EARMO: An Energy-Aware Refactoring Approach for Mobile Apps

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

Abstract—The energy consumption of mobile apps is a trending topic and researchers are actively investigating the role of coding practices on energy consumption. Recent studies suggest that design choices can conflict with energy consumption. Therefore, it is important to take into account energy consumption when evolving the design of a mobile app. In this paper, we analyze the impact of eight type of anti-patterns on a testbed of 20 android apps extracted from F-Droid. We propose EARMO, a novel anti-pattern correction approach that accounts for energy consumption when refactoring mobile anti-patterns. We evaluate EARMO using three multiobjective search-based algorithms. The obtained results show that EARMO can generate refactoring recommendations in less than a minute, and remove a median of 84 percent of anti-patterns. Moreover, EARMO extended the battery life of a mobile phone by up to 29 minutes when running in isolation a refactored multimedia app with default settings (no Wi-Fi, no location services, and minimum screen brightness). Finally, we conducted a qualitative study with developers of our studied apps, to assess the refactoring recommendations made by EARMO. Developers found 68 percent 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 development concerns. One critical concern of mobile apps

- *R. Morales, R. Saborido, F. Khomh, and G. Antoniol are with Polytechynique Montéal, Montreal, QC H3T 1J4, Canada. E-mail: {rodrigo.morales, ruben.saborido-infantes, foutse.khomh}@polymtl.ca, antoniol@ieee.org.*
- *F. Chicano is with the University of Málaga, Málaga 29016, Spain. E-mail: chicano@uma.es.*

Manuscript received 18 Sept. 2016; revised 2 Aug. 2017; accepted 25 Sept. 2017. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Rodrigo Morales.)
Recommended for acceptance by E. Bodden.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2757486

development is reducing energy consumption, 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 consumption 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.

One could argue that reducing energy consumption of an app, and improving traditional quality attributes like readability, flexibility, extendability, reusability do not arise at the same time during the software development process, and it is only in the compiled product that the software engineer is concerned about energy efficiency. However, we surmise automated refactoring as a way to support software developers to write "good" code, so that other developers can understand and maintain easily. The definition of "good" refers not only to traditional quality attributes, but also energy efficiency. Hence, the refactoring operations proposed by an automated approach will have design choices that developers can learn to produce a more energy-efficient code. Once these design choices have been adopted by developers, they can be easily applied to different platforms. If we use a second tool in a later phase (at binary code generation, for example), we run the risk of wrongly assuming that (1) all energy improvements can be performed during compilation phase, and that (2) developers are not responsible of the energy efficiency of their apps, i.e., developers will not consider energy efficiency of apps each time they have to evolve the current design. Consequently, the cost of maintaining two refactoring tools, instead of one that considers energy and software quality in a single phase is expected to be higher.

To address these limitations, we propose a multiobjective refactoring approach called Energy-Aware Refactoring approach for MObile apps (EARMO) to detect and correct anti-patterns in mobile apps, while improving energy consumption. 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 consumption. Our approach leverages information about the energy cost of anti-patterns to generate refactoring sequences automatically. We experimentally evaluated EARMO on a testbed of 20 open-source Android apps extracted from the *F-Droid* marketplace, an Android app repository.

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 consumption of mobile apps. We also propose a methodology for a correct measurement of the energy consumption of mobile apps. Our obtained results provide evidence to support the claim that developer's design choices can improve/decrease the energy consumption of mobile apps.
2) We present a novel automated refactoring approach to improve the design quality of mobile apps, while controlling energy consumption. The proposed approach provides developers the best trade-off between two conflicted objectives, design quality and energy.
3) We evaluate the effectiveness of EARMO using three different multiobjective metaheuristics from which EARMO is able to correct a median of 84 percent anti-patterns.
4) We perform a manual evaluation of the refactoring recommendations proposed by EARMO for 13 apps. The manual evaluation is conducted in two steps. (1) Each refactoring operation in a sequence is validated and applied to the corresponding app. (2) The app is executed in a typical user context and the energy consumption gain is recorded. The sequences generated by EARMO achieve a median precision score of 68 percent. EARMO precision is close to previously published refactoring approaches (e.g., Ouni et al. [19] reports that Kessentini et al. [20] achieves a precision of 62-63 percent and Harman et al. [21]. a precision of 63-66 percent). In addition, EARMO extended the battery life by up to 29 minutes when running in isolation a refactored multimedia app with default settings (no Wi-Fi, no location services, minimum screen brightness).
5) From the manual validation, we provide guidelines for toolsmith interested in generating automated refactoring tools.
6) We perform the evaluation of the design quality of the refactored apps using a widely-used Quality Model (QMOOD) [22] and report a median improvement of 41 percent in extendibility of app's design.
7) We evaluate the usefulness of the solutions proposed by EARMO from the perspective of mobile developers through a qualitative study and achieve an acceptance rate of 68 percent. These results complement the manual verification in terms of precision and design quality (e.g., extendability, reusability), and serve as external evaluation.

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 consumption. 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 [23], 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, the possible number of sequences generated from a list of refactorings is $|S| = \lfloor e \cdot n! \rfloor \forall n \geq 1; |S| = 1, n = 0$. Where $|S|$ is the possible number of refactoring sequences (size of the search space), and n is the number of available refactoring operations (the list of refactoring operations available at the beginning of the search) [24], 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 [25], [26], to evolutionary techniques like genetic algorithm, and multiobjective approaches: e.g., NSGA-II and MOGA [27], [28], [29], [30]; MOCell, NSGA-II, and SPEA2 [31].

Recent works [16], [32] 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 consumption 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 consumption. We target two categories of anti-patterns: (i) anti-patterns that stem from common Object-oriented design pitfalls [33], [34] (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 [32] and in the Android documentation [6], [32] (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 consumption of mobile apps. In the following sections, we explain how we measure and include energy consumption in our proposed approach.

2.2 Energy Measurement of Mobile Apps

Energy consumption, 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. Other works use the LEAP power measurement device [35]. LEAP contains an ATOM processor that runs Android-x86 version 2.x. Its analog-to-digital converter samples CPU energy consumption at a frequency of 10 kHz.

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* and *LEAP*. 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 (I) in a proportional output voltage (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.

A low sampling frequency can make it very hard to assess the energy consumption of any given method. Consider, for example, the *glTron*[3] application. According to our measurements, the method `com.glTron.Video.HUD.draw` has an execution time (inclusive of called methods) of 91.96 milliseconds. Thus, sampling at 125 kHz (one sample each eight microseconds) or 10 kHz (one sample each 0.1 milliseconds) does not make a big difference as enough data points will be collected. However, if we consider for the same package (`com.glTron`) the method `...Video.GraphicUtils.ConvToFloatBuffer`, its execution lasts only 732 microseconds. Measuring at 10 kHz, limits the collection of data points about this method to no more than 7 samples, while measuring at 125 kHz we could collect data points up to 92 samples. In essence, if a method execution last more than one millisecond, such as in `com.glTron.Video.HUD.draw`, the errors will generally averaged out, making the energy estimation error low or even negligible. However, in methods of short duration (less than one millisecond) the error may be higher. Li et al. [36] studied what granularity of measurements is sufficient for measuring energy consumption. They concluded that nanosecond level measurement is sufficient to capture all API calls and methods. This raises another problem, the bottleneck in

1. https://www.msoon.com/LabEquipment/PowerMonitor/
2. http://www.eevblog.com/projects/ucurrent/
3. https://f-droid.org/wiki/page/com.glTron

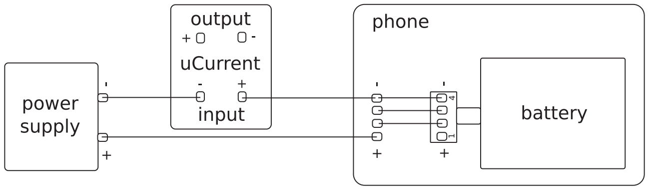

Fig. 1. Connection between power supply and the Nexus 4 phone.

high-frequency power sampling due to the storage system, which cannot save power samples at the same frequency as the power meter can generate them. However, Saborido et al. [37] found that sampling at 125 kHz just accounts for about 0.7 percent underestimation error. Therefore we consider that 125 kHz is sufficient to measure the energy consumption of mobile applications.

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,[4] 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.[5] This application is free and it is available for download in the *Play Store*.[6]

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 multi-objective 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 use of multiobjective algorithms have shown to be useful in finding good solutions in a search space. There is even a procedure called multi-objectivization that transforms a single-objective problem into a multiobjective one, by adding some helper functions [38]. Hence, the use of a multiobjective optimization techniques is suitable to solve the refactoring scheduling problem as they lessen the need for complex combination of different, potentially conflicting, objectives and allows software maintainers to evaluate different candidate solutions to find the best trade.

The set of all non-dominated solutions is called the Pareto Optimal Set and its image in the objective space is called Pareto Front. Very often, the search of the Pareto Front is NP-hard [39], hence researchers focus on finding an approximation set or reference front (RF) as close as possible to the Pareto Front.

As our aim is to improve the design quality of mobile apps, while controlling for energy consumption, 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 [40]. 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 consumption dimensions.

The *Non-dominated sorting genetic algorithm* (NSGA-II) [41] 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) [42] uses variation operators to evolve a population, like NSGA-II, 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 [43], 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 [44]. 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.

4. https://goo.gl/6guUpf
5. The mobile phone has to be rooted first.
6. https://goo.gl/wyUcdD

3 Preliminary Study

The main goal of this paper is to propose a novel approach to improve the design of mobile apps while controlling for energy consumption. To achieve this goal, the first step is to measure the impact of anti-patterns (i.e., poor design choices) on energy consumption. Understanding if anti-patterns affect the energy consumption 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 consumption during a refactoring process. On the other hand, if anti-patterns significantly affect energy consumption, developers and practitioners should be equipped with refactoring approaches that control for energy consumption 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) *Do anti-patterns influence energy consumption?*

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

(PQ2)*Do anti-pattern's types influence energy consumption differently?*

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 consumption 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 [33], [34], and (ii) *Android anti-patterns (AA)* defined by [6], [32]. Concerning (AA), previous works have evaluated the impact on energy consumption of *private getter and setters* [45], [46], [47] and found an improvement in energy consumption after refactoring. Table 1 presents the details of the considered anti-patterns types an the refactoring strategies used to remove them. 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], [32], [33], [34].

To study the impact of the anti-patterns, we write a web crawler to download apps from *F-droid* , an open-source Android app repository.[7] The total number of apps retrieved by the date of April 14th 2016 is 200. These apps come from five different categories (Games, Science and Education, Sports and health, Navigation, and Multimedia). We filtered out 47 apps which Android version is lower than 2.1 because our transformation environment runs Windows 10 which supports Android SDK 2.1 or higher.

From the remaining 153 apps, we take a random sample that was determined using common procedures in survey design, with a confidence interval of 10 percent and a confidence level of 95 percent. Using these values, we obtained that the required sample size is 59 apps. This means that the results we get from our empirical study have an error at most of 10 percent with probability 0.95.

Next, we filtered apps where libraries referenced are missing or incomplete; apps that required to have *username* and *password* for specific websites; apps written in foreign languages and that we could not fully understand their functionality; apps that does not compile; apps that crashed in the middle of execution, or simply would not run in our phone device. The last filter is that the selected apps should contain at least one instance of any of the anti-patterns studied.

After discarding the apps that do not respect 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* [49]. We use ReCon's current implementation for correcting object-oriented anti-patterns, and add two new OO anti-patterns (*Blob* and *Refused bequest*); we also add three Android anti-patterns based on the guidelines defined by Gottschalk [32], and the Android documentation [6]. ReCon supports two modes, root-canal (i.e., to analyze all the classes in the system) and floss-refactoring (i.e., to analyze only the classes related to an active task in current developer's workspace provided by a task management integration plug-in). 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 scenario that exercises the code containing anti-patterns using the Android application *HiroMacro*.[8] This software allows us to generate scripts containing touch and move events, imitating a 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)

7. https://f-droid.org/

8. https://play.google.com/store/apps/details?id=com.prohiro.macro

TABLE 1
List of Studied Anti-Patterns

Type	Description	Refactoring(s) strategy
	Object-oriented anti-patterns	
Blob (BL) [33]	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 [26].
Lazy Class (LC) [34]	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) [34]	A class with one or more methods having a long list of parameters, specially when two or more methods are sharing a long list of parameters that are semantically connected.	*Introduce parameter object (IPO)*. Extract a new class with the long list of parameters and replace the method signature by a reference to the new object created. Then access to this parameters through the parameter object
Refused Bequest (RB) [34]	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) [34]	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.
	Android anti-patterns	
Binding Resources too early (BE) [32]	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* [48] 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[a] 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.

[a] *https://source.android.com/devices/tech/*

TABLE 2
Apps Used to Conduct the Preliminary Study

App	Version	LOC	Category	Description
blackjacktrainer	0.1	3,783	Games	Learning BlackJack
calculator	5.1.1	13,985	Science & Education	Make calculations
gltron	1.1.2	12,074	Games	3D lightbike racing game
kindmind	1.0.0	6,555	Sports & Health	Be aware of sad feelings and unmet needs
matrixcalc	1.5	2,416	Science & Education	Matrix calculator
monsterhunter	1.0.4	27,368	Games	Reference for Monster Hunter 3 game
mylocation	1.2.1	1,146	Navigation	Share your location
oddscalculator	1.2	2,226	Games	Bulgarian card game odds calculator
prism	1.2	4,277	Science & Education	Demonstrates the basics of ray diagrams
quicksnap	1.0.1	18,487	Multimedia	Basic camera app
SASAbus	0.2.3	9,349	Navigation	Bus schedule for South Tyrol
scrabble	1.2	3,165	Games	Scrabble in french
soundmanager	2.1.0	5,307	Multimedia	Volume level scheduler
speedometer	1	139	Navigation	Simple Speedometer
stk	0.3	4,493	Games	A 3D open-source arcade racer
sudowars	1.1	22,837	Games	Multiplayer sudoku
swjournal	1.5	5,955	Sports & Health	Track your workouts
tapsoffire	1.0.5	19,920	Games	Guitar game
vitoshadm	1.1	567	Games	Helps you to make decisions
words	1.6	7,125	Science & Education	Helps to study vocabulary for IELTS exam

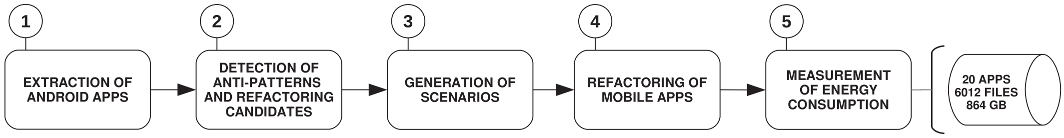

Fig. 2. Data extraction process.

to *Monkeyrunner*format. Thus, the collected actions can be played automatically from a script using the *Monkeyrunner* [50] Android tool. In Table 3 we provide a brief description of each scenario. Note that the scenarios are generated with the main objective of executing the code segment(s) related to the anti-patterns in the original version, and the refactorings applied in the refactored version, and as a disclaimer, many of them may seem trivial, but fit for the purpose of this preliminary study.

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 ensure that a refactored code fragment is executed in the scenario, we first set breakpoints to validate that the debugger stops on it. If this occurs, we build the corresponding apk and check that method invocations to the refactored code appeared in the execution trace. To activate the generation of execution trace file, we use the methods provided in *Android Debug Class* [51], 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 and directly storing the collected results to the personal computer at runtime.

TABLE 3
Description and Duration (in Seconds) of Scenarios Generated for the Studied Apps in Our Preliminary Study

App	Scenario	Duration
blackjacktrainer	Press in {. . .}, then {settings}, and close app.	14.87
Calculator	Make the operation six times five and close app.	17.94
GLTron	Wait until app is loaded and close app.	33.94
kindmind	Press in first category and close app.	21.37
matrixcalc	Fill matrix with number five, press {Calculate}, and close app.	52.47
monsterhunter	Press in {Weapons}, press in first category, select first weapon, press the {+} button, select the {My Wishlist}, press {Ok}, and close the app.	16.39
mylocation	Press the square button, go back, and close app.	15.59
oddscalculator	Wait until app is loaded and close app.	15.72
prism	Wait until app is loaded and close app.	10.84
quicksnap	Wait until app is loaded and close app.	13.8
SASAbus	Wait until DB is downloaded, press {OK} button, wait until maps are downloaded, and close app.	71.72
scrabble	Wait to load board and close app.	35.83
soundmanager	Go to menu, mute/unmute, and close app.	18.74
speedometer	Wait until app is loaded and close app.	13.99
stk	Wait until app is loaded and content downloaded and close app.	35.1
sudoWars	Wait until app is loaded and close app.	10.76
swjournal	Start a workout, filling the two fields, and close app.	28.87
tapsoffire	Press in {Play}, slide down, press over the green color, press {Play}, {API}, {Medium}, and {Play}; close app.	25.96
vitoshadm	Wait until app is loaded and close app.	14.78
words	Wait until app is loaded and close app.	10.75

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 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 Wi-Fi if the app does not require it to be correctly executed, etc.).

Our script starts the oscilloscope and the app, which we modify to generate the execution trace. Both are different files where the first time-stamp is zero.

When users launch an app, the app goes through an initialization process running the methods `onCreate`, `onStart` , and `onResume` . In Fig. 3 we present a simplified flow-chart of the state paths of a single-activity Android app. The app is visible after the `onStart`

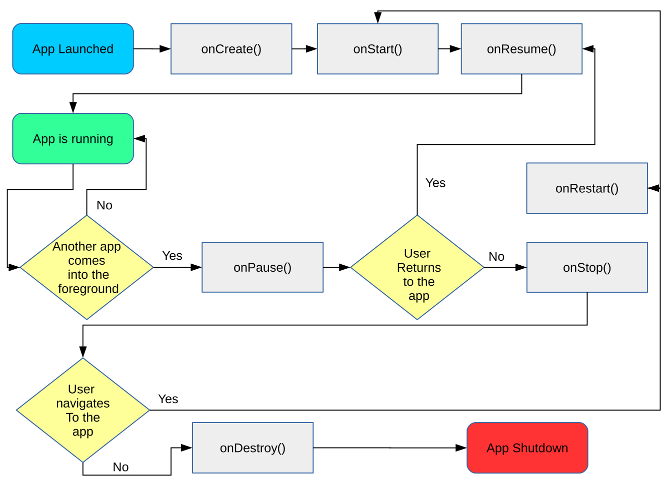

Fig. 3. Android app flow-chart.

method is executed and the user can interact with the app after the `onResume` method is executed. We consider that an Android app is completely loaded after method `onResume` ends. The times reported in Table 3 are the times required to completely load each app and run the corresponding scenario. For all scenarios, the last action of the scenario is to manually close the app, which takes between three and five seconds.

Additionally, the generated execution traces contain, for each method call, global execution times relative to the complete load of apps (whose global time is zero). Based on that we consider the global start time of the method `onCreate` as the instant of time when the execution trace is created once the app is launched.

In order to estimate the existing gap between energy and execution traces we do the following. Once we start the oscilloscope we introduce a timer to measure the time needed to launch an Android app. We consider the difference between this time and the time when the method `onCreate` is executed as the gap between energy and execution traces. For instance, if we consider that an Android app is launched in T seconds and the execution trace is created in instant of time N, the existing gap between the energy and execution trace is calculated as $T - N$. Because for each app's run we know the time required to launch the app and when the method `onCreate` is executed, the gap between traces for each app's run is known.

According to our experiments Android apps are launched in the range of [0.76, 0.92] seconds (average 0.83 seconds = 830000 microseconds) and the method `onCreate` is executed, on average, 0.00009 seconds (90 microseconds) after the app is launched. It means that, in average, the existing gap is (830000-90) = 829010 microseconds. For each apps independent run, energy and execution traces are aligned considering the estimated gap shift.

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. 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 *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. The rationale of removing energy consumption of code that is not inside the package of the app is that we did not detect anti-patterns, neither propose refactoring for those classes. Hence, with the aim of removing noise in our measurements (in case that most of an app's energy consumption is on the library or native functions) we focus on the code that contains anti-patterns, to isolate the effect of applying refactoring on energy consumption. Finally, the median and average energy consumption of each app over the 30 runs is calculated.

Algorithm 1. Steps to Collect Energy Consumption

```
for all runs do
  for all apps do
    Install app in the phone (using adb).
    Start oscilloscope using a script from our test PC.
    Run app (using adb).
    Play scenario (using Monkeyrunner).
    Stop oscilloscope.
    Download execution trace from the phone (using adb).
    Stop app (using adb).
    Clean app files in the phone (using adb).
    Uninstall app (using adb).
  end
end
```

3.3 Data Analysis

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 percent. 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). Do Anti-Patterns Influence Energy Consumption?

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 δ (ES), which indicates the magnitude of the effect size [52] of the treatment on the dependent variable. The effect size is small for $0.147 \leq ES < 0.33$, medium for $0.33 \leq ES < 0.474$, and large for $ES \geq 0.474$ [53].

(PQ2). Do Anti-Pattern's Types Influence Energy Consumption Differently?

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 and Discussion of the Preliminary Study

In Table 4 we present the percentage change in median energy consumption after removing one instance of anti-

TABLE 4
Percentage Change in Median Energy Consumption of Apps After Removing One Instance of Anti-Pattern at Time, Mann—Whitney U Test and Cliff's δ Effect Size (ES)

App	$\gamma(E', E_0)$	*p-value*	*ES*	Magnitude
blackjacktrainer	−0.63	0.2560	−0.15	small
calculator	−1.17	0.1191	−0.25	small
calculator	−0.90	0.4280	−0.10	negligible
gltron	**−1.60**	**2.08E-05**	**−0.70**	**large**
kindmind	0.68	0.2988	0.16	small
matrixcalc	0.56	0.4898	0.09	negligible
monsterhunter	0.50	0.5602	−0.07	negligible
mylocation	−1.56	0.5699	−0.03	negligible
oddscalculator	**−6.01**	**0.0221**	**−0.34**	**medium**
prism	1.50	0.0919	0.17	small
prism	−0.03	0.7151	0.03	negligible
quicksnap	−0.07	0.9515	−0.03	negligible
quicksnap	0.89	0.4898	0.04	negligible
SASAbus	−4.12	0.2286	−0.13	negligible
scrabble	−0.67	0.9838	−0.04	negligible
soundmanager	**−8.38**	**0.0001**	**−0.63**	**large**
soundmanager	**−5.96**	**0.0005**	**−0.53**	**large**
speedometer	**−62.96**	**3.73E-09**	**−0.97**	**large**
stk	0.38	0.5028	0.02	negligible
sudowars	−0.82	0.6408	0.04	negligible
swjournal	−2.21	0.2286	−0.23	small
tapsoffire	−3.52	0.3599	−0.22	small
vitoshadm	**−2.80**	**0.0345**	**−0.29**	**small**
words	**−2.29**	**0.0005**	**−0.44**	**medium**

pattern at time, $\gamma(E', E_0)$. This value is calculated using the following expression

$$\gamma(E', E_0) = \frac{med(E') - med(E_0)}{med(E_0)} \times 100. \quad (1)$$

Where the energy consumption of the app after removing an anti-pattern is represented by E', while the energy consumption of the original app is E_0. $med(E)$ is the median of the energy consumption values of the 30 independent runs. Negative values indicate a reduction of energy consumption after refactoring, positive values indicate an increase of energy consumption. In total, we manually correct 24 anti-patterns inside the set of apps that make up our testbed. In seven instances (i.e., 30 percent) the differences are statistically significant, with Cliff's δ effect sizes ranging from small to large. Specifically, we obtained three apps with large effect size: *speedometer*, *gltron*, and *soundmanager*(two types of anti-patterns); two cases 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 consumption 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 consumption, we present in Fig. 4 the percentage change of the energy consumption after removing each type of anti-pattern studied. For the instances where the results are statistically significant ($p\text{-}value < 0.05$) we add an "$*$" symbol, the exact value and *ES* is shown in Table 4.

Regarding object-oriented (OO) anti-patterns, on top of Fig. 4, we observe that removing *lazy class* reduces energy consumption in *blackJacktrainer*. This trend holds for *tapsoffire* and *soundmanager* respectively, with the latter one having statistically significance and magnitude of the difference (i.e., ES) is large. In the case of *Refused Bequest* , two out of three apps show that removing the anti-pattern saves energy, and the difference is statistically significant for *vitoshadm*. For the *Blob* anti-pattern, all refactored versions report a decrease in energy consumption, though the differences are not statistically significant.

Concerning *Long Parameter list* (LP), and *Speculative Generality* (SG), both report a negative impact on energy

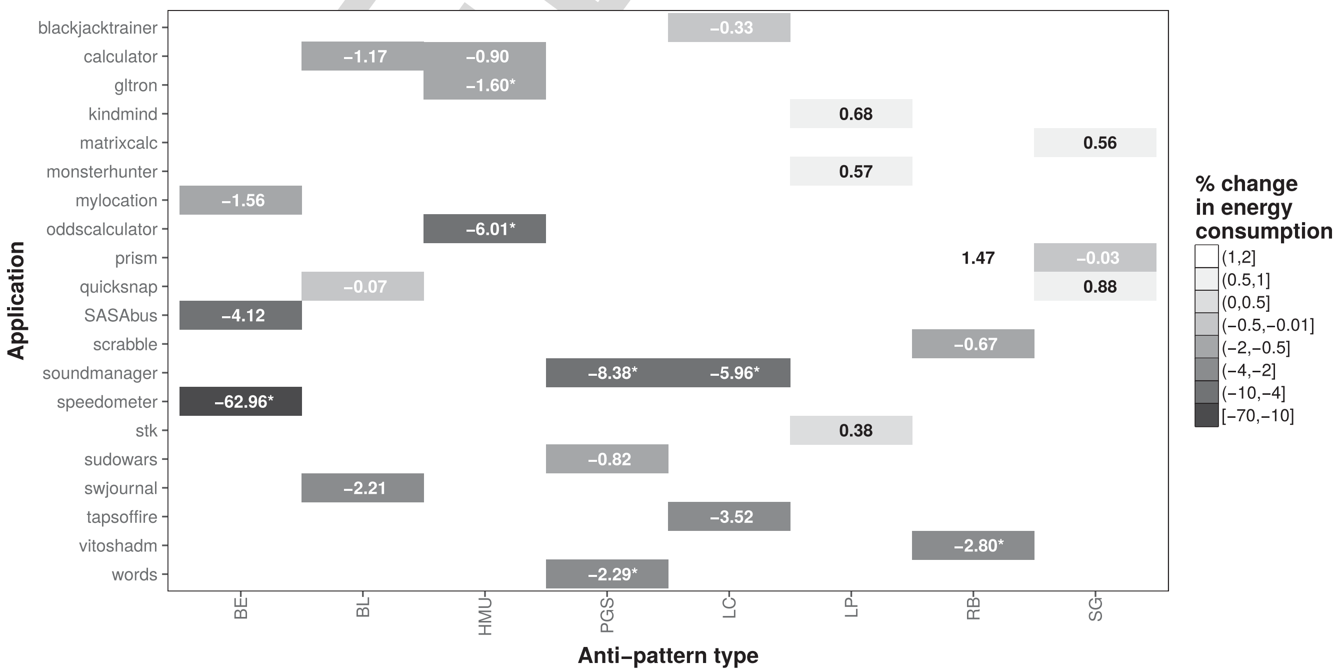

Fig. 4. Percentage change in median energy consumption when removing different types of anti-patterns.

consumption 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 from Android apps, because of their negative effects on energy consumption. Note that the amount of energy saved is influenced by the context in which the application runs. For example, *SASAbus* , which is a bus schedule app, downloads the latest bus schedule at start, 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 [54], 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, there is evidence to show that removing Binding resources too early, Private getters and setters, Refused Bequest, and Lazy class anti-patterns can improve energy efficiency in some cases. We do not find any statistically significant cases were removing an anti-pattern increases energy consumption. Removing Blob, Long Parameter List, and Speculative Generality anti-patterns does not produce a statistically significant increase or decrease.*

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 energy consumption. 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 consumption. Measuring in real-time the energy consumption of a refactoring sequence can be prohibitive, because it requires to apply each refactoring element of the sequence in the code, compile it, generate the binary code (*APK*) and download it into the phone; all of these steps for each time the search-based process requires to evaluate a solution. That is why we define a strategy to estimate the impact of each refactoring operation on energy consumption, based on the results obtained in our preliminary study (Section 3) and without measuring during the search process. The strategy consists of the following steps:

1) We compute the energy consumption of an app using the following formulation:

$$EC(a) = \sum_{m \in M} EC(a_m). \tag{2}$$

Where M is the set of methods in a.

2) We prepare two versions of the same app with and without one instance of an anti-pattern type, and we call them a^{ORI}, and a^k. To isolate possible aggregation effects, we remove only one instance of anti-pattern using the same refactoring operations. For example, if we want to remove a Lazy class, we apply inline class to the class that contained that anti-pattern.

3) The energy consumption coefficient of a refactoring applied to remove an anti-pattern of type k, in app a is calculated using the following expression:

$$\delta EC(a^k) = \frac{med(EC(a^k)) - med(EC(a^{ORI}))}{med(EC(a^{ORI}))}. \tag{3}$$

Where $med(.)$ is the median value of the 30 independent runs for $EC(a^k)$ and $EC(a^{ORI})$. If the value of $\delta EC(a^k)$ is negative, it means that 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) To determine a global refactoring energy coefficient $\delta EC(k)$, we take three apps from our testbed for each type of anti-pattern k. $\delta EC(k)$ is calculated using the following expression:

$$\delta EC(k) = med(\delta EC(a^k)); \forall a^k \in A^k. \tag{4}$$

Where A^k is the set of apps that were refactored to remove a single instance of anti-pattern type k.

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

4.1 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], [33], [34]. In the final step, a multi-objective 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 consumption of the code design. In the following, we describe in detail each of these steps.

4.2 Step 1: Energy Consumption Estimation

This step requires to provide (1) the energy consumption of the app (E_0). Developers can measure E_0 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 [55]. (2) The coefficient $\delta EC(k)$ of each refactoring type analyzed. We derive $\delta EC(k)$ values for each refactoring type based on the results of the preliminary study. EARMO uses this information in the last step to evaluate the energy consumption of a candidate refactoring solution during the search-based process.

4.3 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 consumption. A code meta-model describes programs at different levels of abstractions. We consider three levels of abstractions to model programs. A code-level model (inspired by UML) which includes all of the constituents found in any object-oriented system: classes, interfaces, methods, and fields. An idiom-level model of a program that is a code-level model extended with binary-class relationships, detected using static analysis. A design-level model that contains information about occurrences of design motifs or of code smells and anti-patterns. A code-meta model must differentiate among use, association, aggregation, and composition relationships. It should also provide methods to manipulate the design model and generate other models. The objective of this step is to manipulate the design model of a system programmatically. Hence, the code meta-model is used to detect anti-patterns, apply refactoring sequences and evaluate their impact in the design quality of a system. More information related to code meta-models, design motifs and micro-architecture identification can be found in [56], [57].

4.4 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.

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, scen)
  `/* 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 */`
  RA = 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*, *RA*, E_0)
  /* This is a generic template of the EARMO algorithm that finds the optimal set of refactoring sequences */
Procedure `Generation of an optimal set of refactoring sequences` (AM, RA, E_0)
  $P_0 = GenerateInitialPopulation(RA)$
  $X_0 = \varnothing$
  `/* 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 */`
    $AM' = clone(AM)$
    $apply\_refactorings(AM', S_i)$
    $compute\_Design\_Quality(AM', S_i)$
    $compute\_Energy\_Consumption(AM', S_i, E_0)$
  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**
      $AM' = clone(AM)$
      $apply\_refactorings(AM', S_i)$
      $compute\_Design\_Quality(AM', S_i)$
      $estimate\_Energy\_Consumption(AM', S_i, E_0)$
    end for
    $X_t = Update(X_t, P_t)$
end while
$best\_solution = X_t$
return best_solutions

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

TABLE 5
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[])

"pkg" is the package name of an app.

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 [58]. 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.

4.5 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 consumption 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.

4.5.1 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 5 we present an example of a refactoring sequence. The ID is used to identify whether a RO already exists 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 *inline class*.

4.1.2 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 [59].

4.1.3 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 Fig. 5.

For mutation, we consider the same operator used in our previous work [31] 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 valid refactoring operations to add (i.e., that do not cause conflict with the existent ones in the sequence). We provide an example in Fig. 6.

4.1.4 Fitness Functions

We define two fitness functions to evaluate the quality and the energy consumption 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. [28]. We follow this

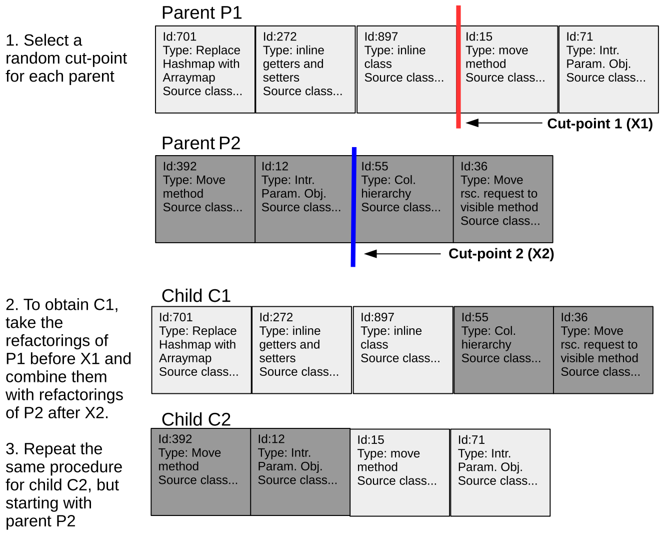

Fig. 5. Example of cut and slice technique used as crossover operator.

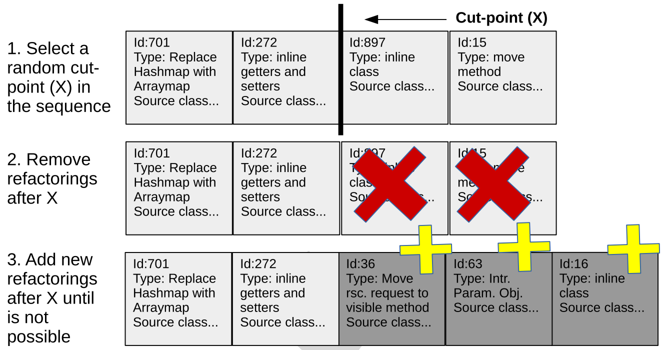

Fig. 6. Example of the mutation operator used.

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 6-29). The process starts by generating an initial population of refactoring sequences from the code 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 11-13). The next step is to extract the non-dominated solutions (line 15). From line 20 to 25, 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. The stopping criterion, which is defined by the software maintainer, is a fixed number of evaluations. Finally, in lines 28-29, 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 consumption. The *quality focus* is the improvement of the design quality and energy consumption 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 consumption. 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* [60]. 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. As previously discussed in Section 3.2, 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 Fig. 7.

Replace HashMap with Array Map (RHA). *ArrayMap* is a light-weight-memory mapping data structure included since Android API 19. The refactoring consists 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 are required for this refactoring, as depicted in Fig. 8.

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. This is useful when both classes are very similar, or the child class does not add extra functionality, but was introduced presumably for handling future enhancements that never occurred. In Fig. 9 we

```java

private SplashView splashView;

private SplashView getSplashView() {
        return splashView;
    }
//This setter is not even used!
 private void setSplashView(SplashView splashView) {
        this.splashView = splashView;
    }

 public void initialize() {
        final boolean firstLaunch = isFirstLaunch();

        if (firstLaunch) {
            getSplashView().showLoading();
        }
        ...
        getSplashView().renderImportError();
        ...
        getSplashView().renderSplashScreenEnded();
            }
       ...

        getSplashView().renderFancyAnimation();
    }
```

```java
private SplashView splashView;

// We inline private getters and setters
 public void initialize() {
        final boolean firstLaunch = isFirstLaunch();

        if (firstLaunch) {
           splashView.showLoading();
        }
        ...
        splashView.renderImportError();
        ...
        splashView.renderSplashScreenEnded();
            }
       ...

        splashView.renderFancyAnimation();
    }
```

Fig. 7. Example of *inline private getters and setters* refactoring. Original code on the top, and refactored code on the bottom.

provide an example of SG anti-pattern found in *Calculator* app. 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

```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>();
                ...
        }
}
```

Fig. 8. Example of replacing HashMap with ArrayMap refactoring. Original code on the top, and refactored code on the bottom.

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 Fig. 10, an example of IPO in a method 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 Fig. 11.

Move Method (MM). This refactoring is applied to decompose a *Blob* class using *move method* and it is originally proposed by Seng et al. [26]. 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.

5.1 Descriptive Statistics of the Studied Apps

Table 6 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

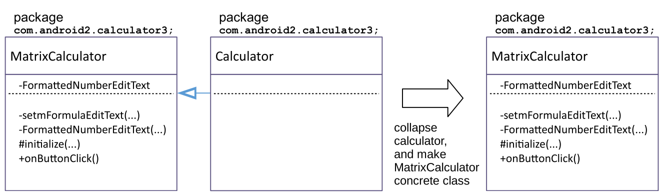

Fig. 9. An example of SG in *Calculator* . Original class diagram on the left, and refactored class diagram on the right.

```
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);
        ...
        }
```

```
public static Uri addImage(AddImageParameter parObj) {
        OutputStream outputStream = null;
        String filePath = parObj.directory + "/" + parObj.filename;
        ...
        if (parObj.source != null) {
                parObj.source.compress(CompressFormat.JPEG,
                parObj.degree[0] = 0;
                CameraApplication.JPEG_HIGH_QUALITY, outputStream);
        } 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);
        ...
}
```

Fig. 10. Example of introduce parameter object refactoring. Original code on the top, and refactored code on the bottom.

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 consumption 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 consumption?*

This research question aims to assess the effectiveness of EARMO at improving design quality, while reducing energy consumption.

(RQ2) *What is the precision of the energy improvement reported by EARMO?*

This research question aims to examine if the estimated energy improvements reported by EARMO reflect real measurements.

(RQ3) *To what extent is design quality improved by EARMO according to an external quality model?*

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 [22] 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.

(RQ4) *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.

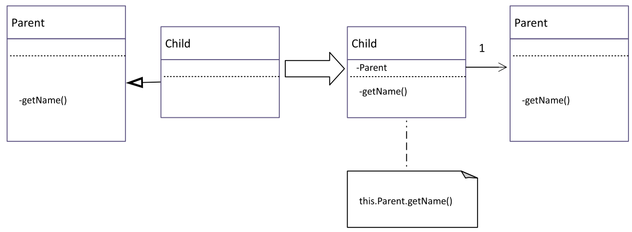

Fig. 11. An example of applying RIWD in a class. Original class diagram on the left, and refactored class diagram on the right.

5.3 Evaluation Method

In the following, we describe the approach followed to answer *RQ1* , *RQ2* , *RQ3* and *RQ4*.

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:

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

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.

TABLE 6
Descriptive Statistics Showing Anti-Pattern Occurrences in the Studied Apps

		O.O. AP					Android AP		
App	NOC	BL	LC	LP	RB	SG	BE	HMU	PGS
Calculator	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

- Estimated energy consumption improvement (EI). EI is computed using the following formulation:

$$EI(a) = \frac{EC(a') - EC(a)}{EC(a)} \times 100. \tag{6}$$

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 EI expresses an increment $(+)$/ decrement $(-)$ and the value represents the amount in percentage. High negative values are desired.

The *independent variables* are the three selected *EMO* metaheuristics, i.e., MOCell, NSGA-II, and SPEA2. We choose them because they are well-known evolutionary techniques that have been successfully applied to solve optimization problems, including refactoring [28], [61]. We implement all the metaheuristics used in this study using the jMetal Framework [62], 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 stopping criterion, population size, and the probability of the variation operators. We use number of evaluations as the stopping criteria. As the maximum number of evaluations increase, the algorithm obtains better quality results on average. The increase in quality is usually very fast when the maximum number of evaluation is low. That is, the slope of the curve quality versus maximum number of evaluations is high at the very beginning of the search. But this slope tends to decrease as the search progresses. Our criterion to decide the maximum number of evaluations is to select a value for which this slope is low enough. In our case *low enough* is when we observe that no more anti-patterns are removed after that number of evaluations. We empirically tried different number of evaluations in the range of 1,000 to 5,000 and found 2,500 to be the best value.

For selection operator we use the same operator defined by Deb et al. [41] for NSGA-II, and *binary tournament* for the other EAs, which are the default operators used in *JMetal* for these algorithms.

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.

TABLE 7
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

With respect to energy estimation, we show in Table 7 the energy consumption coefficient $\delta EC(k)$ for each refactoring type, that we use in our experiment. These coefficients 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* (i.e., *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) [63]* and *SPREAD [41]*.

We also perform Whitney U Test test pair-wise comparisons between the three algorithms to validate the results obtained for these two performance indicators.

For *RQ2*, we perform an energy consumption validation experiment to evaluate the accuracy of EARMO using our measurement setup described in Section 2.2. This is important to observe how close is the estimated energy improvement (i.e., EI) compared to the real measurements. 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 a typical usage scenario, and compute the difference between the obtained values. We compare the obtained result with EI.

For *RQ3*, we use the Quality Model for Object-Oriented Design (QMOOD) [22] 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 [25], [64], which allows for a replication and comparison of the obtained results.

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 8. More details about the metrics and quality attributes can be found in the original source [22]. In this work we do not consider the functionality quality attribute because refactoring being

TABLE 8
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.

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:

$$QG(A_y) = \frac{A_y(a') - A_y(a)}{|A_y(a)|} \times 100. \quad (7)$$

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 *RQ4* , 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 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 four research questions that aim to evaluate EARMO.

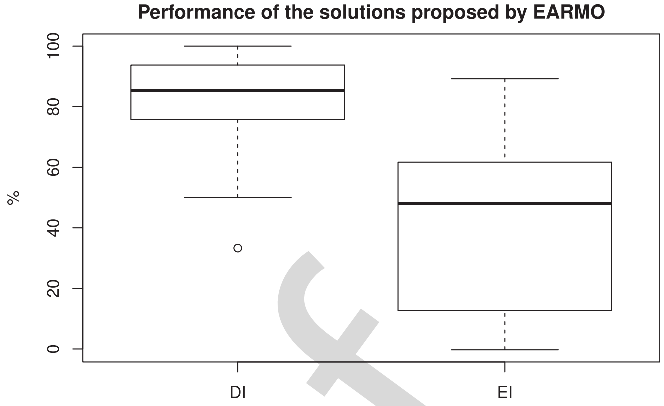

Fig. 12. Distribution of anti-patterns and energy consumption reduction in the studied apps.

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

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 Fig. 12, we present the distribution of DI and EI metric values, for each solution in the Pareto reference front. Fig. 12 highlights a median correction of 84 percent of anti-patterns and estimated energy consumption improvement of 48 percent. To provide insights on the performance of EARMO, we present, in Table 9, 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

TABLE 9
Minimum and Maximum Values (%) of DI and EI Obtained for Each App After Applying EARMO

App	Solutions	DI Min.	DI Max.	EI Min.	EI 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

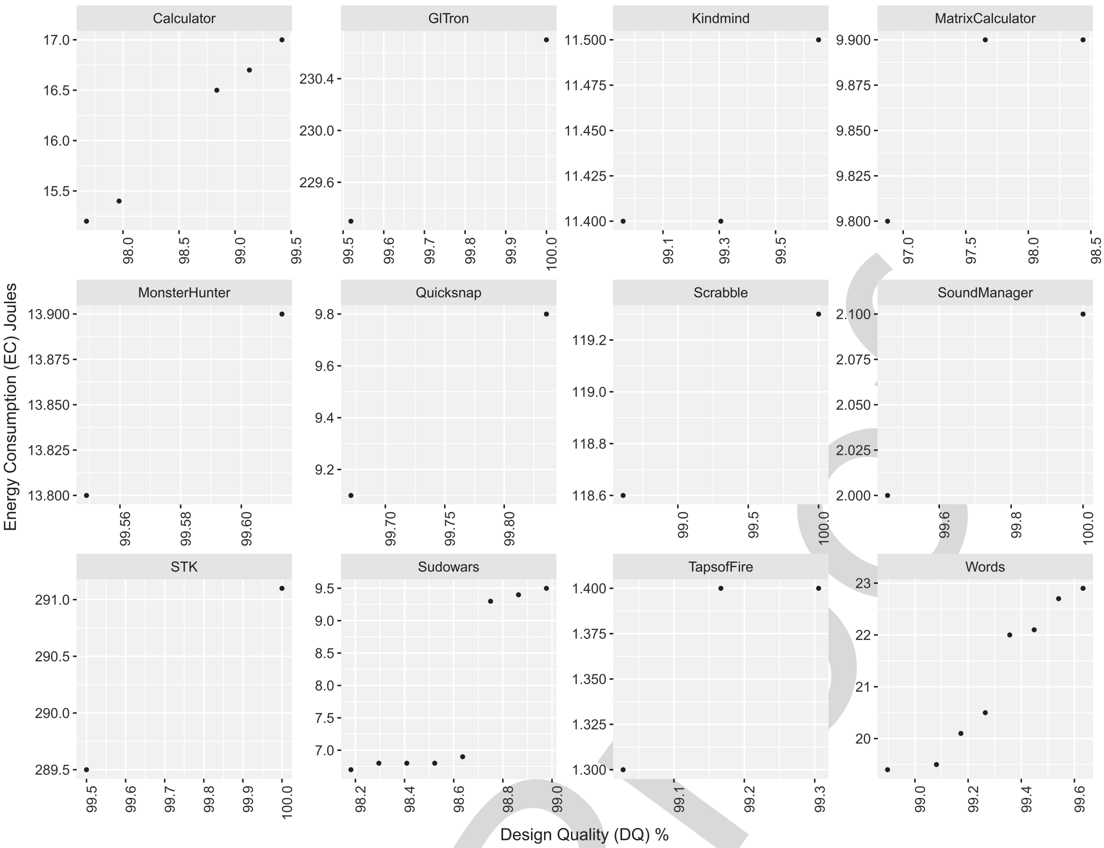

Fig. 13. Pareto front of apps with more than one non-dominated solution.

achieved a compromise in terms of design quality and energy consumption. Table 9 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 percent of anti-patterns correction with a maximum EI of 89 percent. 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 for DI too, except for the apps with more than two solutions (i.e., *Calculator* and *Words*). We observe that more than 65 percent of the apps contain more than one solution. To have an insight on those apps, we present in Fig. 13 the Pareto Front (PF) for each app, where each point represents a solution with their corresponding values, DQ (x-axis) and EC (y-axis). The most attractive solutions are located in the bottom right of the plot.

According to the concept of dominance, every Pareto point is an equally acceptable solution of the multiobjective optimization problem [65], but developers might show preference over the ones that favors the metric they want to prioritize. They could select the refactorings that improve more the energy consumption (e.g., they can chose to correct more Android anti-patterns), or apply more OO refactorings to improve the maintainability of their code. 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 10, 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.

TABLE 10
Median Values of Anti-Patterns Corrected by Type (%)

	O.O. anti-patterns					Android anti-patterns		
App	BL	LC	LP	RB	SG	BE	HMU	PGS
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

TABLE 11
Hypervolume

	MOCell	NSGAII	SPEA2
calculator	$1.32e-1_{8.3e-2}$	$8.92e-02_{1.3e-1}$	$9.47e-2_{1.8e-1}$
gltron	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
kindmind	$0.00e+0_{1.0e-1}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
matrixcalculator	$2.50e-1_{0.0e+0}$	$2.50e-1_{0.0e+0}$	$2.50e-1_{0.0e+0}$
monsterhunter	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
prism	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
quicksnap	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
scrabble	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
soundmanager	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
stk	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
sudowars	$4.25e-1_{1.3e-1}$	$4.95e-1_{1.2e-1}$	$5.45e-1_{1.2e-1}$
tapsoffire	$0.00e+0_{0.0e+0}$	$0.00e+0_{3.7e-2}$	$0.00e+0_{3.7e-2}$
words	$3.00e-1_{5.3e-2}$	$2.69e-1_{7.3e-2}$	$2.73e-1_{7.0e-2}$

Median and IQR.

TABLE 12
Spread

	MOCell	NSGAII	SPEA2
calculator	$6.89e-1_{3.0e-1}$	$1.12e+0_{4.7e-1}$	$8.73e-1_{5.6e-1}$
gltron	$6.78e-1_{1.8e-1}$	$1.07e+0_{1.8e-1}$	$1.08e+0_{2.7e-1}$
kindmind	$6.93e-1_{1.0e-1}$	$9.71e-1_{2.2e-1}$	$7.66e-1_{3.0e-1}$
matrixcalculator	$5.00e-1_{0.0e+0}$	$1.39e+0_{0.0e+0}$	$1.49e+0_{3.5e-3}$
monsterhunter	$8.97e-1_{4.3e-1}$	$9.70e-1_{2.1e-1}$	$9.27e-1_{1.1e-1}$
prism	$0.00e+0_{0.0e+0}$	$1.94e+0_{3.8e-2}$	$1.92e+0_{4.6e-2}$
quickSnap	$1.95e-1_{4.1e-1}$	$1.29e+0_{6.0e-1}$	$1.00e+0_{1.4e+0}$
scrabble	$5.00e-1_{1.0e+0}$	$1.50e+0_{3.8e-1}$	$1.62e+0_{7.8e-1}$
soundmanager	$1.00e+0_{1.7e-1}$	$1.00e+0_{0.0e+0}$	$1.00e+0_{0.0e+0}$
stk	$0.00e+0_{0.0e+0}$	$1.95e+0_{2.9e-02}$	$1.91e+0_{1.5e-1}$
sudowars	$7.96e-1_{1.3e-1}$	$8.53e-1_{1.4e-1}$	$8.41e-1_{1.3e-1}$
tapsoffire	$7.53e-1_{5.4e-1}$	$1.00e+0_{1.7e-1}$	$1.00e+0_{8.6e-2}$
words	$6.84e-1_{2.5e-1}$	$9.42e-1_{2.2e-1}$	$7.07e-1_{1.6e-1}$

Median and IQR.

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

> *We conclude that including energy-consumption 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*) [63] is a quality indicator often used for this purpose, and higher values of this metric are desirable.

In Table 11 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. SPEA2 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* [41]. 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 12 we present the median and IQR results of the Spread indicator. We observe that MOCell outperforms the other techniques in 92 percent (12 apps) of cases, while *soundmanager* reports the same value for the three *EMO* s. SPEA2 gets the second best in 69 percent (nine apps), and NSGAII only in 8 percent (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 Whitney U Test, with a confidence level of 95 percent. The results of these tests are summarized in Table 13. We introduce a special notation to facilitate the comprehension of the results. The ▲ symbol in a column indicates that the metaheuristic in the left 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.

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 Whitney U Test test, MOCell is the best performing technique with respect to solution diversity, while regarding HV the performance of the three *EMO*

TABLE 13
Pair-Wise Whitney U Test 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

algorithms is similar. Developers and software maintainers should consider using MOCell when applying EARMO.

RQ2: What is the precision of the energy improvement reported by EARMO?

The output of EARMO is a sequence of refactorings that balances anti-pattern correction and energy consumption. Developers select from the Pareto front, the solutions that best fits their needs. To validate the estimations of EARMO, we play the role of a software maintainer who wants to prioritize the energy consumption of his/her app over design quality.

The process of validation consists in manually applying the sequence of refactorings to their corresponding source code, for each of the studied apps. We ran the scenario after applying each sequence to ensure that we are not introducing code regression. Finally, we compile and generate the APK file to deploy it in the mobile device and measure their energy consumption using our experimental setting described in Section 3. With this *EC* validation, we want to estimate EARMO's median error with respect to real measurements.

Concerning the scenarios used for *EC* validation, we defined new ones for the apps where we consider that the scenario used in the preliminary study do not reflect a typical usage. The reason is that in the preliminary study we were only interested in executing the code segment related to an anti-pattern instance in the original version and its corresponding refactored code segment. The scenarios of Table 3 were just designed to check if a correlation exists between energy consumption and anti-pattern occurrences. Some scenarios designed for the preliminary study just required to start the app, wait certain seconds, and close it to execute the refactored code segment. For the *EC* validation we want to reflect the actions that a user typically will perform with an app, according the purpose of their creators, instead of scenarios designed to maximize other metrics like coverage which do not reflect the daily use of normal users. To validate EARMO (and perform optimization) we replace the scenarios in Table 3, i.e., the ones that only load and close an app, by the ones presented in Table 14. Note that in some cases we have to modify the code to remove any sources of randomness that may alter the execution path between different runs. For example, *Sudowars* is a sudoku game where the board is randomly generated. Because in the scenario we introduce fixed numbers in fixed positions of the board, we need to ensure that the same board is always displayed to produce the same execution path over the 30 independent runs. Hence, we fixed the random seed used in the app to force to display always the same board. A similar case happens to another board game (*scrabble*).

For the manual application of the sequence of refactorings, two of the authors of this work (PhD. candidates with more than 5 years of experience in Java), and an intern (MsC. Student with two years of programming experience) worked together. After each team member finished to apply a refactoring sequence to an app, we shared the control version repository with the other team members for approval. In case of disagreement, we vote for either apply or exclude a refactoring operation(s) from a sequence. Additionally, 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 git repositories containing the refactored versions available online at http://swat.polymtl.ca/rmorales/EARMO/.

TABLE 14
Description of Scenarios Generated for the *EC* Validation and Duration (in Seconds)

App	Scenario	Duration
Calculator	Same scenario as preliminary study.	17.94
GLTron	Tap screen to start the game and wait until the moto crashes.	40.08
kindmind	Select each category, wait for the relaxation message, and close app.	80.06
monsterhunter	Same scenario as preliminary study.	16.39
oddscalculator	Select two players, {7 heart}, {8 heart}, {9 heart}, tap {calculate}, wait five seconds, and close app.	45.83
quicksnap	Take a picture and close app.	16.30
SASAbus	Same scenario as preliminary study.	71.72
scrabble	Assign the first four letters to the first cells, tap {confirm}, and close app.	65.11
soundmanager	Same scenario as preliminary study.	18.74
stk	Wait until content is downloaded, tap {karts}, tap first row, back, back, tap {tracks}, tap first row, and close app.	86.55
sudowars	Wait until app is loaded, tap {manual}, tap {single player}, tap {tick} button, select first square and write values 1, 2, 3, 4, 5, and 6, tap {...}, tap {assistant}, give up, tap yes, tap back, close app.	53.13
tapsoffire	Same scenario as preliminary study.	25.96
words	Select a category, tap {play}, tap {flash card}, tap {green hand}, tap {flash card}, tap {red hand}, tap {back}, and close app.	57.34

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 *move method* refactoring, and in *replace inheritance with delegation* , where it is possible to introduce regression despite the fact that the refactoring is semantically correct. Due to the absence of a test suite, we execute the defined scenario on the phone after applying each refactoring, to validate the correct execution of it. This is crucial, because an app could be executed even if the refactoring applied introduces regression until we exercise the functionality related to the code fragment touched by the refactoring. When we notice that the refactoring is not exercised in the defined scenario, we separately test that functionality.

In Table 15 we present the results of the manual refactoring application. The column *Discarded ref.* is the number of refactorings discarded from the sequence; *Applied ref.* the refactorings applied, and *Total* is the sum of both columns. *Precision* is the ratio of refactorings generated over the valid refactorings. Overall, EARMO shows a good precision score (68 percent) for all apps. In fact, only in 20 percent of the apps, the precision is less than 50 percent. From these apps, we discuss Prism, which is the app with lowest precision score. We found one out of five refactorings to be valid, and that one is the *IGS* type; three refactorings attempt to inline autogenerated classes from Android build system (e.g., `R`, `BuildConfig`); one attempts to inline a class that extends

TABLE 15
Summary of Manual Refactoring Application for the EC Validation

App	DI%	EI%	Discarded ref.	Applied ref.	Total	Precision (%)
Calculator	75	54	19	45	64	70
BlackJackTrainer	75	6	3	1	4	25
GlTron	94	26	19	13	32	41
Kindmind	80	19	7	23	30	77
MatrixCalculator	33	1	0	1	1	100
MonsterHunter	82	44	29	83	112	74
mylocation	100	2	1	1	2	50
OddsCalculator	100	15	0	6	6	100
Prism	86	9	4	1	5	20
Quicksnap	92	85	69	119	188	63
SASAbus	82	27	3	8	11	73
Scrabble	86	13	0	6	6	100
SoundManager	94	36	3	5	8	63
SpeedoMeter	100	6	1	1	2	50
STK	83	12	2	3	5	60
Sudowars	71	64	38	75	113	66
Swjournal	100	6	13	6	19	32
TapsofFire	83	89	21	139	160	89
Vitoshadm	100	4	0	2	2	100
Words	75	63	23	76	99	77
			Total	614	Median	68

from `android.app.Activity` class which is not invalid. From the four refactorings discarded of *Prism*, three can be consider valid but useless, and only one will introduce regression. Later, we provide guidelines for toolsmiths interested in developing refactoring tools for Android. With respect to the total number of refactorings applied, we observe that in seven cases we apply more than 20 refactorings, and from this subset two of them require more than 100. This validate our idea, that an automated approach can be useful for developers and software maintainers interested in improving the design of their apps, but with limited budget time to perform a dedicated refactoring session for all classes existing in their app.

In Table 16 we present EARMO median execution time $Exec.Time$, estimation values of energy consumption EC, median energy consumption of an app before (E_0) and after (E') refactoring. The difference between EC and E', $\gamma(EC, E')$ is calculated by subtracting $EC - E'$ and dividing the result by E' and the result is multiplied by 100. Similarly, we calculate the difference between E' and E_0, $\gamma(E', E_0)$. From the statistical tests between E_0, E', the $p-$value, and *effect size* (ES). The last column is the median difference of battery life duration, in minutes, between the original and the refactored version ($Diff.Batterylife$). This is of special interest for software maintainers to assess if the impact of applying a refactoring sequence would be noticeable to end users. We provide details of how to compute the last column below. This procedure has been used in previous works [66].

For each app we calculate its battery usage (in %) using Equation (8) to estimate the percentage of battery charge that is consumed by an app when running the defined scenario. E is the energy consumption in Joules of an app (derived from the median of the 30 independent runs), and V and C are the voltage and electric charge (in mAh), respectively, of the phone battery. For Nexus 4, $V = 3.8$ and $C = 2100$ mAh

$$Battery_{usage} = \frac{E}{V} \times \frac{1000}{C \times 3600} \times 100. \quad (8)$$

After obtaining the battery usage for both versions (original, and refactored) of each app, we use it to compute the battery life (in hours) using Equation (9) where ET is the execution time of the app (in seconds). We consider the battery life of an app to be the time that it takes to drain the battery if the scenario associated to the app is continuously run

$$Battery_{life} = \frac{(ET \times 100)/Battery_{usage}}{3600}. \quad (9)$$

Finally, we calculate the average battery life for each app (original and refactored) and subtracted these values to obtain the difference of battery life ($Diff.Batterylife$). Positive values are desired, as they mean that the battery life is longer using the refactored version, while negatives values mean the opposite effect.

Note that we did not consider apps in the validation where the number of refactorings applied is one, that accounts for six apps. The reason is that for these apps the energy improvement estimation EI is inferior to 10 percent before the manual application of refactorings, so we do not

TABLE 16
EARMO Execution Time (Seconds), EC Estimation (J), Median Energy Consumption E_0 and E' (J), γ Values, Statistical Tests, and Difference in Battery Life (Minutes)

App	$Exec.Time$	EC	E_0	E'	$\gamma(EC, E')$	$\gamma(E', E_0)$	$p-value$	ES	$Diff.Batterylife$
calculator	154.90	17.40	21.28	19.49	−11%	−8%	**1.86E-09**	−0.94	2.55
gltron	55.98	242.27	256.44	252.15	−4%	−2%	**8.01E-08**	−0.77	0.42
kindmind	34.59	17.10	18.72	18.9	−10%	1%	0.1294	0.21	−4.61
monsterhunter	237.10	13.63	16.07	16.05	−15%	0%	0.6263	−0.03	−0.82
oddscalculator	8.98	29.25	30.61	30.94	−5%	1%	0.1094	0.22	−2.06
quicksnap	418.82	11.52	15.33	15.29	−25%	0%	0.9193	−0.04	3.33
SASAbus	32.39	3.79	4.61	5.49	−31%	19%	0.7922	0.09	−2.03
scrabble	18.55	88.68	94.56	94.14	−6%	0%	0.9193	−0.03	2.45
soundmanager	25.70	1.75	1.96	2.00	−13%	2%	0.3492	0.16	1.88
stk	24.58	240.82	252.81	249.29	−3%	−1%	0.1403	−0.16	0.99
sudowars	203.60	46.21	54.27	53.99	−14%	−1%	0.0879	−0.20	1.07
tapsoffire	281.00	3.30	6.80	6.59	−50%	−3%	0.9354	−0.02	1.97
words	119.65	25.16	27.01	25.13	0%	−7%	**0.0384**	−0.27	29.71

expect a measurable energy consumption change. In addition, we also omit *Swjournal* , in which we applied six refactorings out of 13, but given its low EI of 6 percent it is unlikely to report a noticeable change either.

For the remaining 13 apps, we observe that the median execution time for generating the refactoring sequences is less than a minute (56 seconds). Concerning energy estimation (EC), the direction of the trend holds for all the apps in the testbed according to the results measured E'. Concerning the accuracy of the estimation, EARMO values are more optimistic than the actual measurements but in an acceptable level. There are some remarkable exceptions, like *Tapsoffire* where the difference is 50 percent. In this app, most of the refactorings are *move method* type (120). If we multiply 120 by E_0, and the result by $\delta EC(move\,method)$ we have an energy consumption decrease of -1.64 J; 12 refactorings of *inline private getter and setters* type that account for -1.92 J. These two refactorings consume in total 3.56 J. The rest of the energy is divided between six *IPO* and one *replace HashMap with ArrayMap*. However, the impact on energy for this app is far from this value, probably because the scenario does not exercise (enough) the code that is modified by the refactorings to report a considerable gain. On the other side, *STK* reports the most close prediction with a difference of 3 percent. The refactorings applied are three *inline getter and setters*. If we compare the results obtained by EARMO compared with the preliminary study, the energy consumption trend holds for all the apps. However it is hard to make a fair comparison because in the Preliminary study we measure the effect of one instance of each anti-pattern type at a time, but in the energy consumption validation of EARMO we apply few to several refactorings. Although we assume that the effect of refactoring is aggregated, it is difficult to prove it with high precision, since we could not exercise all possible paths related to the refactored code in the proposed scenarios. Yet, the median error of $\gamma(EC, E')$ is in acceptable level of 12 percent, like the one reported by Wan et al. [67], when estimating the energy consumption of graphic user interfaces in a testbed of 10 apps.

Concerning the difference in energy consumption after refactoring, we observe that for three apps we obtain statistical significant results, with large effect size (results are in bold). This corroborates the findings in the preliminary study, for these apps. Although, for the rest of the apps the results are not statistically significant, we still we believe that the results are sound with respect to the energy consumption improvements reported. A recent work by Banerjee reported an energy consumption improvement from 3 to 29 percent in a testbed of 10 *F-Droid* apps with an automated refactoring approach for correcting violations of the use of energy-intensive hardware components [68]. With respect to battery life, EARMO could extend the duration (for the apps where the difference is statistically significant) of the battery from a few minutes up to 29 minutes (see the remarkable increment reported for *Words*). Note that to obtain a similar outcome in battery life, the proposed scenarios should be executed continuously, draining the battery from full to empty, which is not impossible, but rather unlikely. Yet, the benefits of improving design quality of the code, and potentially reducing the energy consumption of an app should not be underestimated. Not only because battery life is one of the main concerns of Android users and every small action performed to keep a moderate energy usage in apps is well appreciated. But, even if there is not a noticeable gain in energy reduction, software maintainers are safe to apply refactoring recommendations proposed by EARMO without fearing to introduce energy leaks.

Guidelines for toolsmiths designing refactoring recommendation tools.

We discuss some issues that should be considered for toolsmiths interested in designing refactoring recommendation tools for Android based on our experience applying the suggestions generated by EARMO. We should note that the tool that we use for detecting anti-patterns, which is *DECOR*, is not developed for Android platform. Hence, it does not consider the control flow depicted in Fig. 3 and the *OS* mechanisms of communication between apps. This could generate false positives and consequently impact the generation of refactoring opportunities. Toolsmiths interested to develop refactoring tools for mobile platforms, based on anti-pattern detection tools aimed to target *OO*, should adapt the detection heuristics to avoid generating invalid refactoring operations. We discuss some strategies to consider below.

Excluding Classes Autogenerated by Android Build System. The classes `<app package>.R`, and `<app package>.BuildConfig` should not be considered for analysis of anti-patterns as they are automatically generated when (re) building an app.

Classes Extending Classes from `android.content` *Package and its Corresponding Subpackages*. This package provides classes for accessing and publishing data on a mobile device and messaging between apps. As an example, consider `android.content.BroadcastReceiver`, which allows an app to receive notifications from relevant events beyond the apps flow, e.g., a user activating the airplane mode. An app can receive broadcasts in two different ways. (1) declaring a *broadcast receiver*in the apps manifest; (2) creating an instance of class `BroadcastReceiver`, and register within a context [69]. We focus in the first method, as is the one that could lead developers to introduce regression (even using IDEs refactoring tool support). In *manifest-declared* receivers, the receiver element is registered in the apps manifest, and a new class is extended from `BroadcastReceiver`which requires to implement `onReceive(context, Intent)` method, to receive the contents of the broadcast. Let us briefly discuss the main issue when generating refactoring opportunities for classes extending from android.content packages (in this example we focus in `BroadcastReceiver`) depending on the type of refactoring to be applied. *Collapse hierarchy refactoring* is not considered as `BroadcastReceiver` does not belong to the apps package. *Replace inheritance with delegation* will introduce regression when removing the hierarchy relationship with `BroadcastReceiver`. We observe the same issue with *inline class* when trying to move the methods and attributes to other potential class. *Move method*will introduce regression too, when trying to move inherited methods like `onReceive`to another class.

Collapsing Hierarchy of Classes Registered as Android Activity. When a refactoring operation consists of applying *Collapse hierarchy refactoring* to a class that extends from

TABLE 17
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 for all PF solutions	**−1.24**		**−0.94**		**−4.07**		**3.14**		**40.78**	

`Activity`, it is also necessary to update the apps manifest with the name of the parent class.

Replacing Hashmap with ArrayMap. It is necessary to replace the imports for `android.support.v4.util` when Android API is less than 19, or `android.util` otherwise. It is important to mention that `ArrayMap` is defined as final, so it limits the possibility to derive a new implementation from this class, contrary to `HashMap` and its derived classes (e.g., `LinkedHashMap`).

RQ3: To what extent is design quality improved by EARMO according to an external quality model?

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 17 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.

- *Reusability, understandability and flexibility*. In general, the refactored apps report a slight decrease that ranges from 0.9 to 4 percent for these attributes. In the case of *reusability*, the *prism* app is an outlier, with a medium deterioration of *reusability* between 31 and 44 percent. EARMO finds two refactoring sequences (or two non-dominated solutions in the Pareto front) that are comprised of five refactoring operations. These refactorings are three *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 percent (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 five attributes studied. Finally, for *flexibility* we report a median of −4.07 percent. One remarkable case is *Mylocation*, with 100 percent 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 nine classes). Similarly, *Oddscalculator* contains one solution with seven *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 percent.
- *Effectiveness*. We report a small gain of 3.14 percent, with two outliers (*Oddscalculator* and *Soundmanager*). As we discussed before, *Oddscalculator* is mainly composed of *inline class* refactorings. *Soundmanager* has two solutions, both contain nine *inline classes*, six *inline getters/setters*, and two *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.

TABLE 18
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

- *Extendibility.* For this attribute we report a considerable improvement of 41 percent. 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.

RQ4: 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, identified as authors in the repository of the apps, were contacted but only 8 responded providing feedback for a total of 8 apps. Table 18 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 percent of them considered refactorings to be useful but only 12 percent 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 18, 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 Fig. 14, 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 percent of the refactorings suggested by EARMO were accepted. For three other apps (*calculator*, *kindmind*, *oddscalculator*) the acceptance ratio range from 40 to 57 percent. The developer of the *GLTron* app rejected all

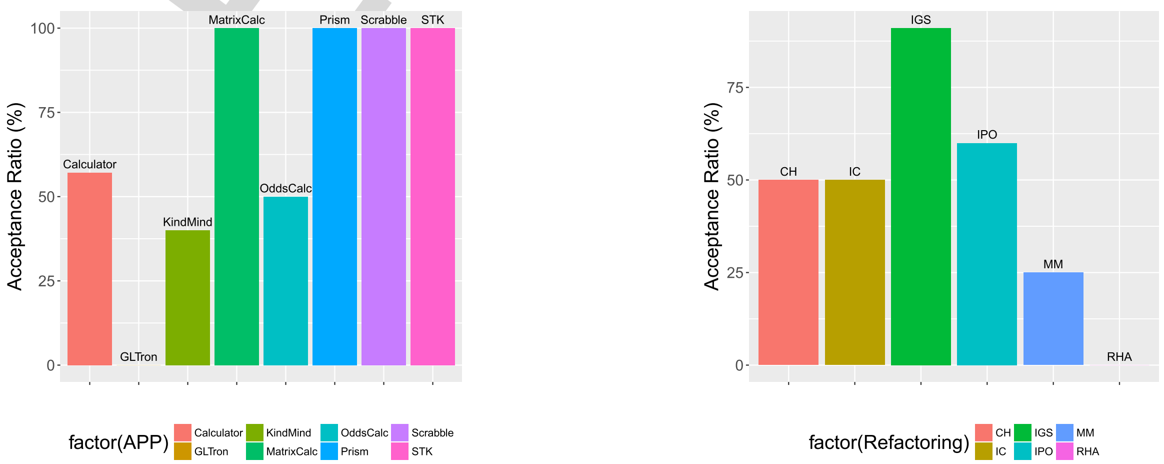

Fig. 14. Acceptance ratio of the refactorings proposed by EARMO.

the refactorings recommended for the app. However, some of the reasons behind her/his rejections are not convincing as we will discuss in the following paragraph. Overall, 68 percent 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 [48] 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 percent. 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 Blob 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 *GLTron* 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 latter 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 *Oddscalculator*. 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 did not 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 consumption of their apps.*

6 Threats to Validity

This section discusses the threats to validity of our study following common guidelines for empirical studies [70].

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], [32]. 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 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* [60]. 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. Because interpreted code runs slowly when profiling is enabled, it is probable that the energy consumption associated with each method call is higher. However, given that the profiling was enabled in all the experiments, we can assume that the instrumentation overhead introduced by the production of execution traces is constant between different runs of the same scenario.

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 times to ensure statistical validity. From the set of anti-patterns studied, we target one that is related to the use of device sensors, that is Binding Resources too early. Because our setup is measured inside a building, device location might be computed using Wi-Fi instead of GPS if the reception is not good enough. In that case, it is likely to be less than the cost of using GPS sensor outdoors. This also applies to network connections, where the cost incurred for connecting through Wi-Fi is likely to be less than the one incurred for using a cellular network. Additionally, in the evaluation of EARMO we use MonkeyRunner to communicate with apps though simulated signals rather than signals triggered through real sensors (for example, touchscreens or gravity sensors) on mobile devices, that could be regarded as not realistic. In case that a more realistic measurement is required, we can substitute intrusive methods, like using Monkeyrunner, with a robot arm that uses the same cyber-physical interface as the human user [71].

As explained in the *construct validity* our measurement apparatus is at least as precise as previous measurement setups.

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 δ, 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 apps 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. These results have to be interpreted carefully as they may depend on the specific device where we ran the experiments, the operating system and the virtual machine (VM) used by the operating system. For the former one, it is well known that in ART (Android Run Time used in this work) the apps are compiled to native code once, improving the memory and CPU performance, while previous VM for Android (Dalvik) runs along with the execution of an app, and may perform profile-directed optimizations on the fly. To validate this threat, we execute the energy consumption validation using Dalvik and ART VMs and found ± 1 percent of difference in the median of $\gamma(E', E_0)$ values for the apps used in the energy consumption validation. Hence, we suggest that our results area valid for both VMs, for the set of anti-patterns, apps, and scenarios used in this work.

Our study focuses on 20 android apps with different sizes and belonging to different domains from *F-Droid*, which is one of the largest repositories of open-source Android apps. Still, it is unclear if our findings would generalize to all Android applications. 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 [72], 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], [73].

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 [21] 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. [64] proposed a MO approach based on NSGA-II, with three conflicting objectives: removing anti-patterns, preserving semantic coherence, and history of changes. For the first objective, they generated a set of rules to characterize anti-patterns from a set of bad design examples. The second objective measure the semantic similarity among classes. Finally, history of changes refers to the similarity of the refactoring proposed with previous refactorings applied in the past. Mkaouer et al. [74] 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 consumption; 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 1,705 Mylyn interaction histories, Morales et al. [49] 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 [31], we proposed 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 percent. This previous work differs from EARMO in the targeted systems (desktop vs mobile), and the fact that energy consumption was not considered, but the testing effort of classes.

Recently, Banerjee et al. [68] proposed an approach to refactor mobile apps by relying on energy-consumption guidelines to control for energy-intensive device components. They report a reduction in energy consumption from 3 to 29 percent of in their testbed which was comprised of 10 *F-Droid* apps. While this work focuses only on improving the energy consumption, 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 consumption 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. [75] leveraged DECOR to detect 18 OO anti-patterns in mobile apps. Through a study of 1,343 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 [76] leveraged refactoring tools, such as PMD[9] or JDeodorant [77] 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. [78] proposed a catalog of 30 quality smells specific to the Android platform. These smells were reported to have a negative impact on quality attributes like consumption, user experience, and security. Reimann et al. also performed detections and corrections of certain code smells using the REFACTORY tool [79]. However, this tool has not been validated on Android apps [10].

Li et al. [46] investigate the impact of android developing practices and found that *accessing class fields*, *extracting array length into a local variable* in a for-loop and *inline getter and setters* can reduce the energy consumption of an app in test harness developed specifically for this purpose.

Hecht et al. [10] 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. Recently, they also evaluated the impact of removing three types of Android anti-patterns (two of them also studied in this work, e.g., *HashMap usage*, and *private getters and setters*) using a physical measurement setup [80].

Our proposed approach differs from these previous works in the sense that beside detecting anti-patterns in mobile apps, we propose a multiobjective approach to generate optimal sequences of refactorings that achieve a maximum removal of anti-patterns from the mobile apps, while controlling for energy consumption. In this way we avoid a direct aggregation of different, potentially conflicting objectives, allowing software maintainers to select among different trades or achieve a compromise between the two of them.

We validate our results by measuring the energy consumption of apps on a real mobile phone.

7.3 Energy Consumption

There are several works on the energy consumption of mobile apps [55], [81], [82], [83], [84], [85].

Some studies proposed software energy consumption frameworks [55] and tools [81] to analyze the impact of software evolution on energy consumption.

Green Miner [55] is a dedicated hardware mining software repositories testbed. The *Green Miner* physically measures the energy consumption of Android mobile devices and automates the reporting of measurements back to developers and researchers. A *Green Miner* web service[10] 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 4.1 V 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. [86] proposed *eprof*, a fine-grained energy profiler for Android apps, that can help developers understand and optimize their apps energy consumption. In [87], 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 percent 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. [88].

Other works aimed to understand software energy consumption [83], its usage [15], or the impact of users' choices on it [84], [89].

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.

9. https://pmd.github.io/

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

Sahin et al. [90] 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. In another work, Sahin et al. [15] 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 also highlighted the need for energy-aware refactoring approaches that can be integrated in IDEs.

Banerjee et al. [91] proposed a technique to identify energy hotspots in Android apps by the generation of test cases containing a sequence of user-interactions. They evaluate their technique using a testbed of 30 apps from *F-Droid*.

Pinto [92] suggested a refactoring approach to improve the energy consumption of parallel software systems. The approach was manually applied to 15 open source projects and reported an energy saving of 12 percent.

Li et al. [93] proposed an approach to transform web apps to improve energy consumption of mobile apps and achieved an improvement of 40 percent, with an acceptance rate of 60 percent among the users in a testbed of seven web apps. To address the same problem, but using multiobjective technique, Linares-Vásquez et al. [94] proposed an approach to generate *energy-friendly* color palettes that are consistent with respect to the original design in a testbed of 25 apps.

Wan et al. [67] propose a technique for detecting graphic user interfaces that consumes more energy than desirable. Their energy prediction estimation reached 12 percent compared to the real measurements on a testbed of 10 apps

Bruce et al. [95] leverage *Genetic Improvement* to improve the energy consumption of three *MiniSAT* downstream apps achieving 25 percent of improvement.

Manotas et al. [96] proposed a framework (*SEEDS*) to automatically select the most energy efficient Java's Collections API and achieve 17 percent of energy usage improvement in a testbed of seven Java apps.

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.

Our work differs from the ones presented in this category since we aim to improve the design quality of the apps, using anti-patterns as proxy for design quality, while maximizing energy efficiency. Therefore, our work contributes to fill a 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 consumption. 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 propose solutions to remove a median of 84 percent of anti-patterns, with a median execution time of less than a minute. We also quantify the battery energy gain of EARMO and found that in a multimedia app, when the proposed scenario is executed continuously until the battery drained out, it could extend battery life by up to 29 minutes.

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 multi-objective formulation.

Concerning the quality of the solutions proposed, we manually evaluated the precision of the sequences generated by EARMO and obtained of median 68 percent precision score. We study the cases where some of the refactorings in a sequence are not valid and provide guidelines for toolsmiths to improve the precision of automated refactoring approaches.

We also evaluated the overall design quality of the refactored apps in terms of five high-level quality attributes assessed by an external model, 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 percent 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.

Acknowledgments

This work has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Consejo Nacional de Ciencia y Tecnología, México (CONACyT).

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," *IEEE Softw.*, vol. 29, no. 4, pp. 25–27, Jul. 2012.

[3] D. L. Parnas, "Software aging," in *Proc. 16th Int. Conf. Softw. Eng.*, 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," *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 1–12, Jan. 2001.

[5] M. Gottschalk, J. Jelschen, and A. Winter, "Energy-efficient code by refactoring," *Softwaretechnik Trends*, vol. 33, no. 2, pp. 23–24, May 2013.

[6] Android performance tips. Jun. 2016. [Online]. Available: https://developer.android.com/training/articles/perf-tips.html

[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. Eng.*, 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. 29th Int. Conf. Softw. 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 *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas.*, 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 *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2015, Art. no. 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 *Proc. 7th Int. Conf. Quality Inf. Commun. Technol.*, 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," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan./Feb. 2010.

[13] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2004, pp. 350–359.

[14] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *Proc. 12th Eur. Conf. Softw. Maintenance Reengineering*, 2008, pp. 329–331.

[15] C. Sahin, L. L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 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 *Proc. 26th Int. Conf. Softw. Eng. Knowl. Eng.*, 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 *Proc. 1st Workshop de Sistemas Embarcados*, 2010, pp. 145–150.

[18] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proc. Int. Workshop Mobile Softw. Eng. Syst.*, 2016, pp. 59–69. [Online]. Available: http://doi.acm.org/10.1145/2897073.2897100

[19] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multi-objective evolutionary algorithm," in *Proc. 15th Annu. Conf. Genetic Evol. Comput.*, 2013, pp. 1461–1468.

[20] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, 2011, pp. 81–90.

[21] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proc. 9th Annu. Conf. Genetic Evol. Comput.*, 2007, pp. 1106–1113.

[22] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[23] P. Bourque and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2014.

[24] R. Morales, F. Chicano, F. Khomh, and G. Antoniol, "Exact search-space size for the refactoring scheduling problem," *Automated Softw. Eng. J.*, 2017. [Online]. Available: http://dx.doi.org/10.1007/s10515-017-0213-6

[25] M. O'Keeffe and M. O. Cinnéide, "Search-based software maintenance," in *Proc. 10th Eur. Conf. Softw. Maintenance Reengineering*, 2006, pp. 10 pp.–260.

[26] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proc. Annu. Conf. Genetic Evol. Comput.*, 2006, pp. 1909–1916.

[27] C. L. Simons, I. C. Parmee, and R. Gwynllyw, "Interactive, evolutionary search in upstream object-oriented class design," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 798–816, Nov./Dec. 2010.

[28] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," *Automated Softw. Eng.*, vol. 20, no. 1, pp. 47–79, 2013.

[29] R. Mahouachi, M. Kessentini, and M. O. Cinnéide, *Search-Based Refactoring Detection Using Software Metrics Variation*. Berlin, Germany: Springer, 2013, pp. 126–140.

[30] M. W. Mkaouer, M. Kessentini, S. Bechikh, and M. O. Cinnéide, *A Robust Multi-Objective Approach for Software Refactoring Under Uncertainty*. Berlin, Germany: Springer, 2014, pp. 168–183.

[31] 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 *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, 2016, pp. 24–35.

[32] M. Gottschalk, "Energy refactorings," Master's thesis, Softw. Eng. Group, Carl von Ossietzky Univ., Oldenburg, Germany, 2013.

[33] 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*, 1st ed. Hoboken, NJ, USA: Wiley, Mar. 1998.

[34] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1st ed. Reading, MA, USA: Addison-Wesley, Jun. 1999.

[35] D. Singh and W. J. Kaiser, "The atom LEAP platform for energy-efficient embedded computing," *Center for Embedded Netw. Sens. UCLA: Center for Embedded Netw. Sens.*, 2010, http://escholarship.org/uc/item/88b146bk

[36] D. Li, S. Hao, J. Gui, and W. G. J. Halfond, "An empirical study of the energy consumption of android applications," in *Proc. Int. Conf. Softw. Maintenance Evolution.*, Sep. 2014, pp. 121–130.

[37] R. Saborido, V. Arnaoudova, G. Beltrame, F. Khomh, and G. Antoniol, "On the impact of sampling frequency on software energy measurements," *PeerJ PrePrints*, vol. 3, 2015, Art. no. e1219. [Online]. Available: http://dx.doi.org/10.7287/peerj.preprints.1219v2

[38] D. F. Lochtefeld and F. W. Ciarallo, "Multi-objectivization via decomposition: An analysis of helper-objectives and complete decomposition," *Eur. J. Oper. Res.*, vol. 243, no. 2, pp. 395–404, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221714009916

[39] J. Knowles, L. Thiele, and E. Zitzler, "A tutorial on the performance assessment of stochastic multiobjective optimizers," Comput. Eng. Netw. Laboratory, ETH Zurich, Zurich, Switzerland, TIK Rep. 214, 2006.

[40] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. Hoboken, NJ, USA: Wiley, 2001.

[41] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[42] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization," in *Proc. Evol. Methods Design, Optimization Control Applicat. Ind. Problems (EUROGEN '01), Barcelona, Spain: Inte. Center Numerical Methods Eng. (CIMNE)*, 2002, pp. 95–100.

[43] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "MOCell: A cellular genetic algorithm for multiobjective optimization," *Int. J. Intell. Syst.*, vol. 24, no. 7, 2009, pp. 726–746, http://dx.doi.org/10.1002/int.20358

[44] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "Design issues in a multiobjective cellular genetic algorithm," in *Proc. Conf. Evol. Multi-Criterion Optimization*, 2007, pp. 126–140.

[45] C. Sahin, L. Pollock, and J. Clause, "From benchmarks to real apps: Exploring the energy impacts of performance-directed changes," *J. Syst. Softw.*, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216000893

[46] D. Li and W. G. J. Halfond, "An investigation into energy-saving programming practices for android smartphone app development," in *Proc. 3rd Int. Workshop Green Sustain. Softw.*, 2014, pp. 46–53. [Online]. Available: http://doi.acm.org/10.1145/2593743.2593750

[47] A. R. Tonini, L. M. Fischer, J. C. B. de Mattos, and L. B. de Brisolara, "Analysis and evaluation of the android best practices impact on the efficiency of mobile applications," in *Proc. 3rd Brazilian Symp. Comput. Syst. Eng.*, 2013, pp. 157–158.

[48] Android API: ArrayMap. [Online]. Available: https://developer.android.com/reference/android/support/v4/util/ArrayMap.html. Accessed on: May 18, 2017.

[49] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers context for automatic refactoring of software anti-patterns," *J. Syst. Softw.*, vol. 128, pp. 236–251, 2017.

[50] Monkey runner concepts. [Online]. Available: https://developer.android.com/studio/test/monkeyrunner/index.html. Accessed on: May 18, 2017.

[51] Debugging Android apps. [Online]. Available: https://developer.android.com/reference/android/os/Debug.html. Accessed on: May 18, 2017.

[52] N. Cliff, *Ordinal Methods for Behavioral Data Analysis*. Hove, U.K.: Psychology Press, 2014.

[53] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices," in *Proc. Annu. Meeting Southern Assoc. Institutional Res.*, 2006.

[54] Android API guides: Location strategies. [Online]. Available: https://developer.android.com/guide/topics/location/strategies.html. Accessed on: May 18, 2017.

[55] 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 *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 12–21.

[56] Y.-G. Gueheneuc and H. Albin-Amiot, "Recovering binary class relationships: Putting icing on the UML cake," *ACM SIGPLAN Notices*, vol. 39, no. 10, pp. 301–314, 2004.

[57] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multi-layered framework for design pattern identification," *IEEE Trans. Softw. Eng.*, vol. 34, no. 35, pp. 667–684, Sep. 2008.

[58] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Dept. Comput. Sci., Univ. Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[59] B. L. Miller and D. E. Goldberg, "Genetic algorithms, tournament selection, and the effects of noise," *Complex Syst.*, vol. 9, no. 3, pp. 193–212, 1995.

[60] Y.-G. Guéhéneuc, "PTIDEJ: Promoting patterns with patterns," in *Proc. 1st ECOOP Workshop Building Syst. Using Patterns*, 2005, pp. 1–9.

[61] 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

[62] J. J. Durillo and A. J. Nebro, "jMetal: A java framework for multi-objective optimization," *Advances Eng. Softw.*, vol. 42, pp. 760–771, 2011.

[63] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach," *IEEE Trans. Evol. Comput.*, vol. 3, no. 4, pp. 257–271, Nov. 1999.

[64] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *J. Syst. Softw.*, vol. 105, pp. 18–39, 2015.

[65] K. Miettinen, *Nonlinear Multiobjective Optimization*. New York, NY, USA: Springer, 2012. [Online]. Available: https://books.google.ca/books?id=bnzjBwAAQBAJ

[66] C. Sahin, et al., "How does code obfuscation impact energy usage?" *J. Softw.: Evol. Process*, vol. 28, pp. 565–588, 2016.

[67] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond, "Detecting display energy hotspots in Android apps," in *Proc. 8th IEEE Int. Conf. Softw. Testing Verification Validation*, Apr. 2015, pp. 1–10.

[68] A. Banerjee and A. Roychoudhury, "Automated re-factoring of android apps to enhance energy-efficiency," in *Proc. Int. Workshop Mobile Softw. Eng. Syst.*, 2016, pp. 139–150. [Online]. Available: http://doi.acm.org/10.1145/2897073.2897086

[69] Android API guides: Broadcasts. [Online]. Available: https://developer.android.com/guide/components/broadcasts.html. Accessed on: May 18th, 2017.

[70] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. Thousand Oaks, CA, USA: SAGE Publications, 2002.

[71] K. Mao, M. Harman, and Y. Jia, "Robotic testing of mobile apps for truly black-box automation," *IEEE Softw.*, vol. 34, no. 2, pp. 11–16, Mar./Apr. 2017.

[72] W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang, "The app sampling problem for app store mining," in *Proc. 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 123–133. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820535

[73] 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 *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 101–110.

[74] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. ÓCinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 331–336.

[75] M. Linares-V ásquez, 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 *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 232–243.

[76] D. Verloop, *Code Smells in the Mobile Applications Domain*. Delft, The Netherlands: Delft Univ. Technol., 2013.

[77] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of feature envy bad smells," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2007, pp. 519–520.

[78] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," *Softwaretechnik-Trends*, vol. 34, no. 2, 2014, http://pi.informatik.uni-siegen.de/stt/34_2/01_Fachgruppenberichte/MMSM2014/MMSM2014_Paper6.pdf

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

[80] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *Proc. IEEE 24th Int. Conf. Softw. Anal. Evol. Reengineering*, Feb. 2017, pp. 115–126.

[81] K. Aggarwal, A. Hindle, and E. Stroulia, "GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 311–320.

[82] I. Polato, D. Barbosa, A. Hindle, and F. Kon, "Hybrid HDFS: Decreasing energy consumption and speeding up hadoop using SSDs," *PeerJ PrePrints*, vol. 3, 2015, Art. no. e1320. [Online]. Available: http://dx.doi.org/10.7287/peerj.preprints.1320v1

[83] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about the energy consumption of software?" *PeerJ PrePrints*, vol. 3, 2015, Art. no. e886.

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

[85] K. Rasmussen, A. Wilson, and A. Hindle, "Green mining: Energy consumption of advertisement blocking methods," in *Proc. 3rd Int. Workshop Green Sustain. Softw.*, 2014, pp. 38–45.

[86] 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 *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 29–42.

[87] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 92–101.

[88] M. Linares-V ásquez, 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 *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597085

[89] R. Saborido, G. Beltrame, F. Khomh, E. Alba, and G. Antoniol, "Optimizing user experience in choosing android applications," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, Mar. 2016, pp. 438–448.

[90] C. Sahin, et al., "Initial explorations on design pattern energy usage," in *Proc. 1st Int. Workshop Green Sustain. Softw.*, 2012, pp. 55–61.

[91] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 588–598.

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

[93] D. Li, A. H. Tran, and W. G. J. Halfond, "Making web applications more energy efficient for OLED smartphones," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 527–538. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568321

[94] M. Linares-V ásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of GUIs in android apps: A multi-objective approach," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 143–154.

[95] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proc. Annu. Conf. Genetic Evol. Comput.*, 2015, pp. 1327–1334.

[96] I. Manotas, L. Pollock, and J. Clause, "SEEDS: A software engineer's energy-optimization decision support framework," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 503–514. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568297

Rodrigo Morales received the BS degree in computer science from Polytechnic of Mexico, in 2005 and the MS degree in computer technology from the Polytechnic of Mexico, in 2008, where he also worked as a professor in the Computer Science Department for five years. He is working toward the PhD degree at Polytechnique Montreal. He has also worked in the bank industry as a software developer for more than three years. He is currently supervised by Foutse Khomh, Giuliano Antoniol (Poly Montreal), and Francisco Chicano (University of Malaga, Spain). His research interests include software design quality, anti-patterns, and automated-refactoring. He is a member of the IEEE.

Rubén Saborido received the MS degree in software engineering and artificial intelligence from the University of Malaga (Spain), where he also worked for three years as a researcher. He is working toward the PhD degree in software engineering at Polytechnique Montreal. His research focuses on search based software engineering applied to performance and energy optimization of mobile devices. He is also interested in the use of metaheuristics to solve complex multiobjective optimization problems and in the design of algorithms to approximate a part of the whole Pareto optimal front taking into account user preferences. He has published six papers in ISI indexed journals, and conference papers in MCDM, SANER, and ICPC. He co-organized the International Conference on Multiple Criteria Decision Making, in 2013. He is a member of the IEEE.

Foutse Khomh received the PhD degree in software engineering from the University of Montréal, in 2010. He is an associate professor with Polytechnique Montréal, where he heads the SWAT Lab on software analytics and cloud engineering research (http://swat.polymtl.ca/). His research interests include software maintenance and evolution, cloud engineering, service-centric software engineering, empirical software engineering, and software analytic. He has published several papers in international conferences and journals, including ICSM(E), MSR, SANER, ICWS, HPCC, IPCCC, the *Journal of Systems and Software*, the *Journal of Software: Evolution and Process*, and EMSE. His work has received three Best Paper Awards and fourteen nominations for Best paper Awards. He has served on the program committees of several international conferences including ICSM(E), SANER, MSR, ICPC, SCAM, ESEM and has reviewed for top international journals such as SQJ, EMSE, TSE and TOSEM. He is program chair for Satellite Events at SANER 2015, program co-chair of SCAM 2015 and ICSME 2018, and general chair of ICPC 2018. He is one of the organizers of the RELENG workshop series (http://releng.polymtl.ca) and has been guest editor for special issues in the *IEEE Software Magazine* and the *Journal of Software: Evolution and Process*. He is a member of the IEEE.

Francisco Chicano received the degree in physics from the National Distance Education University and the PhD degree in computer science from the University of Malaga. Since 2008 he is in the Department of Languages and Computing Sciences of the University of Malaga. His research interests include the application of search techniques to Software Engineering problems. In particular, he contributed to the domains of software testing, model checking, software project scheduling, and requirements engineering. He is also interested in the application of theoretical results to efficiently solve combinatorial optimization problems. He is in the editorial board of several international journals and has been program chair in international events.

Giuliano Antoniol received the Laurea degree in electronic engineering from the Universita di Padova, Italy, in 1982. In 2004 he received the PhD degree in electrical engineering from Polytechnique Montréal. He worked in companies, research institutions and universities. In 2005 he was awarded the Canada Research chair Tier I in software change and evolution. He has participated in the program and organization committees of numerous IEEE-sponsored international conferences. He served as program chair, industrial chair, tutorial, and general chair of international conferences and workshops. He is a member of the editorial boards of four journals: The *Journal of Software Testing Verification & Reliability*, the *Journal of Empirical Software Engineering* and the *Software Quality Journal* and the *Journal of Software Maintenance and Evolution: Research and Practice*. He served as deputy chair of the Steering Committee for the IEEE International Conference on Software Maintenance. He contributed to the program committees of more than 30 IEEE and ACM conferences and workshops, and he acts as referee for all major software engineering journals. He is currently full professor with Polytechnique Montréal, where he works in the area of software evolution, software traceability, search based software engineering, software testing and software maintenance. He is a senior member of the IEEE.