anti-patterns in its entities, and (2) determining refactoring operations to correct them. For example, the correction of *Binding resources too early* anti-pattern can be divided in the following steps: detect classes with code statements that initialize energy-intensive components, *e.g.,* GPS or Wi-Fi, before the user or the app can interact with them; move the conflicting statements from its current position to a more appropriate method, *e.g.,* when the app interacts with the user, preventing an unnecessary waste of energy.

The correction of certain anti-patterns requires not only the analysis of a class as a single entity, but also their relationship with other classes (inter-class anti-patterns). For example, to correct instances of Blob in an app, we need to determine information related to the number of methods and attributes implemented by a given class, and compare it with the rest of the classes in the system. Then, we need to estimate the cohesion between its methods and attributes, and determine the existence of "controlling" relationships with other classes. After performing these inter-class analysis, we can propose refactorings to redistribute the excess of functionality from Blob classes to related classes, *i.e.,* move method refactoring.

Before adding a refactoring operation to the list of candidates, we validate that it meets all pre- and post-conditions for its refactoring type, to preserve the semantic of the code *cf.,* Opdkye [39]. For example, a pre-condition is that we cannot move a method to a class where there is a method with the same signature. An example of post-condition is that once we move a method from one class to another, there is no method in the source class that has the same signature as the method that was moved.

Step 4: Generation of optimal set of refactoring sequences

In this final step, we aim to find different refactoring sequences that remove a maximum number of anti-patterns, while improving the energy efficiency of mobile apps. Hence, we use *EMO* algorithms to obtain from all the set of possible refactoring combinations, the optimal solutions, *i.e.,* the ones that are not dominated. In the following, we describe the key elements of our multiobjective optimization process.

Solution representation

We represent a refactoring solution as a vector, where each element represents a refactoring operation (RO) to be applied, *e.g.,* a subset of refactoring candidates obtained by EARMO. Each refactoring operation is composed of several fields like an identification number (ID), type of refactoring, the qualified name of the class that contains the anti-pattern, and any other field required to apply the refactoring in the model. For example, in a *move method* operation we also need to store the name of the method to be moved, and the name of the target class, while in the correction of *long parameter list* we store the names of the long-parameter-list methods to be refactored. In Table 4 we present an example of a refactoring sequence. The ID is used to identify whether a RO already exist in a sequence when adding new refactoring candidates. The order is the position of the RO in the vector. We use the source class, and any other additional fields, to detect possible conflicts between existent ROs in a sequence. For example, it is not valid to have a *move method* RO after *inline class* if the name of the source class for both ROs is the same, as the class is removed after applying IC.

Table 4: Representation of a refactoring sequence.

ID	Type	Source class	Additional fields
4	Inline private getters and setters	[pkg].CalculatorWidget	private getters and setters: getDecimal()
52	Move Method	[pkg].BasicCalculator	target class: [pkg].CalculatorExpressionEvaluator method name cleanExpression(String)
2	Move resource request to visible method	[pkg].SelectLocationActivity	NONE
187	Collapse Hierarchy	[pkg].BasicCalculator	target class: [pkg].PanelSwitchingCalculator
189	Replace Inheritance with delegation	[pkg].Calculator	target class: [pkg].MatrixCalculator
8	Inline class	[pkg].CalculatorPadViewPager	target class: [pkg].ResizingButton
145	Replace Hashmap with Arraymap	[pkg].LruCache	HashMaps to Replace: mLruMap, mWeakMap
847	Introduce parameter object	[pkg].ImageManager	long-parameter-list methods: addImage(ContentResolver, String, long, Location, String, String, Bitmap, byte[], int[])

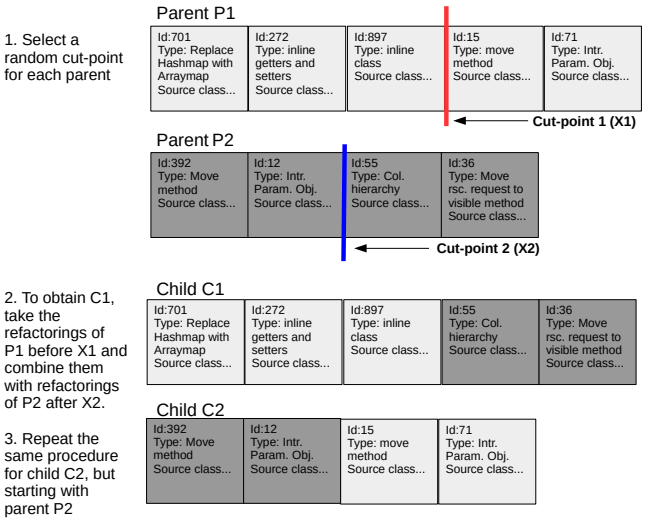

Figure 5: Example of cut and slice technique used as crossover operator.

Selection operator

The selection operator controls the number of copies of an individual (solution) in the next generations, according to its quality (fitness). Examples of selection operators are tournament selection or fitness proportionate selection [40].

Variation Operators

The variation operators allow metaheuristics to transform a candidate solution so that it can be moved through the decision space in the search of the most attractive solutions, and to escape from local optima. In EMO algorithms, we often find two main variation operators: crossover and mutation. Crossover consists of combining two or more solutions (known as parents) to obtain one or more new solutions (offspring). We implement the *Cut and splice technique* as crossover operator, which consists in randomly setting a *cut point* for two parents, and recombining with the elements of the second parent's cut point and vice-versa, resulting in two individuals with different lengths. We provide an example in Figure 5.

For mutation, we consider the same operator used in our previous work [27] that consists of choosing a random point in the sequence and removing the refactoring operations from that point to the end. Then, we complete the sequence by adding new random refactorings until there are no more

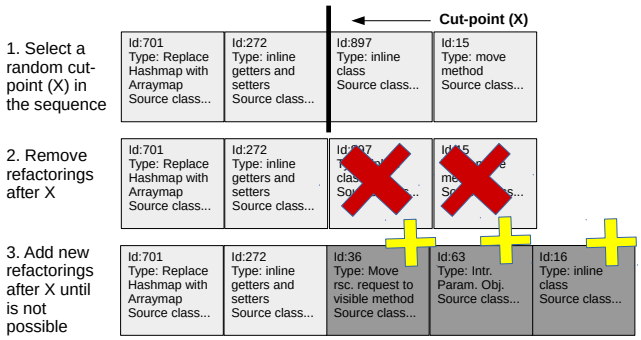

Figure 6: Example of the mutation operator used.

valid refactoring operations to add (*i.e.*, that do not cause conflict with the existent ones in the sequence).

Fitness functions

We define two fitness functions to evaluate the quality and the energy efficiency of the refactoring solutions. The function to evaluate the quality of the design is $DQ = 1 - \frac{NDC}{NC \times NAT}$, where NDC is the number of classes that contain anti-patterns, NC is the number of classes, and NAT is the number of different types of anti-patterns. The value of DQ, which is normalized between 0 and 1, rises when the number of anti-patterns in the app is reduced. A value of 1 represents the complete removal of anti-patterns, hence we aim to maximize the value of DQ. This objective function was introduced by Ouni et al. [21]. We follow this formulation because it is easy to implement and computationally inexpensive.

To evaluate the energy consumption of an app (expressed in Joules) after refactoring, we define the following formulation: let E_0 be the estimated energy consumption of an app a, r_i a refactoring operation type in a sequence $S = (r_1, \ldots, r_n)$. We estimate the energy consumption $EC(a)$ of the app resulting from the application of the refactoring sequence S to the app a as follows: $EC(a) = E_0 + \sum_{i=1}^{n} E_0 \times \delta EC(r_i)$, where $\delta EC(r_i)$ is the energy coefficient value of the refactoring operation r_i. We aim to minimize the value of EC during the search process.

In Algorithm 2, we present a generic pseudocode for the EMO algorithms used by our approach (lines 7-30).

The process starts by generating an initial population of refactoring sequences from the meta-model assessment step. Next, it applies each refactoring sequence in the code meta-model and measures the design quality (number of anti-patterns) and the energy saved by applying the refactorings included in the sequence (lines 12-16). The next step is to extract the non-dominated solutions (lines 17). From line 19 to 28, the main loop of the metaheuristic process is executed. The goal is to evolve the initial population, using the variation operators described before, to converge to the Pareto optimal front. Finally, in lines 29-30, the optimal refactoring sequences are retrieved.

5 Evaluation of EARMO

In this section, we evaluate the effectiveness of EARMO at improving the design quality of mobile apps while optimizing energy efficiency. The *quality focus* is the improvement of the design quality and energy efficiency of mobile apps, through search-based refactoring. The *perspective* is that of researchers interested in developing automated refactoring tools for mobile apps, and practitioners interested in improving the design quality of their apps while controlling for energy efficiency. The *context* consists of the 20 Android apps studied in Section 3, and three multiobjective metaheuristics (MOCell, NSGA-II, and SPEA2). We instantiate our generic EARMO approach using the three multiobjective metaheuristics, described in Section 2.3.

The code meta-model is generated using *Ptidej Tool Suite* [41]. We select this tool suite because it has more than ten years of active development and it is maintained in-house. Additionally, since October 10th, 2014, its source code have become open-source and released under the GNU Public License v2, easing replication.

The anti-patterns considered in the evaluation of EARMO are the ones described in Section 3.1. In the following, we describe the strategies implemented in EARMO to correct Android and object-oriented (OO) anti-patterns.

Move resource request to visible method (MRM). To determine the appropriate method to initialize a high-power-consumption component, it is necessary to understand the vendor platform. In our case, we illustrate the refactoring based on Android, but the approach can be extended to other operating systems. Figure 7 depicts a simplified flow-chart of the state paths of a single-activity android app. When users launch an app, the app goes through an initialization process that ends after the *onStart* method is executed (the app is visible). After the *onResume* method is executed, the user can interact with the app, but not before that. Hence, switching on a high-power-consumption component in the body of *OnCreate* is a terrible idea, in terms of energy consumption. Consequently, the refactoring consists in moving any hardware resource request from *onCreate* to *OnResume*.

Inline private getters and setters (IGS). The use of private getters and setters is expensive in Android mobile devices in comparison to direct field access. Hence, we inline the getters and setters, and access the private field directly. An illustrative example is provided in Figure 8.

Replace HashMap with array map (RHA). *ArrayMap* is a light-weight-memory mapping data structure included since Android API 19. The refactoring consists

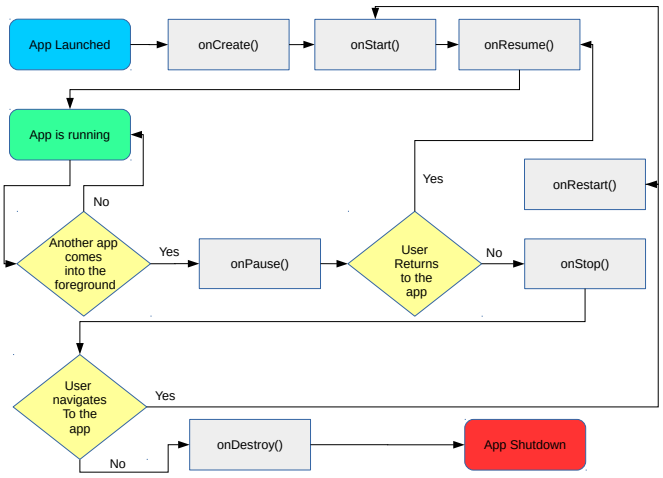

Figure 7: Android App flow-chart

```
1
2  private SplashView splashView;
3
4  private SplashView getSplashView() {
5          return splashView;
6      }
7  //This setter is not even used!
8   private void setSplashView(SplashView splashView) {
9          this.splashView = splashView;
10     }
11
12  public void initialize() {
13         final boolean firstLaunch = isFirstLaunch();
14
15         if (firstLaunch) {
16             getSplashView().showLoading();
17         }
18         ...
19         getSplashView().renderImportError();
20         ...
21         getSplashView().renderSplashScreenEnded();
22             }
23        ...
24
25         getSplashView().renderFancyAnimation();
26     }
```

```
1  private SplashView splashView;
2
3  // We inline private getters and setters
4   public void initialize() {
5          final boolean firstLaunch = isFirstLaunch();
6
7          if (firstLaunch) {
8             splashView.showLoading();
9          }
10         ...
11         splashView.renderImportError();
12         ...
13         splashView.renderSplashScreenEnded();
14             }
15        ...
16
17         splashView.renderFancyAnimation();
18     }
```

Figure 8: Example of *inline private getters and setters* refactoring. Original code on the top, and refactored code on the bottom.

in replacing the import of `java.util.HashMap` with `android.Util.Arraymap`, and any *HashMap* reference with *ArrayMap*. *ArrayMap* is compatible with the standard Java container APIs (*e.g.*, iterators, etc), and not further changes is required for this refactoring, as depicted in Figure 9.

Collapse hierarchy (CH). With this refactoring, we aim to *collapse* the features of a *unique* child class to the parent class, to reduce the complexity of the design, specially when both classes are really similar, or the child class does not add extra functionality, but was introduced presumably for handling future enhancements that never occurred. In Figure 10 we provide an example of SG anti-pattern found in *Android-*

```java
package com.glTron.Sound;

import java.util.HashMap;
...
public class SoundManager {

        private static HashMap<Integer, Integer> mSoundPoolMap;
        ...
        public static void initSounds(Context theContext)
        {
                ...
                mSoundPoolMap = new HashMap<Integer, Integer>();
                ...
        }
}
```

```java
package com.glTron.Sound;

import android.util.ArrayMap;
...
public class SoundManager {

        private static ArrayMap<Integer, Integer> mSoundPoolMap;
        ...
        public static void initSounds(Context theContext)
        {
                ...
                mSoundPoolMap = new ArrayMap<Integer, Integer>();
                ...
        }
}
```

Figure 9: Example of replacing HashMap with ArrayMap refactoring. Original code on the top, and refactored code on the bottom.

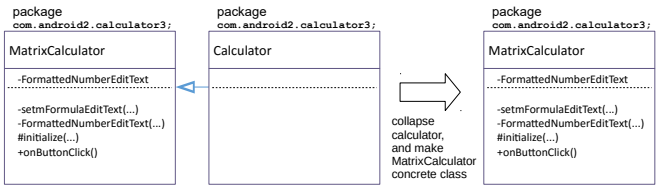

Figure 10: An example of SG in AndroidCalculator. Original code on the left, and refactored code on the right.

Calculator. We can observe that the class calculator does not implement any method, so there is no need to keep it in the design as it is, so the refactoring consists in removing the *abstract* modifier of the `MatrixCalculator` class, and replace all `Calculator` class references in the app to `MatrixCalculator`, including the *AndroidManifest.xml* file, as this class is declared as an Android activity.

Inline Class (IC). This refactoring consists in removing a *lazy class* in the system and transfering all its functionalities (if any) to any other class that is related to the LC (we assume that there is no hierarchy relationship, if so we would apply collapse hierarchy instead). To select such a class, we iterate over all the classes in the systems, searching for methods and attributes that access the LC features directly, or by public accessors (getters or setters). From those classes we choose the one with the larger number of access to the LC.

Introduce parameter object (IPO). In this refactoring, we extract a long list of parameters into a new object to improve the readability of the code. First we create a new class that will contain the extracted parameters. Then, we create a new instance of the parameter object with the values that we used to send to the LPL method. Next, in the LPL method, we remove the old parameters and add the new parameter object that we created. Finally, we replace each parameter from the method body with fields of the new parameter object. We show in Figure 11, an example of IPO in a method

```java
public static Uri addImage(ContentResolver cr, String title, long
        dateTaken,
        Location location, String directory, String filename,
        Bitmap source, byte[] jpegData, int[] degree) {
        OutputStream outputStream = null;
        String filePath = directory + "/" + filename;
        if (source != null) {
                source.compress(CompressFormat.JPEG,
                  CameraApplication.JPEG_HIGH_QUALITY, outputStream);
                degree[0] = 0;
        }
        else {
        outputStream.write(jpegData);
        degree[0] = getExifOrientation(filePath);
        }
        ...
        long size = new File(directory, filename).length();
        ContentValues values = new ContentValues(9);
        values.put(Images.Media.TITLE, title);
        ...
        values.put(Images.Media.DATE_TAKEN, dateTaken);
        ...
        }
```

```java
public static Uri addImage(AddImageParameter parObj) {
        OutputStream outputStream = null;
        String filePath = parObj.directory + "/" + parObj.filename;
        ...
        if (parObj.source != null) {
                parObj.source.compress(CompressFormat.JPEG,
                CameraApplication.JPEG_HIGH_QUALITY, outputStream);
                parObj.degree[0] = 0;
        } else {
                outputStream.write(parObj.jpegDatas);
                parObj.degree[0] = getExifOrientation(filePath);
        }
        ...
        long size = new File(parObj.directory, parObj.filename).length()
        ;
        ContentValues values = new ContentValues(9);
        values.put(Images.Media.TITLE, parObj.title);
        ...
        values.put(Images.Media.DATE_TAKEN, parObj.dateTaken);
        ...
}
```

Figure 11: Example of introduce parameter object refactoring. Original code on the top, and refactored code on the bottom.

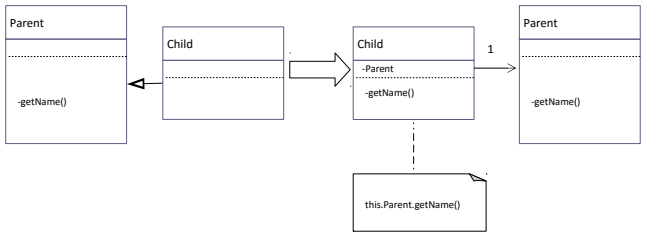

Figure 12: An example of applying RIWD in a class. Original code on the left, and refactored code on the right.

extracted from Quicksnap, which contains nine parameters.

Replace inheritance with delegation (RIWD). This refactoring is applied when we find a class that inherits a few methods from its parent class. To apply this refactoring, we create a field of the parent class, and for each method that the child use, we delegate to the field (parent class type), replacing the inheritance by an association. We present an example of this refactoring in Figure 12.

Move method (MM). This refactoring is applied to decompose a *Blob* class using *move method* and it is originally proposed by Seng et al. [23]. For each method in the *Blob* class, we search candidate classes from the list of parameter types in the method only if the target class is not a primitive type and the source code is reachable inside the app. Otherwise we select from the field types of the source class following the same rules.

Table 5: Descriptive statistics showing anti-pattern occurrences in the studied apps.

		O.O. AP					Android AP		
App	NOC	BL	LC	LP	RB	SG	BE	HMU	IGS
AndroidCalculator	43	2	3	0	8	5	0	14	0
BlackJackTrainer	13	1	3	0	0	0	0	0	0
GlTron	26	1	3	5	0	0	0	6	1
Kindmind	36	4	0	2	4	0	0	5	0
MatrixCalculator	16	1	0	2	1	2	0	0	0
MonsterHunter	194	11	1	2	32	0	0	3	0
mylocation	9	0	1	0	0	0	1	0	0
OddsCalculator	10	0	6	0	0	0	0	1	0
Prism	17	0	3	0	1	2	0	1	0
Quicksnap	76	3	6	1	1	1	0	10	4
SASAbus	49	0	1	0	0	1	2	7	0
Scrabble	9	0	4	0	0	1	0	2	0
SoundManager	23	0	9	1	0	0	0	6	2
SpeedoMeter	3	0	1	0	0	0	1	0	0
STK	25	0	1	1	0	0	0	4	0
Sudowars	110	26	2	3	21	6	0	9	1
Swjournal	19	0	1	1	0	0	0	0	0
TapsofFire	90	4	5	7	4	1	0	19	1
Vitoshadm	9	0	0	0	1	1	0	0	0
Words	136	10	4	12	6	1	0	15	0
Median	24	1	3	1	1	1	0	4	0
Total	913	63	54	37	79	21	4	102	9

5.1 Descriptive statistics of the studied Apps

Table 5 presents relevant information about anti-patterns contained in the studied apps. The second column contains the number of classes (NOC), and the following columns contain the occurrences of OO anti-patterns (3-7) and android anti-patterns (8-10). The last two rows summarize the median and total values for each column.

5.2 Research Questions

To evaluate the effectiveness of EARMO at improving the design quality of mobile apps while optimizing energy efficiency and its usability by software developers, we formulate the following three research questions:

(RQ1) To what extent EARMO can remove anti-patterns while controlling for energy efficiency?
This research question aims to assess the effectiveness of EARMO at improving design quality, while reducing energy consumption.
(RQ2) To what extent is design quality improved after applying energy-aware refactoring?
While the number of anti-patterns in a system serves as a good estimation of design quality, there are other quality attributes such as those defined by the QMOOD quality model [19] that are also relevant for software maintainers, *e.g.*, reusability, understandability and extendibility. This research question aims to assess the impact of the application of EARMO on these high-level design quality attributes.
(RQ3) Can EARMO generate useful refactoring solutions for mobile developers?
This research question aims to assess the quality of the refactoring recommendations made by EARMO from the point of view of developers. We aim to determine the kind of recommendation that developers find useful and understand why they may chose to discard certain recommendations.

5.3 Evaluation Method

In the following, we describe the approach followed to answer **RQ1**, **RQ2**, and **RQ3**.

For **RQ1**, we measure two *dependent variables* to evaluate the effectiveness of EARMO at removing anti-patterns in mobile apps while controlling their energy consumption:

- Design Improvement (DI). DI represents the delta of anti-patterns occurrences between the refactored (a') and the original app (a) and it is computed using the following formulation.

$$DI(a) = \frac{AC(a') - AC(a)}{AC(a)} \times 100. \quad (4)$$

  Where $AC(a)$ is the number of anti-patterns in an app a and $AC(a) \geq 0$. The sign of DI expresses an increment (+)/decrement (-) and the value represents the improvement amount in percentage. High negative values are desired.
- Estimated energy consumption improvement (EI). EI is computed using the following formulation.

$$EI(a) = \frac{EC(a') - EC(a)}{EC(a)} \times 100. \quad (5)$$

  Where $EC(a)$ is the energy consumption of an app a and $EC(a) \geq 0$. EI captures the improvement in the energy consumption of an app a after refactoring operation(s). The sign of EC expresses an increment (+)/decrement (-) and the value represents the amount in percentage. High negative values are desired.

In addition to estimating the energy consumption improvement (*i.e.*, EI) of apps using Equation 5, we also conduct manual energy consumption measurements on the original and refactored versions of five randomly selected apps, using our measurement setup described in Section 2.2. This step is important to verify if the estimated energy improvement (*i.e.*, EI) matches the real energy improvement obtained when refactoring recommendations are implemented in the code. For each selected app, we compute refactoring recommendations using EARMO and implement the refactorings in the source code of the app. Then, we measure the energy consumption of the original and refactored versions of the apps using the same scenario, and compute the difference between the obtained values. We compare the obtained result with EI.

The *independent variables* are the three selected EMO metaheuristics, *i.e.*, MOCell, NSGA-II, and SPEA2. We chose them because they are well-known evolutionary techniques that have been successfully applied to solve optimization problems, including refactoring [21], [42]. We implement all the metaheuristics used in this study using the jMetal Framework [43], which is a popular framework for solving optimization problems.

The performance of a metaheuristic can be affected by the correct selection of its parameters. The configurable settings of the search-based techniques used in this paper correspond to number of evaluations, population size, and the probability of the variation operators. To select the stopping criteria, we empirically tried different number of evaluations in the range of 1000 to 5000 and found no big

Table 6: Deltas of energy consumption by refactoring type.

Refactoring Type	δEC (ratio)
Collapse hierarchy	0.0056
Inline class	-0.0315
Inline private getters and setters	-0.0237
Introduce parameter object	0.0047
Move method	-0.0020
Move resource request to visible method	-0.0412
Replace HashMap with ArrayMap	-0.0160
Replace Inheritance with delegation	-0.0067

improvements after 2500, so we use this value to keep the execution time as short as possible.

For population size, we use a default value of 100 individuals; and for the probability of applying a variation operator we selected the parameters using a factorial design in the following way: we tested 16 combinations of mutation probability $p_m = (0.2, 0.5, 0.8, 1)$, and crossover probability $p_c = (0.2, 0.5, 0.8, 1)$, and obtained the best results with the pair $(0.8, 0.8)$.

Concerning the particular problem of automated-refactoring, the initial size of the refactoring sequence is crucial to find the best sequence in a timely manner. If the sequence is too long, the probability of conflicts between refactorings rises, affecting the search process. On the other hand, small sequences produce refactoring solutions of poor quality. To obtain a trade-off between this two scenarios, we experimented running the metaheuristics with four relative thresholds: 25, 50, 75 and 100 percent of the total number of refactoring opportunities, and found that 50 percent is the most suitable value for our search-based approach.

With respect to energy estimation, we show in Table 6 the energy consumption coefficient δec for each refactoring type, that we use in our experiment. These coefficients (*i.e.,* δec) were obtained from the formulation described in Section 4.

Note that for the *move method* refactoring, we did not use the energy consumption measured for the correction of *Blob*, as correcting a *Blob* requires many *move methods* to be applied. Hence, we measured the same apps used for *Blob* (*swjournal*, *quicksnap* and *calculator*) with and without moving exactly one method to estimate the effect of this refactoring. The results, which are not statistically significant, show a decrement in energy consumption.

In order to determine which one of our three EMO algorithms (*i.e.,* MOCell, NSGA-II, and SPEA2) achieves the best performance, we compute two different performance indicators: *Hypervolume (HV) [44]* and *SPREAD [33]*. We also perform Wilcoxon rank-sum test pair-wise comparisons between the three algorithms to validate the results obtained from HIV and SPREAD.

For **RQ2**, we use the Quality Model for Object-Oriented Design (QMOOD) [19] to measure the *impact* of the refactoring sequences proposed by EARMO, on the design quality of the apps. QMOOD defines six design quality attributes in the form of metric-quotient weighted formulas that can be easily computed on the design model of an app, which makes it suitable for automated-refactoring experimentations. Another reason for choosing the QMOOD quality model is the fact that it has been used in many previous works on refactoring [22], [45], which allows for a replication and comparison of the obtained results.

Table 7: QMOOD Evaluation Functions.

Quality Attribute	Quality Attribute Calculation
Reusability	-0.25 * DCC + 0.25 * CAM + 0.5 * CIS + 0.5 * DSC
Flexibility	0.25 * DAM - 0.25 * DCC + 0.5 * MOA +0.5 * NOP
Understandability	-0.33 * ANA + 0.33 * DAM - 0.33 * DCC + 0.33 * CAM -0.33 * NOP - 0.33 * NOM - 0.33 * DSC
Effectiveness	0.2 * ANA + 0.2 * DAM + 0.2 * MOA + 0.2 * MFA + 0.2 * NOP
Extendibility	0.5 * ANA -0.5 * DCC + 0.5 * MFA + 0.5 * NOP

where DSC is design size, NOM is number of methods, DCC is coupling, NOP is polymorphism, NOH is number of hierarchies, CAM is cohesion among methods, ANA is avg. num. of ancestors, DAM is data access metric, MOA is measure of aggregation, MFA is measure of functional abstraction, and CIS is class interface size.

In the following, we present a brief description of the quality attributes used in this study. Formulas for computing these quality attributes are described in Table 7. More details about the metrics and quality attributes can be found in the original source [19]. In this work we do not consider the functionality quality attribute because refactoring being a behavior-preserving maintenance activity, should not impact apps' functionalities.

- Reusability: the degree to which a software module or other work product can be used in more than one software program or software system.
- Flexibility: the ease with which a system or component can be modified for use in apps or environments other than those for which it was specifically designed.
- Understandability: the properties of a design that enables it to be easily learned and comprehended. This directly relates to the complexity of the design structure.
- Effectiveness: the design's ability to achieve desired functionality and behavior using OO concepts.
- Extendibility: The degree to which an app can be modified to increase its storage or functional capacity.

We compute the quality gain (QG) for each quality attribute using the following formulation.

$$QG(A_y) = \frac{A_y(a') - A_y(a)}{|A_y(a)|} \times 100 \quad (6)$$

Where $A_y(a)$ is the quality attribute y measurement for an app a, and a' is the refactored version of the app a. The sign expresses an increment (+)/decrement (-) and the value represents the improvement amount in percentage. Note that since the calculation of QMOOD attributes can lead to negative values in the original design, it is necessary to compute the absolute value of the divisor.

For **RQ3**, we conducted a qualitative study with the developers of our studied apps. For each app, we randomly selected some refactoring operations from the refactoring sequence recommended by EARMO, and submitted them to the developers of the app for approval or rejection. We choose three examples for each type of refactoring and for each app.

To measure developers' taking of the refactorings proposed, we compute for each app the *acceptance ratio*, which is the number of refactorings accepted by developers divided by the total number of refactorings submitted to the developers of the app. We also compute the *overall acceptance*

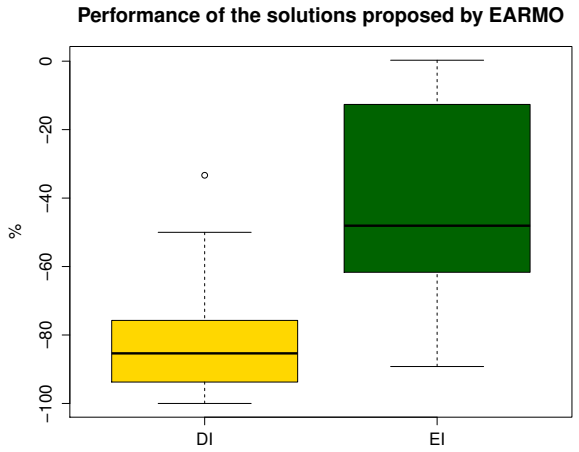

Figure 13: Distribution of anti-patterns and energy consumption reduction in the studied apps.

ratio for each type of anti-pattern, considering all the apps together.

5.4 Results of the Evaluation

In this section we present the answers to our three research questions that aim to evaluate EARMO.

RQ1: To what extent EARMO can remove anti-patterns while controlling for energy efficiency?

Because the metaheuristic techniques employed in this work are non-deterministic, the results might vary between different executions. Hence, we run each metaheuristic 30 times, for each studied app, to provide statistical significance. As a result, we obtain three reference Pareto front approximations (one per algorithm) for each app. From these fronts, we extract a global reference front that combines the best results of each metaheuristic for each app and, after that, dominated solutions are removed.

In Figure 13, we present the distribution of DI and EI metrics values, for each solution in the Pareto global reference front. Figure 13 highlights a median correction of 84% of anti-patterns and energy consumption improvement of 48% . To provide insights on the performance of EARMO, we present, in Table 8, the number of non-dominated solutions found for each app (column 2), the minimum and maximum values with respect to DI (columns 3-4), and EI metrics (columns 5-6). The number of non-dominated solutions are the number of refactorings sequences that achieved a compromise in terms of design quality and energy efficiency. Table 8 reports 2.5 solutions on average with a maximum of eight solutions (*words*). Thus, for the studied apps, a software maintainer has approximately three different solutions to choose to improve the design of an app.

In general, we observe that the results for DI and EI metrics are satisfactory, and we find that in nine apps EARMO reach 100% of anti-patterns correction with a maximum *EI* of 89%. With respect to the variability between apps with more than one solution, for EI metrics the difference between the maximum and minimum value is small, and

Table 8: Minimum and Maximum values (%) of DI and EI obtained for each app after applying EARMO.

	Solutions	DI		EI	
App		Min.	Max.	Min.	Max.
blackJacktrainer	1	-75	-75	-6.14	-6.14
calculator	5	-75	-93.75	-48.07	-53.55
gltron	2	-93.75	-100	-25.85	-26.32
kindmind	3	-80	-93.33	-18.42	-18.76
matrixcalculator	3	-33.33	-66.67	0.28	-0.67
monsterhunter	2	-81.63	-83.67	-43.95	-44.42
mylocation	1	-100	-100	-2.05	-2.05
oddscalculator	1	-100	-100	-14.64	-14.64
prism	2	-85.71	-100	-7.94	-9.18
quicksnap	2	-92.31	-96.15	-83.65	-84.88
SASAbus	1	-81.82	-81.82	-27.09	-27.09
scrabble	2	-85.71	-100	-12.36	-12.92
soundmanager	2	-94.44	-100	-35.36	-35.83
speedometer	1	-100	-100	-6.17	-6.17
stk	2	-83.33	-100	-11.05	-11.53
sudowars	8	-60.29	-76.47	-48.77	-63.93
swjournal	1	-100	-100	-5.67	-5.67
tapsoffire	3	-82.93	-87.8	-88.26	-89.21
vitoshadm	1	-100	-100	-3.57	-3.57
words	8	-75	-91.67	-56.83	-63.37

for DI too, except for the apps with more than two solutions (*i.e., calculator* and *words*). We observe that more than 65% of the apps contain more than one solution. To have an insight on those apps, we present in Figure 14 the Pareto Front (PF) for each app, where each point represents a solution with their corresponding values, DQ (x-axis) and EI (y-axis). The most attractive solutions are located in the bottom right of the plot.

All the points in the PF are considered equivalently good, but developers might show preference over the ones that favors the metric they want to prioritize. In our approach there are only two refactorings that we consider with a negative impact on the energy efficiency (*i.e.,* IPO and CH) so developers interested in improving energy efficiency could avoid to apply these two refactorings. For the rest of the refactorings, they could select the refactorings that improve more the energy efficiency (*e.g.,* they can chose to correct more Android anti-patterns). Other developers might be more conservative and select solutions located in the middle of these two objectives. Developers have the last word, and EARMO supports them by providing different alternatives.

Impact of refactoring sequences with respect to the type of anti-patterns. The anti-patterns analyzed in this study affect different quality metrics, and their definitions can be opposed, *e.g., Blob* and *Lazy class*. In Table 9, we present the median values of the DI metric for the non-dominated solutions of each type of anti-pattern. The results fall into two different categories.

- **Medium**. *Speculative generality* and *Blob* anti-patterns have median correction rates of 50% and 67%, respectively, while *Long parameter list* reached 75%.
- **High**. For the rest of the studied anti-patterns, the median correction rate is 100%, including the three Android anti-patterns studied and two OO anti-patterns (*i.e., Refused bequest, Lazy class*)

Energy consumption validation. The output of EARMO is a sequence of refactorings that balances anti-pattern correction and energy consumption. Developers select from the

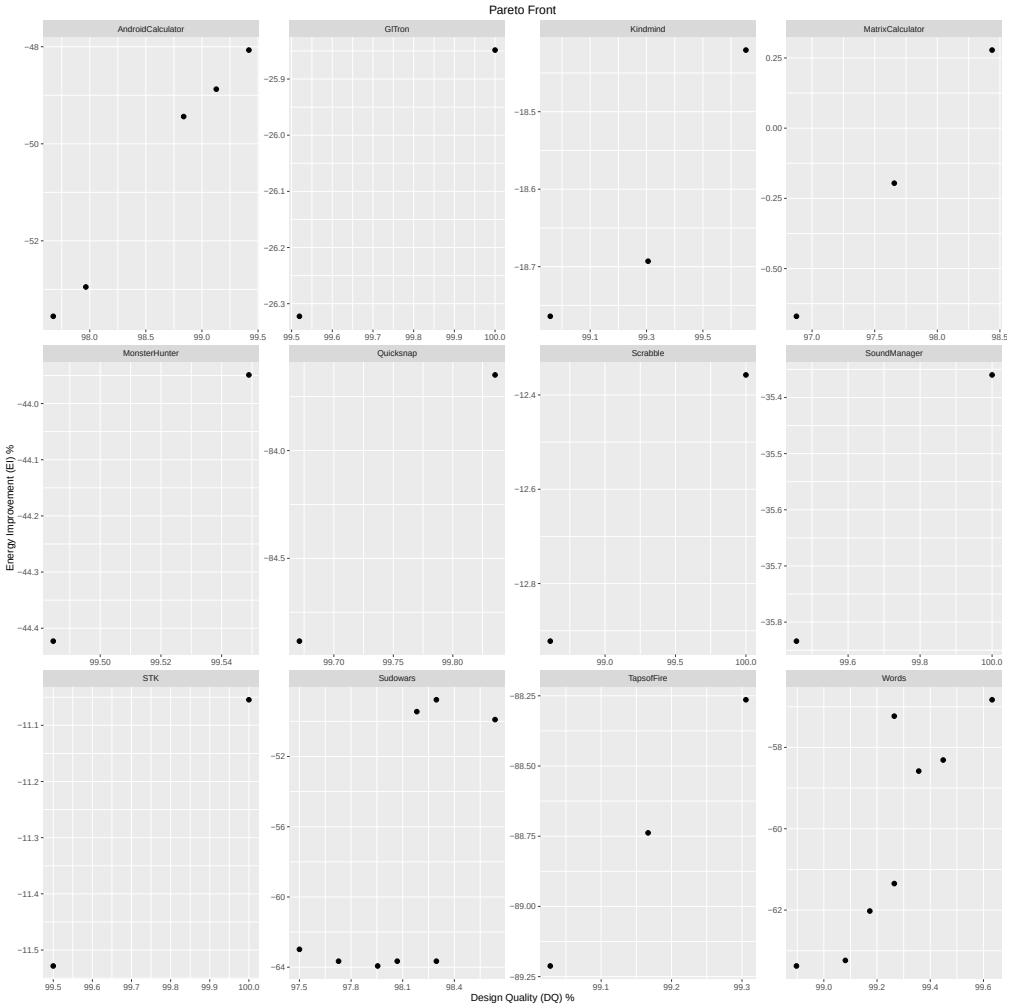

Figure 14: Pareto Front of apps with more than one non-dominated solution

Table 9: Median values of anti-patterns corrected by type (%).

	O.O. anti-patterns					Android anti-patterns		
App	BL	LC	LP	RB	SG	BE	HMU	IGS
blackjacktrainer	0	-100	NA	NA	NA	NA	NA	NA
calculator	-100	-100	NA	-75	-60	NA	NA	-100
gltron	-100	-100	-90	NA	NA	NA	-100	-100
kindmind	-100	NA	-50	-100	NA	NA	NA	-100
matrixcalculator	0	NA	-50	-100	-50	NA	NA	NA
monsterhunter	-27.27	-100	-75	-100	NA	NA	NA	-100
mylocation	NA	-100	NA	NA	NA	-100	NA	NA
oddscalculator	NA	-100	NA	NA	NA	NA	NA	-100
prism	NA	-100	NA	-100	-75	NA	NA	-100
quicksnap	-66.67	-100	-100	-100	-50	NA	-100	-100
SASAbus	NA	-100	NA	NA	0	-100	NA	-100
scrabble	NA	-100	NA	NA	-50	NA	NA	-100
soundmanager	NA	-100	-50	NA	NA	NA	-100	-100
speedometer	NA	-100	NA	NA	NA	-100	NA	NA
stk	NA	-100	-50	NA	NA	NA	NA	-100
sudowars	-59.62	-100	-66.67	-80.95	-66.67	NA	-100	-94.44
swjournal	NA	-100	-100	NA	NA	NA	NA	NA
tapsoffire	-75	-40	-85.71	-100	0	NA	-100	-100
vitoshadm	NA	NA	NA	-100	-100	NA	NA	NA
words	-85	-100	-91.67	-33.33	50	NA	NA	-100

Pareto front, the solutions that best fits their needs. For this manual validation we considered the preferences of a developer who wants to prioritize the energy efficiency of his/her app over design quality. Hence, we randomly selected five apps where the number of solutions in the Pareto front were greater than one. From this subset, we selected the solutions that report the best energy consumption. Then, we manually applied the sequence of refactorings (associated to each solution) to their corresponding source code. We ran the scenario after applying each individual solution to ensure that we were not introducing code regression. Finally, we compiled and generated the *apk* file to measure their energy consumption using our experimental setting described in Section 3. With this sanity check, we aim to verify that the trend of energy efficiency improvement estimated by EARMO corresponds to real measurements. For doing the measurement we use the same scenarios and methodology described in Section 3.2.

In Table 10 we present the solutions found by EARMO for the five apps that we randomly selected for the sanity check. We also include information about the refactorings that we applied and the ones that we discarded from the sequence due to errors occurring when running the scenario. It is important to mention that we applied the refactorings using the Android Studio tool support, and we do not find cases where refactorings violate any semantic pre- and post-condition. However, there are many cases, specially in

Table 10: Randomly selected apps for EC validation.

App	DI	EI	Discarded ref.	Applied ref.
calculator	88%	54%	19	45
gltron	94%	26%	19	13
quicksnap	96%	84%	69	119
sudowars	71%	64%	38	75
words	79%	63%	19	45

Table 11: Statistical tests for the randomly selected apps for EC validation.

App	$p-$value	Effect size	Effect magnitude
calculator	1.86E-09	-9.73E-01	large
gltron	5.97E-06	-8.31E-01	large
quicksnap	3.84E-02	-3.24E-01	medium
sudowars	1.86E-09	-1.00E+00	large
words	2.77E-02	-3.58E-01	medium

move method refactoring, where it is possible to introduce regression despite the fact that the refactoring is semantically correct. The refactorings were applied by one of the authors (PhD candidate with more than 5 years of experience in Java), and an intern (Msc. Student with two years of programming experience) working in pair. Whenever we observed an abnormal behavior in the app after applying a refactoring, we rolled back to the previous code version and discarded the conflicting refactoring. We provide a link to the original and refactored code that we make available online at http://swat.polymtl.ca/rmorales/EARMO/.

In Figure 15, we present the distribution of the energy consumption for the five apps. We can observe that all of them report energy improvements that follow the trends suggested by EARMO, but in smaller proportion (from 3% to 34%). This can be explained to some extent by the fact that we discarded some suggested refactorings from the sequence during our manual implementation. Another reason is the executed scenario, because EARMO suggested refactorings for all the classes in the system, but the scenario that we executed when measuring the energy could not cover all possible interactions with the app. Still we believe that the results are sound. A recent work by Banerjee reported an energy efficiency improvement from 3% to 29% in a testbed of 10 F-Droid apps with an automated refactoring approach for correcting violations of the use of energy-intensive hardware components [46].

To validate whether the results were statistically significant, we performed the Mann-Whitney U Test and computed Cliff's d effect size. We found that the results are statistically significant for all the apps as depicted in Table 11.

We conclude that including energy-efficiency as a separate objective when applying automatic refactoring can reduce the energy consumption of a mobile app, without impacting the anti-patterns correction performance.

Performance of the metaheuristics employed. As mentioned before, EARMO makes use of EMO techniques to find optimal refactoring sequences. Therefore, the results can vary from one technique to another. A software maintainer might be interested in a technique that provides the best results in terms of diversity of the solutions, and convergence of the algorithm employed. In the MO research community, the Hypervolume (*HV*) [44] is a quality indicator often used for this purpose, and higher values of this metric are desirable.

Table 12: HV. Median and IQR.

	MOCell	NSGAII	SPEA2
calculator	$1.32e-01_{8.3e-02}$	$8.92e-02_{1.3e-01}$	$9.47e-02_{1.8e-01}$
gltron	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$
kindmind	$0.00e+00_{1.0e-01}$	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$
matrixcalculator	$2.50e-01_{0.0e+00}$	$2.50e-01_{0.0e+00}$	$2.50e-01_{0.0e+00}$
monsterhunter	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$
prism	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$
quicksnap	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$
scrabble	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$
soundmanager	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$
stk	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$	$0.00e+00_{0.0e+00}$
sudowars	$4.25e-01_{1.3e-01}$	$4.95e-01_{1.2e-01}$	$5.45e-01_{1.2e-01}$
tapsoffire	$0.00e+00_{0.0e+00}$	$0.00e+00_{3.7e-02}$	$0.00e+00_{3.7e-02}$
words	$3.00e-01_{5.3e-02}$	$2.69e-01_{7.3e-02}$	$2.73e-01_{7.0e-02}$

Table 13: SPREAD. Median and IQR.

	MOCell	NSGAII	SPEA2
calculator	$6.89e-01_{3.0e-01}$	$1.12e+00_{4.7e-01}$	$8.73e-01_{5.6e-01}$
gltron	$6.78e-01_{1.8e-01}$	$1.07e+00_{1.8e-01}$	$1.08e+00_{2.7e-01}$
kindmind	$6.93e-01_{1.0e-01}$	$9.71e-01_{2.2e-01}$	$7.66e-01_{3.0e-01}$
matrixcalculator	$5.00e-01_{0.0e+00}$	$1.39e+00_{0.0e+00}$	$1.49e+00_{3.5e-03}$
monsterhunter	$8.97e-01_{4.3e-01}$	$9.70e-01_{2.1e-01}$	$9.27e-01_{1.1e-01}$
prism	$0.00e+00_{0.0e+00}$	$1.94e+00_{3.8e-02}$	$1.92e+00_{4.6e-02}$
quickSnap	$1.95e-01_{4.1e-01}$	$1.29e+00_{6.0e-01}$	$1.00e+00_{1.4e+00}$
scrabble	$5.00e-01_{1.0e+00}$	$1.50e+00_{3.8e-01}$	$1.62e+00_{7.8e-01}$
soundmanager	$1.00e+00_{1.7e-01}$	$1.00e+00_{0.0e+00}$	$1.00e+00_{0.0e+00}$
stk	$0.00e+00_{0.0e+00}$	$1.95e+00_{2.9e-02}$	$1.91e+00_{1.5e-01}$
sudowars	$7.96e-01_{1.3e-01}$	$8.53e-01_{1.4e-01}$	$8.41e-01_{1.3e-01}$
tapsoffire	$7.53e-01_{5.4e-01}$	$1.00e+00_{1.7e-01}$	$1.00e+00_{8.6e-02}$
words	$6.84e-01_{2.5e-01}$	$9.42e-01_{2.2e-01}$	$7.07e-01_{1.6e-01}$

In Table 12 we present the median and interquartile range (IQR) of the *HV* indicator for each metaheuristic and for each app with more than one solution. A special notation has been used in this table: a dark gray colored background denotes the best technique while lighter gray represents the second-best performing technique. For the apps with more than two solutions we observe a draw in *matrixcalculator*, while MOCell outperforms the other algorithms in two apps. SPEA-II outperforms the rest in *sudowars*, and gets second best in two more apps. NSGA-II obtains second-best in *sudowars*. In the cases where the metaheuristics cannot find more than one optimal solution, the value of *HV* is zero. Hence, the outperforming technique according to this quality indicator remains unknown.

Another quality indicator often used is the *Spread* [33]. It measures the distribution of solutions into a given front. Low values close to zero are desirable as they indicate that the solutions are uniformly distributed. In Table 13 we present the median and IQR results of the Spread indicator. We observe that MOCell outperforms the other techniques in 92% (12 apps) of cases, while *soundmanager* reports the same value for the three EMOs. SPEA2 gets the second best in 69% (nine apps), and NSGAII only in 8% (three apps).

To validate the results obtained by the *HV* and the *Spread* indicators, we perform pair-wise comparisons between the three metaheuristics studied, using the Wilcoxon rank sum test, with a confidence level of 95%. The results of these tests are summarized in Table 14. We introduce a special notation to facilitate the comprehension of the results. The ▲ symbol in a column indicates that the metaheuristic in the left

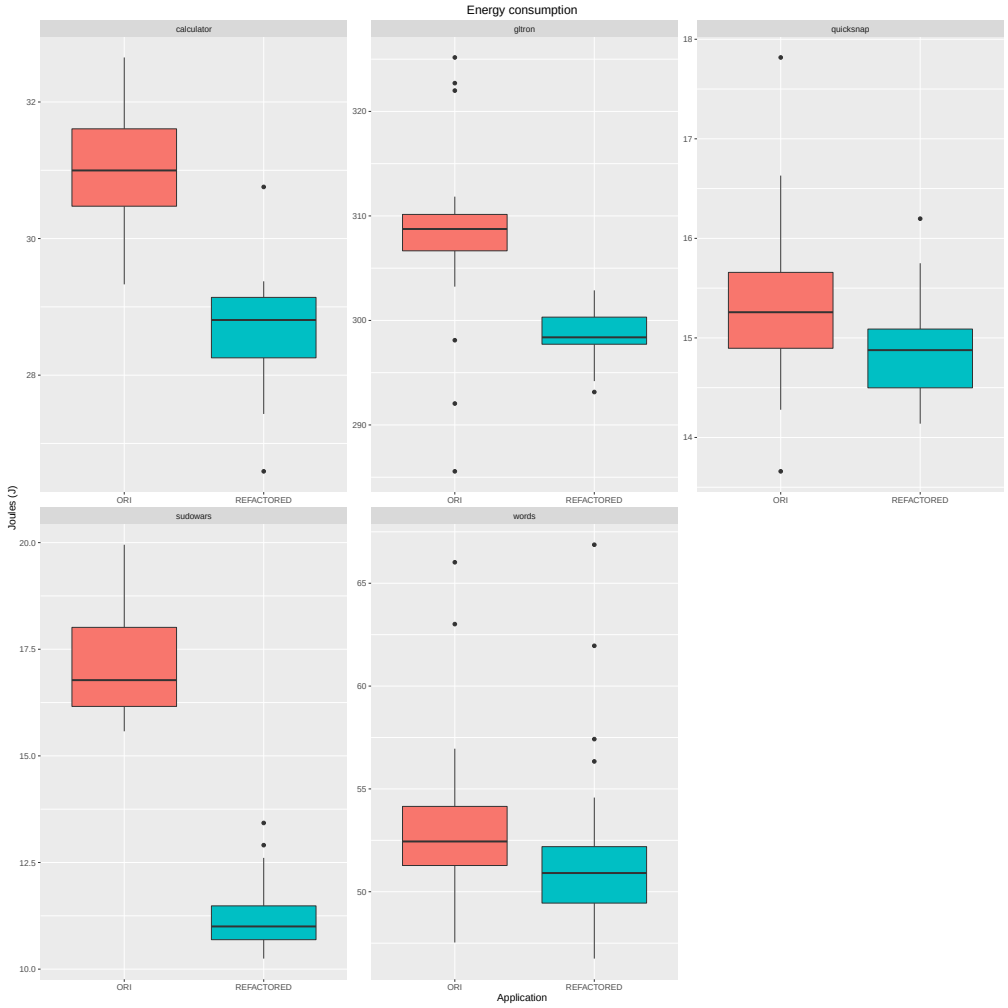

Figure 15: Energy consumption of the four apps selected for the sanity check.

side achieved a better performance than the one positioned after a comma. The ▽ indicates the opposite, and the – symbol in a column indicates that there is no statistically significant difference to reject the null hypothesis (*i.e.*, the two distribution have the same median). In each cell, the integer value represents the number of apps that fall in each of the aforementioned categories.

Table 14: Pair-wise Wilcoxon rank-sum test for HV and Spread indicators.

EMO Pair	Quality Indicator	▲	▽	–
MOCell, SPEA2	HV	0	0	13
	Spread	7	0	6
MOCell, NSGA-II	HV	0	1	12
	Spread	10	0	3
NSGA-II, SPEA2	HV	0	0	13
	Spread	1	0	12

Concerning *HV* indicator, only one app (*sudowars*) was statistically significant in the pair MOCell-NSGAII favoring the former one. So we can conclude that in general the performance of the three algorithms is similar. With respect to the *Spread* indicator, MOCell outperforms SPEA2 in seven apps, and NSGA-II in 10. In the pair NSGA-II-SPEA2, there is one app (*matrixcalculator*) where NSGA-II outperforms SPEA2. Hence, the solutions obtained by MOCell are better spread through the entire Pareto front than the other algorithms. Regarding execution time, we did not observed a significant difference between the execution time of the studied metaheuristics.

According to the Wilcoxon tests, MOCell is the best performing technique with respect to solution diversity, while regarding HIV the performance of the three EMO algorithms is similar. Developers and software maintainers should consider using MOCell when applying EARMO.

RQ2: To what extent is design quality improved after applying energy-aware refactoring?

In **RQ1**, we have shown that EARMO is able to find optimal refactoring sequences to correct anti-patterns while controlling for energy consumption. Although anti-patterns occurrences are good indicators of design quality, a software maintainer might be interested in knowing whether the applied refactorings produce code that is for example readable, easy to modify and–or extend. To verify such high-level design quality attributes, we rely on the QMOOD quality model. Table 15 presents the maximum and minimum quality gain achieved after applying the refactorings suggested by EARMO, for each app studied and for each QMOOD quality attribute.

Table 15: Quality gain (min. and max.) values derived from QMOOD design quality attributes for each app.

	Reusability		Understandability		Flexibility		Effectiveness		Extendibility	
App Name	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.
blackjacktrainer	-3.96	-3.96	-4.05	-4.05	-11.13	-11.13	9.29	9.29	94.86	94.86
calculator	-1.06	-0.58	-1.00	0.11	-14.52	6.73	1.85	3.18	13.51	21.07
gltron	-8.19	-2.83	-4.25	-2.39	-10.54	-4.93	3.79	6.12	38.01	40.79
kindmind	-1.10	-0.67	0.87	0.93	-0.12	1.78	-0.25	0.36	58.08	58.62
matrixcalculator	0.00	2.16	0.05	0.33	0.34	35.64	-0.51	-0.25	89.87	100.36
monsterhunter	0.08	0.10	0.00	0.10	0.43	0.73	0.42	0.48	57.22	57.69
mylocation	-1.56	-1.56	1.49	1.49	100.00	100.00	7.39	7.39	1.25	1.25
oddscalculator	-5.31	-5.31	-5.28	-5.28	70.86	70.86	28.93	28.93	42.15	42.15
prism	-44.36	-31.27	-8.14	-6.10	-14.46	-10.60	7.53	10.22	65.17	78.30
quicksnap	-2.74	-2.72	-3.77	-3.51	-39.15	-37.23	1.89	2.25	4.15	4.91
sasabus	-0.24	-0.24	-0.07	-0.07	-0.41	-0.41	1.11	1.11	64.57	64.57
scrabble	-8.41	-7.30	-0.80	-0.05	-13.41	-10.20	9.79	12.77	-1.67	1.60
soundmanager	-7.39	-5.67	-5.02	-3.40	-14.65	-5.92	24.11	26.17	32.32	44.32
speedometer	-0.93	-0.93	-1.22	-1.22	55.56	55.56	9.72	9.72	-124.16	-124.16
stk	-0.01	0.53	0.18	0.34	1.21	3.74	1.35	1.35	55.05	55.96
sudowars	-2.71	-0.76	-2.10	-1.12	-12.42	-5.43	-0.94	0.24	25.16	30.52
swjournal	-4.14	-4.14	-2.45	-2.45	-45.33	-45.33	0.87	0.87	6.88	6.88
tapsoffire	-0.39	-0.07	-2.97	-2.90	-13.36	-12.24	4.87	4.98	18.38	19.13
vitoshadm	-0.21	-0.21	0.10	0.10	8.71	8.71	3.79	3.79	153.06	153.06
words	2.11	3.92	0.44	0.81	4.19	8.11	-6.27	-3.70	72.88	74.27
Median values	**-1.24**		**-0.94**		**-4.07**		**3.14**		**40.78**	

- *Reusability, understandability and flexibility.* In general, the refactored apps report a slight decrease that ranges from 0.9% to 4% for these attributes. In the case of *reusability*, the *prism* app is an outlier, with a medium deterioration of *reusability* between 31% and 44%. EARMO finds two refactoring sequences (or two non-dominated solutions in the Pareto front) that are comprised of 5 refactoring operations. These refactorings are 3 *inline* operations, which have negatively impacted the *reusability* value because of the weight (*i.e.,* 0.5) that *reusability* assigns to the number of entities in the system (DSC metric). The fourth refactoring is *Inline private getters and setters*, which negatively affects the cohesion among methods (CAM) because one getter is inlined in the system. The last refactoring of the first refactoring sequence is *replace inheritance with delegation* which negatively impacts the coupling between classes (DCC), leading to a drop of 44.36% (minimum value) of *reusability*. In the second refactoring sequence, the last refactoring is *collapse hierarchy* which negatively impact DSC metric as well. Concerning *understandability*, we observe little variation through all the apps, making it the least impacted attribute among the 5 attributes studied. Finally, for *flexibility* we report a median of -4.07%. One remarkable case is *mylocation*, with 100% gain for this attribute. It has one solution comprised of two refactorings, *inline class* and *move resource request to visible method*. While the former one does not have a direct impact on the design, the inline of a class positively impacted this attribute because the number of classes is small (only 9 classes). Similarly, *oddscalculator* contains one solution with 7 *inline class* refactorings, and one *inline private getter*. On the other hand, *swjournal* has one solution composed mainly by *move method* refactorings (19), and one *inline* class. The *inline class* operation is likely responsible for the drop of the value of the attribute to 45%.
- *Effectiveness.* We report a small gain of 3.14%, with two outliers (*oddscalculator* and *soundmanager*). As we discussed before, *oddscalculator* is mainly composed of *inline class* refactorings. *Soundmanager* has two solutions, both contain 9 *inline classes*, 6 *inline getters/setters*, 2 *replace HashMap usage*. In addition, the second solution includes *introduce parameter-object* refactoring, which adds a new class to the design, has the highest *effectiveness* value for this app.
- *Extendibility.* For this attribute we report a considerable improvement of 41%. We attribute this increment to the removal of unnecessary inheritance (through *inline class*, *collapse hierarchy* and *refused bequest* refactorings). In fact, the *extendibility* function assigns a high weight to metrics related to hierarchy (*i.e.,* MFA, ANA). These are good news for developers interested in improving the design of their apps through refactoring, as the highly-competitive market of Android apps requires adding new features often and in short periods of time. Hence, if they interleave refactoring before the release of a new version, it will be easier to extend the functionality of their apps.

We conclude that our proposed approach EARMO can improve the design quality of an app, not only in terms of anti-patterns correction, but also their extendibility, and effectiveness.

RQ3: Can EARMO generate useful refactoring solutions for mobile developers?

We conducted a qualitative study with the developers of the 20 apps studied in this paper to gather their opinion about the refactoring recommendations of EARMO. The study took place between August 17th and September 17th 2016. 23 developers working in the apps were contacted but only 8 responded; providing feedback for a total of 8 apps. Table 16 provides some background information on the developers that took part in our qualitative study. Each developer has more than 3 years of experience and their primary programming language is Java. Half of the developers use Android Studio to program. 100% of them considered

refactorings to be useful but only 12% said that they perform refactoring frequently. We asked each developer to name the three refactorings that they perform the most. As we can see in Table 16, the most frequent refactorings performed by the developers are: to remove dead code, move method, inline class, extract class/superclass, collapse hierarchy, and extract interface. They also mentioned to extract repetitive code into new functions (extract method), and adjusting data structures.

For each app, we randomly selected three refactorings for each refactoring type, from the refactoring sequence in the Pareto front with the highest energy gain. We submitted the proposed refactorings to the developers of the app. We asked the developers if they accept the solution proposed by EARMO, and if not, to explain why. We also asked if there were any modification(s) that they would like to suggest to improve the proposed refactoring recommendations. In Figure 16, we present the acceptance ratio of the refactoring solutions proposed by EARMO, by app (left), and by anti-pattern (right).

We can observe that for four apps (*prism, scrabble, stk, matrixcalculator*), 100% of the refactorings suggested by EARMO were accepted. For three other apps (*calculator, kindmind, oddscalculator*) the acceptance ratio range from 40 to 57 %. The developer of the *GLTron* app rejected all the refactorings recommended for the app. However, the reasons behind her/his rejections are not convincing as we will discuss in the following paragraph. Overall, 68% of recommendations suggested by EARMO were accepted by developers.

The refactoring with the highest acceptance ratio is *inline private getters and setters*, while the one with the lowest acceptance ratio is *replace hashmap with arraymap*. The only app for which *replace hashmap with arraymap* was recommended is *GLTron*. The argument provided by the developer of *GLTron* to justify his disapproval of the refactoring is that because "GLtron runs on many platforms, introducing too many Android specific APIs would be a bad idea from a portability point of view". He also mentioned that because the hashmap contains few objects, the impact on performance is minimal. However, the Android documentation [47] emphasizes the advantages of using ArrayMap when the number of elements is small, in the order of three digits or less. In addition to this, the performance in energy consumption should not be ignored.

***Move method* refactoring** has an acceptance rate of 25%. The following reasons were provided by developers to justify their decision to reject some *move method* refactorings suggested by EARMO. For the *Calculator* app, the developer rejected two suggested *move method* refactorings, arguing that the candidate methods' concerns do not belong to the suggested target classes. However, s/he agrees that the source classes are Blobs classes that should be refactored. We obtained a similar answer from the developer of *Kindmind*, who also agrees that the classes identified by EARMO are instances of Blob, but proposes other target classes as well. To justify her/his rejection of all the three *move method* refactorings that were suggested for her/his app, the developer of *GLTron* argued that there are more important issues than moving a single method. However, she/he didn't indicate what were those issues.

Introduce parameter object. We found *long-parameter list* instances in *matrixcalculator*, *STK* and *GLTron*, and its only in emphGLTron that the developer rejected the two refactorings proposed, claiming that the new object will bloat the calling code of the method; and for the second one, that the method has been already refactored in a different way.

Collapse hierarchy. We found two instances of *speculative generality*, one in *Prism* (which was accepted) and another in *Calculator*; the former one was rejected because the collapsed class (which is empty) implements a functionality in the paid version. The developer wanted to keep the empty class to maintain compatibility between the two versions of the app (*i.e.,* free and paid versions). However, the developer agrees that the solution proposed by EARMO is correct, and will consider to remove the empty class in the future.

Inline class. Two *inline class* refactorings were proposed by EARMO, one in *Scrabble* and another in *OddsCalc*. The former one was rejected by the developer because she/he considers that inlining the lazy class will change the idea of the design.

Inline private getters and setters. EARMO recommended *Inline private getters and setters* refactorings in 7 out of the 8 apps for which we received developers' feedback. From a total of 11 *Inline private getters and setters* operations that were suggested by EARMO, only one was rejected, and this was in *GLTron*. The developer of *GLTron* argued that a method that is called only once require no performance optimizations.

The majority of recommendations made by EARMO were received favorably. For those that were rejected, it was not because they were incorrect or invalid, but because they affected certain aspects of the design of the apps that developers didn't wanted to change. The recommendations made by EARMO raised the awareness of developers about flaws in the design of their apps. This was true even when the suggested fixes (*i.e.,* the refactorings) for these design flaws were rejected by the developers.

Hence, we conclude that EARMO recommendations are useful for developers. We recommend that developers use EARMO during the development of their apps, since it can help them uncover design flaws in their apps, and improve the design quality and energy efficiency of their apps.

6 Threats to validity

This section discusses the threats to validity of our study following common guidelines for empirical studies [48].

Construct validity threats concern the relation between theory and observation. This is mainly due to possible mistakes in the detection of anti-patterns, when applying refactorings. We detected anti-patterns using the widely-adopted technique DECOR [12] and the guidelines proposed by Gottschalk and Android guidelines for developers [6], [28]. However, we cannot guarantee that we detected all possible anti-patterns, or that all those detected are indeed true anti-patterns. Concerning the application of refactorings for the preliminary study, we use the refactoring tool support of

Table 16: Background information on the surveyed developers.

App Name	Interval Age	Experience	Prog. Language	IDE	Top refactorings
Calculator	18 to 24	5-9 years	Java	Android Studio	Extract method, remove dead code, extract or remove new class/interface
OddsCalculator	35 to 44	3-4 years	Java	Eclipse	Move type to new file, move method/field.
Kindmind	25 to 34	<1 year	Java	Android Studio	Renaming variables and classes, extract method/class
GLTron	35 to 44	3-4 years	Swift	XCode	Adjusting data structures, move method, extract class/superclass, Inline class, Collapse hierarchy and extract interface
Scrabble	35 to 44	3-4 years	python	vim	Extract method, remove dead code, add encapsulation
Prism	45 to 54	10 years or more	Java	Eclipse	Extract variable, extract method, rename
Matrixcalc	18 to 24	3-4 years	Java	Android Studio	Refactoring duplicate code, renaming classes/methods and variables, remove dead code
STK	18 to 24	1-2 years	Java	Android Studio	Extract method, extract class

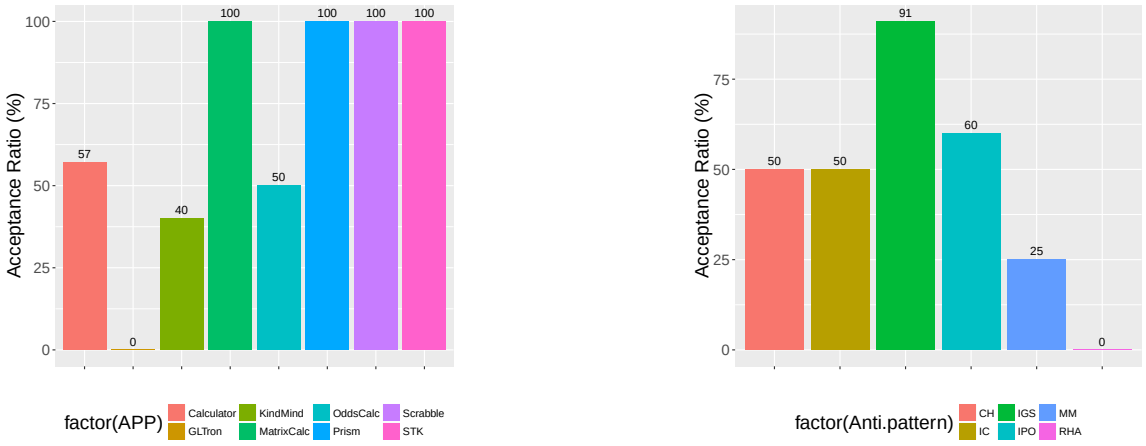

Figure 16: Acceptance ratio of the refactorings proposed by EARMO.

Android Studio and Eclipse, to minimize human mistakes. In addition, we verify the correct execution of the proposed scenarios and inspect the ADB Monitor to avoid introducing regression after a refactoring was applied. Concerning the correction improvement reported by EARMO, we manually validated the outcome of refactorings performed in the source code and the ones applied to the abstract model, to ensure that the output values of the objective functions correspond to the changes performed. However, we rely on the correct representation of the code generated by *Ptidej Tool Suite* [41]. We chose *Ptidej Tool Suite* because it is a mature project with more than ten years of active development, and it has been applied in several studies on anti-patterns, design patterns, and software evolution.

Considering energy measurements we used the same phone model used in other papers. Plus our measurement apparatus has a higher or the same number of sampling bits as previous studies and our sampling frequency is one order of magnitude higher than past studies. Overall, we believe our measurements are more precise or at least as precise as similar previous studies. As in most previous studies we cannot exclude the impact of the operating system. What is measured is a mix of Android and application actions. We mitigate this by running the application multiple times and we process energy and execution traces to take into account only the energy consumption of method calls belonging to the app.

Threats to internal validity concern our selection of anti-patterns, tools, and analysis method. In this study we used a particular yet representative subset of anti-patterns as a proxy for design quality. Regarding energy measurements, we computed the energy using well know theory and scenarios were replicated several time to ensure statistical validity. As explained in the *construct validity* our measurement apparatus is at least as precise as previous measurement setup.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models. In particular, we used a non-parametric test, Mann-Whitney U Test, Cliff's *d*, that does not make assumptions on the underlying data distribution.

Reliability validity threats concern the possibility of replicating this study. The apps and tools used in this study are open-source.

It is important to notice that the same model of phone and version of Android operating system should be used to replicate the study. In addition, considering the scenarios defined for each application, they are only valid for the *apk* versions used in this study, which are also available in our replication package. The reason is that the scenarios were collected considering approaches based on absolute coordinates and not on the identifier of components in the graphical user interface (GUI). Therefore, if another model of phone is used or the app was updated and the GUI changed, the scenarios will not be valid.

Threats to external validity concern the possibility to generalize our results. Our study focuses on 20 android apps with different sizes and belonging to different domains. Yet, more studies and possibly a larger dataset is desirable. Future replications of this study are necessary to confirm our findings. External validity threats do not only apply

to the limited number of apps, but also to the way they have been selected (randomly), their types (only free apps), and provenance (one app store). For this reason this work is susceptible to the App Sampling Problem [49], which exists when only a subset of apps are studied, resulting in potential sampling bias. Nevertheless, we considered apps from different size and domains, and the anti-patterns studied are the most critical according to developers perception [10], [50].

7 Related work

In this section, we discuss related works about automated-refactoring, Android anti-patterns, and the energy consumption of mobile apps.

7.1 Automated-Refactoring

Harman and Tratt [51] were the first to formulate the problem of refactoring as a multiobjective optimization (MO) problem. They defined two conflicting metrics as objectives to satisfy, and demonstrated the benefits of the Pareto optimality for the *Move method* refactoring. Ouni et al. [52] proposed a MO approach based on NSGA-II, with two conflicting objectives: removing anti-patterns, while preserving semantic coherence. For the first objective, they generated a set of rules to characterize anti-patterns from a set of bad design examples. The second objective is defined by the union of two metrics that measure the semantic similarity among classes. The first technique evaluates the cosine similarity of the name of the constituents, *e.g.,* methods, fields, types and the second technique considers dependencies between classes. Mkaouer et al. [53] proposed an extension of this work, by allowing user's interaction with the refactoring solutions. Their approach consists of the following steps: (1) a NSGA-II algorithm proposes a set of refactoring sequences; (2) an algorithm ranks the solutions and presents them to the user who will judge the solutions; (3) a local-search algorithm updates the set of solutions after n number of interactions with the user or when m number of refactorings have been applied.

Our proposed approach differs from the above-mentioned works in the following points: i) the context of our approach is mobile apps, with an emphasis on energy efficiency; ii) the level of automation in our approach is higher, as it does not depend on additional input from the user with respect to anti-patterns detection (*e.g.,* bad design examples).

Using four single-objective metaheuristics and a dataset of 1705 Mylyn interaction histories, Morales et al. [37] proposed an approach to guide the refactoring search using task context information. The difference with this work is that we focus on mobile apps using a multiobjective formulation, while the previous work targets only OO anti-patterns. In EARMO we do not leverage task context information to guide the search for refactoring solutions.

In a previous work [27], we propose a multiobjective approach to remove anti-patterns while controlling for testing effort, and show that it is possible to improve unit testing effort by 21%. This previous work differ from EARMO in the targeted systems (only object-oriented vs mobile), and the fact that energy consumption was not considered, but the testing effort of classes.

Recently, Banerjee et al. [46] proposed an approach to refactor mobile apps by relying on energy-efficiency guidelines to control for energy-intensive device components. They report a reduction in energy consumption from 3% to 29% of in their teastbed which was comprised of 10 F-Droid apps. While this work focuses only on improving the energy efficiency, our work aims to improve design quality by correcting OO and android anti-patterns. In addition, we examined the impact of different anti-patterns on the energy efficiency of apps and we evaluated the usefulness of our proposed refactoring approach using three different multiobjective metaheuristics.

7.2 Mobile anti-patterns

Linares-Vásquez et al. [54] leveraged DECOR to detect 18 OO anti-patterns in mobile apps. Through a study of 1343 apps, they have shown that anti-patterns negatively impact the fault-proneness of mobile apps. In addition, they found that some anti-patterns are more related to specific categories of apps.

Verloop [55] leveraged refactoring tools, such as PMD [12] or JDeodorant [56] to detect code smells in mobile apps, in order to determine if certain code smells have a higher likelihood to appear in the source code of mobile apps. In both works, the authors did not considered Android-specific anti-patterns.

Reimann et al. [57] proposed a catalogue of 30 quality smells specific to the Android platform. These smells were reported to have a negative impact on quality attributes like efficiency, user experience, and security. Reimann et al. also performed detections and corrections of certain code smells using the REFACTORY tool [58]. However, this tool has not been validated on Android apps [10].

Other researchers [10] have analyzed the evolution of the quality of mobile apps through the analysis of 3,568 versions of 106 popular Android apps from the Google Play Store. They used an approach, called *Paprika*, to identify three object-oriented and four Android-specific anti-patterns from the binaries of mobile apps.

Our proposed approach differs from these previous works in the sense that beside detecting anti-patterns in mobile apps, we propose an approach to generate optimal sequences of refactorings that achieve a maximum removal of anti-patterns from the mobile apps, while controlling for energy efficiency. We validate our results by measuring the energy efficiency of apps on a real mobile phone.

7.3 Energy Consumption

There are several works on the energy consumption of mobile apps [38], [59], [60], [61], [62], [63].

Some studies proposed software energy consumption frameworks [38] and tools [59] to analyze the impact of software evolution on energy consumption.

Green Miner [38] is a dedicated hardware mining software repositories testbed. The *Green Miner* physically measures the energy consumption of Android mobile devices

12. https://pmd.github.io/

and automates the reporting of measurements back to developers and researchers. A *Green Miner* web service[13] enables the distribution and collection of green mining tests and their results. The hardware client unit consists of an *Arduino*, a breadboard with an *INA219* chip, a *Raspberry Pi* running the *Green Miner* client, a USB hub, and a *Galaxy Nexus* phone (running Android OS 4.2.2) which is connected to a high-current 4.1V DC power supply. Voltage and amperage measurement is the task of the *INA219* integrated circuit which samples data at a frequency of $50\,Hz$. Using this web service, users can define tests for Android apps and run these tests to obtain and visualize information related to energy consumption.

Energy models can be provided by a *Software Environment Energy Profile (SEEP)* whose design and development enables the per instruction energy modeling. Unfortunately, it is not common practice for manufacturers to provide *SEEPs*. Because of that, different approaches have been proposed to measure the energy consumption of mobile apps. Pathak et al. [64] proposed *eprof*, a fine-grained energy profiler for Android apps, that can help developers understand and optimize their apps energy efficiency. In [65], authors proposed the software tool *eLens* to estimate the power consumption of Android applications. This tool is able to estimate the power consumption of real applications to within 10% of ground-truth measurements. One of the most used energy hardware profilers is the *Monsoon Power Monitor* which has been used in several works. By using this energy hardware profiler a qualitative exploration into how different Android *API* usage patterns can influence energy consumption in mobile applications has been studied by Linares-Vasquez et al. [66].

Other works aimed to understand software energy consumption [61], its usage [15], or the impact of users' choices on it [62], [67].

Da Silva et al. [17] analyzed how the inline method refactoring impacts the performance and energy consumption of three embedded software written in Java. The results of their study show that inline methods can increase energy consumption in some instances while decreasing it in others.

Sahin et al. [68] investigated how high-level design decisions affect an application's energy consumption. They discuss how mappings between software design and power consumption profiles can provide software designers and developers with insightful information about their software power consumption behavior.

Pinto et al. [69] have suggested a refactoring approach to improve the energy consumption of parallel software systems. They manually applied this refactoring approach to 15 open source projects and reported an energy saving of 12%.

Researchers [15] have investigated the impact of six commonly-used refactorings on 197 apps. The results of their study have shown that refactorings impact energy consumption and that they can either increase or decrease the amount of energy used by an app. The findings of [15] also highlighted the need for energy-aware refactoring approaches that can be integrated in IDEs.

13. https://pizza.cs.ualberta.ca/gm/index.py

Hecht et al. [18] conducted an empirical study focusing on the individual and combined performance impacts of three Android performance anti-patterns on two open-source Android apps. These authors evaluated the performance of the original and corrected apps on a common user scenario test. They reported that correcting these Android code smells effectively improves the user interface and memory performance.

Recently, researchers [70] have examined research results published in top software engineering venues and highlighted the need for refactoring approaches that deal with software energy consumption issues.

Our work contributes to fill this gap in the literature.

8 Conclusion and Future Work

In this paper we introduce EARMO, a novel approach for refactoring mobile apps while controlling for energy efficiency. This approach aims to support the improvement of the design quality of mobile apps through the detection and correction of Object oriented and Android anti-patterns. To assess the performance of EARMO, we implemented our approach using three evolutionary multiobjective techniques and we evaluated it on a benchmark of 20 free and open-source Android apps, having different sizes and belonging to various categories. The results of our empirical evaluation show that EARMO can remove 84% of anti-patterns and save 34% of energy.

We also demonstrated that in the instance of search space explored by the metaheuristics implemented, different compromise solutions are found, justifying the need for a multiobjective formulation.

Concerning the quality of the solutions proposed, we evaluated the overall design quality of the refactored apps in terms of five high-level quality attributes, and reported gains in terms of understandability, flexibility, and extendibility of the resulting designs. We conducted a qualitative study to assess the quality of the refactoring recommendations made by EARMO from the point of view of developers. Developers found 68% of refactorings suggested by EARMO to be very relevant.

As future work, we intend to extend our approach to detect and correct more mobile anti-patterns. We also plan to apply EARMO on larger datasets, and further evaluate it through user studies with mobile apps developers.

References

[1] G. Anthes, "Invasion of the mobile apps," *Commun. ACM*, vol. 54, no. 9, pp. 16–18, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/1995376.1995383
[2] J. Voas, J. B. Michael, and M. van Genuchten, "The mobile software app takeover," *Software, IEEE*, vol. 29, no. 4, pp. 25–27, July 2012.
[3] D. L. Parnas, "Software aging," in *ICSE '94: Proc. of the 16th Int'l conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 279–287.
[4] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *Software Engineering, IEEE Transactions on*, vol. 27, no. 1, pp. 1–12, 2001.
[5] M. Gottschalk, J. Jelschen, and A. Winter, "Energy-efficient code by refactoring," in *Softwaretechnik Trends*, vol. 33, no. 2. Bonn: Gesellschaft für Informatik, 05 2013, pp. 23–24.
[6] "Android performance tips," https://developer.android.com/training/articles/perf-tips.html, June 2016.

[7] F. Khomh, M. D. Penta, Y.-G. Gueheneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.
[8] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proc. of the 29th Int'l Conference on Software Maintenance*, 2013, pp. 270–279.
[9] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *3rd Int'l Symposium on Empirical Software Engineering and Measurement, ESEM 2009,*, 2009, pp. 390–400.
[10] G. Hecht, B. Omar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the Software Quality of Android Applications along their Evolution," in *30th IEEE/ACM International Conference on Automated Software Engineering*, ser. Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), L. Grunske and M. Whalen, Eds. Lincoln, Nebraska, United States: IEEE, Nov. 2015, p. 12. [Online]. Available: https://hal.inria.fr/hal-01178734
[11] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Technology (QUATIC), 2010 7th Int'l Conf. on the*. IEEE, 2010, pp. 106–115.
[12] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010.
[13] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *IEEE Int'l Conference on Software Maintenance, ICSM*. IEEE Computer Society, 2004, Conference Proceedings, pp. 350–359.
[14] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 2008, pp. 329–331.
[15] C. Sahin, L. L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *International Symposium on Empirical Software Engineering and Measurement, ESEM*, 2014, pp. 36:1–36:10.
[16] J. J. Park, J. Hong, and S. Lee, "Investigation for software power consumption of code refactoring techniques," in *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.*, 2014, pp. 717–722.
[17] W. G. P. da Silva, L. Brisolara, U. B. Correa, and L. Carro., "Evaluation of the impact of code refactoring on embedded software efficiency," in *In Proceedings of the 1st Workshop de Sistemas Embarcados*. Bonn: Gesellschaft für Informatik, 2010, pp. 145–150.
[18] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: ACM, 2016, pp. 59–69. [Online]. Available: http://doi.acm.org/10.1145/2897073.2897100
[19] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *Software Engineering, IEEE Transactions on*, vol. 28, no. 1, pp. 4–17, 2002.
[20] P. Bourque, R. E. Fairley *et al.*, *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.
[21] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.
[22] M. O'Keeffe and M. O. Cinnéide, "Search-based software maintenance," in *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, 2006, Conference Proceedings, pp. 10 pp.–260.
[23] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," *GECCO 2006: Genetic and Evolutionary Computation Conference, Vol 1 and 2*, pp. 1909–1916, 2006.
[24] C. L. Simons, I. C. Parmee, and R. Gwynllyw, "Interactive, evolutionary search in upstream object-oriented class design," *Software Engineering, IEEE Transactions on*, vol. 36, no. 6, pp. 798–816, 2010.
[25] R. Mahouachi, M. Kessentini, and M. O. Cinnéide, *Search-Based Refactoring Detection Using Software Metrics Variation*. Springer, 2013, pp. 126–140.
[26] M. W. Mkaouer, M. Kessentini, S. Bechikh, and M. O. Cinnéide, *A Robust Multi-objective Approach for Software Refactoring under Uncertainty*. Springer, 2014, pp. 168–183.
[27] R. Morales, A. Sabane, P. Musavi, F. Khomh, F. Chicano, and G. Antoniol, "Finding the best compromise between design quality and testing effort during refactoring," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, Conference Proceedings, pp. 24–35.
[28] M. Gottschalk, "Energy refactorings," Master's thesis, Carl von Ossietzky University, 2013.
[29] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998.
[30] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1^{st} ed. Addison-Wesley, June 1999.
[31] J. Knowles, L. Thiele, and E. Zitzler, "A tutorial on the performance assessment of stochastic multiobjective optimizers," *Tik report*, vol. 214, pp. 327–332, 2006.
[32] K. Deb, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001, vol. 16.
[33] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, 2002.
[34] E. Zitzler, M. Laumanns, L. Thiele, E. Zitzler, E. Zitzler, L. Thiele, and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm," 2001.
[35] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "MOCell: A cellular genetic algorithm for multiobjective optimization," *International Journal of Intelligent Systems*, pp. 25–36, 2007.
[36] ——, "Design issues in a multiobjective cellular genetic algorithm," in *Proceeding of the Conference on Evolutionary Multi-Criterion Optimization, volume 4403 of LNCS*. Springer, 2007, pp. 126–140.
[37] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers context for automatic refactoring of software anti-patterns," *Journal of Systems and Software*, 2016, to appear. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216300632
[38] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "Greenminer: A hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, 2014, pp. 12–21.
[39] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.
[40] B. L. Miller and D. E. Goldberg, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.
[41] Y.-G. Guéhéneuc, "Ptidej: Promoting patterns with patterns," in *Proceedings of the 1st ECOOP workshop on Building a System using Patterns. Springer-Verlag*, 2005.
[42] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, Dec. 2012. [Online]. Available: http://doi.acm.org/10.1145/2379776.2379787
[43] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, pp. 760–771, 2011.
[44] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *evolutionary computation, IEEE transactions on*, vol. 3, no. 4, pp. 257–271, 1999.
[45] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, no. 0, pp. 18 – 39, 2015.
[46] A. Banerjee and A. Roychoudhury, "Automated re-factoring of android apps to enhance energy-efficiency," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: ACM, 2016, pp. 139–150. [Online]. Available: http://doi.acm.org/10.1145/2897073.2897086
[47] "Arraymap," https://developer.android.com/reference/android/support/v4/util/ArrayMap.html, June 2016.
[48] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.
[49] W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang, "The app sampling problem for app store mining," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15.

Piscataway, NJ, USA: IEEE Press, 2015, pp. 123–133. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820535

[50] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Software Maintenance and Evolution (ICSME), 2014 IEEE Int'l Conference on*. IEEE, 2014, pp. 101–110.

[51] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, Conference Proceedings, pp. 1106–1113.

[52] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, Conference Proceedings, pp. 347–356.

[53] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE Int'l Conf. on Automated software engineering*. ACM, 2014, pp. 331–336.

[54] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabane, D. Poshyvanyk, and Y.-G. Guéhéneuc, "Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps," in *Proceedings of the 22nd International Conference on Program Comprehension*, C. K. Roy, A. Begel, and L. Moonen, Eds. ACM, 2014, pp. 232–243.

[55] D. Verloop, *Code Smells in the Mobile Applications Domain*. TU Delft, Delft University of Technology, 2013.

[56] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *ICSM*, 2007, pp. 519–520.

[57] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers." *Softwaretechnik-Trends*, vol. 34, no. 2, 2014.

[58] J. Reimann, M. Seifert, and U. Aßmann, "On the reuse and recommendation of model refactoring specifications," *Software and System Modeling*, vol. 12, no. 3, pp. 579–596, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10270-012-0243-2

[59] K. Aggarwal, A. Hindle, and E. Stroulia, "GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption," in *ICSME*. IEEE, 2015, pp. 311–320.

[60] I. Polato, D. Barbosa, A. Hindle, and F. Kon, "Hybrid HDFS: decreasing energy consumption and speeding up hadoop using ssds," *PeerJ PrePrints*, vol. 3, p. e1320, 2015. [Online]. Available: http://dx.doi.org/10.7287/peerj.preprints.1320v1

[61] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about the energy consumption of software?" *PeerJ PrePrints*, vol. 3, p. e886, 2015.

[62] C. Zhang, A. Hindle, and D. M. Germán, "The impact of user choice on energy consumption," *IEEE Software*, vol. 31, no. 3, pp. 69–75, 2014. [Online]. Available: http://dx.doi.org/10.1109/MS.2014.27

[63] K. Rasmussen, A. Wilson, and A. Hindle, "Green mining: energy consumption of advertisement blocking methods." in *GREENS*, 2014, pp. 38–45.

[64] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *EuroSys*, P. Felber, F. Bellosa, and H. Bos, Eds. ACM, 2012, pp. 29–42.

[65] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating Mobile Application Energy Consumption using Program Analysis," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, May 2013.

[66] M. Linares-Vsquez, G. Bavota, C. Bernal-Crdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597085

[67] R. Saborido, G. Beltrame, F. Khomh, E. Alba, and G. Antoniol, "Optimizing User Experience in Choosing Android Applications," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Mar. 2016.

[68] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. E. Kiamilev, L. L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage." in *GREENS*. IEEE, 2012, pp. 55–61.

[69] G. Pinto, *A Refactoring Approach to Improve Energy Consumption of Parallel Software Systems*. Informatics Center, Federal University of Pernambuco, 2015.

[70] G. Pinto, F. Soares-Neto, and F. C. Filho, "Refactoring for energy efficiency: A reflection on the state of the art," in *4th IEEE/ACM International Workshop on Green and Sustainable Software, GREENS 2015, Florence, Italy, May 18, 2015*, 2015, pp. 29–35.