

# EcoAndroid: An Android Studio Plugin for Developing Energy-Efficient Java Mobile Applications (Extended Abstract)\*

Ana Sofia Gonçalves Ribeiro  
Instituto Superior Técnico, Universidade de Lisboa  
anasofiaribeiro@tecnico.ulisboa.pt

**Abstract**—Mobile devices have become indispensable in our daily life and reducing the energy consumed by them has become essential over recent years. For economical and environmental reasons, as well as enhancing the user experience, extending battery duration has become a non-functional requirement developers should be concern with. However, developing energy-efficient mobile applications is not a trivial task. To address this problem, we present EcoAndroid, a publicly-available Android Studio plugin that automatically applies a set of energy patterns to Java source code. It currently supports ten different cases of energy-related refactorings, over five energy patterns taken from the literature. We used EcoAndroid to analyze 100 Java mobile applications from F-Droid and we found that 35 of the projects had a total of 95 energy code smells detected by the plugin. We used EcoAndroid to automatically refactor all the code smells identified. We submitted the 42 refactorings that introduced code fixes (and not just informational warnings) as pull requests to the maintainers of the respective projects. Of a total of 42 pull requests, we received replies to 25 of them (59.5%); of those, 20 (80%) were accepted and merged into the original projects. In total, we contributed to improve the energy efficiency of 12 different Android mobile applications. These results, together with the results obtained in a user study with 12 participants, show that EcoAndroid is useful, usable, and the alterations proposed by the tool are easily accepted by developers.

**Index Terms**—Sustainable Software, Green Software, Energy Consumption, Energy Patterns, Code Smells, Refactoring.

## I. INTRODUCTION

Mobile devices have become a fundamental accessory in a person's current day-to-day life. They are used as credit cards, work tools, educational helpers, among various useful purposes. Unfortunately, the battery power on them is finite and, despite the advances in hardware and battery technology, the needs of most users are not yet met. As a result, the reduction of the energy consumed by mobile devices has become an important non-functional requirement.

Regarding user practice, decreasing the energy consumption directly reduces the amount of times a mobile device needs to be charged, creating a more convenient experience of the device for the user. A study [2] in 2013, analysing comments left in the Google Play market place for Android applications, concluded that 18% of the complaints were related to energy problems.

\* This extended abstract is an earlier draft of the paper [1] and is required as part of the MSc submission at Instituto Superior Técnico, University of Lisbon.

Environmental concerns are also an incentive to reduce the energy consumed by our devices. The production of energy from fossil fuels produces greenhouse gases which contributes directly to the air pollution. Even if nowadays there are renewable sources to produce energy, such as solar and wind, electricity generated from fossil fuels still accounts for significant percentage of the energy produced. For example, in 2018, 70% of the world's energy was produced from fossil fuels [3].

One way of decreasing the energy consumed by a mobile device is to ensure that the mobile applications that the device runs are energy-efficient. However, improving the energy efficiency of a mobile application is a complex task since a lot of factors can influence energy consumption. Factors include: the mobile networking technology used (3G, GSM or WiFi) [4]; heavy graphic processing; and screen usage while on an application. Taking these factors into consideration is not always trivial, meaning that they can be easily overlooked by developers when coding.

An approach that makes the development of energy-efficient mobile applications easier is following so-called *energy patterns*, which are code patterns known to use energy prudently. Work documenting these patterns has been growing in recent years [5]–[8]. In 2019, Cruz et al. [5] presented a catalog of 22 energy-related patterns. The catalog can be of great assistance to mobile application developers, as it describes each pattern and its context, also providing a series of examples and references. However, the manual application of these patterns is not trivial and can be time-consuming.

### A. Objectives and Contributions

The main goal of this project is to create a developer tool that can assist in the development of energy-efficient mobile applications, by automatically refactoring their source code so that they follow well-known energy patterns. We also aim at: a) giving an overview of previous work done in this field, listing energy patterns already documented (in the search of a subset to support); b) discussing which mobile platforms (Android or iOS) and which languages are the most used in each platform (meaning a bigger impact if supported); c) listing a set of source code refactoring tools that can be used to implement the main goal.

Our main contribution is a publicly-available tool named EcoAndroid<sup>1</sup> that focuses on Android applications and is in the form of an Android Studio plugin that automatically applies a set of energy patterns to Java source code. At the time of writing, the plugin supports ten different cases of energy-related refactorings, over five energy patterns taken from the literature [5]. Out of the ten cases, two are informational warnings, as they only insert `//TODO's` to the source code, due to the complexity applying the refactoring ourselves — for example when it is needed to register the mobile application to set up push notifications on the app.

We used EcoAndroid to analyze 100 projects from F-Droid and we found that 35 of the projects had a total of 95 energy code smells detected by the plugin. We used EcoAndroid to automatically refactor all the code smells identified. We submitted the 42 refactorings that introduced code fixes (and not just informational warnings) as pull requests to the maintainers of the respective projects. Of a total of 42 pull requests, we received replies to 25 of them (59.5%); of those, 20 (80%) were accepted and merged into the original projects. In total, we contributed to improve the energy efficiency of 12 different Android mobile applications. Through this work, we propose to answer the following research questions:

- RQ1.** What energy patterns are already known by the software engineering community?
- RQ2.** What are the most relevant energy patterns to support?
- RQ3.** Are there existing tools that automatically apply energy patterns to the source code of mobile applications?
- RQ4.** What are the challenges in automatically applying energy patterns?

The main contributions can be summarized as follows:

- EcoAndroid, an extendable Android Studio plugin, created to assist developers in creating energy-efficient mobile applications;
- Refactoring of real-world code: we improved the energy efficiency of 12 real-world Android mobile applications by using EcoAndroid to automatically fix 25 code smells;
- A study of the most common code smells in 100 Android mobile applications.

## II. BACKGROUND AND RELATED WORK

### A. Energy Consumption and Energy Profiling

*Energy Profiling* is the process of measuring the energy consumed by a device, and in our specific case, a mobile device. This process has been addressed by the scientific community before. Ahmad et al. [9] present a paper which reviews mobile applications energy profiling. They divided energy profiling schemes into two categories: software-based and hardware-based. Software-based exploits a software module to collect mobile component's power usage statistics to construct power models to estimate application's energy consumption. Hardware-based utilizes external hardware equipment, which are expensive, labor-intensive, and non-scalable compared

with software-based solutions. There are tools [10]–[15], that have tackled this problem, in both categories.

### B. Energy Patterns

An *Energy Pattern* describes, formally, the alterations needed to reduce the energy consumed by a device. Studies [5]–[8], [16] with the goal of documenting energy patterns have emerged over recent years. Table I summarizes every energy pattern documented and the paper it belongs to.

TABLE I  
ENERGY PATTERNS SUMMARY.

Energy Pattern	Paper(s)
Dark UI Colors	[5]–[7]
CPU Offloading	[6]
HTTP Requests	[6], [16]
Software Piracy	[6]
I/O Operations	[6]
Continuously Running App	[6]
Third-Party Advertising	[7]
Binding Resources Too Early	[5], [7]
Statement Change	[7]
Data Transfer	[7]
Use of Memory	[16]
Performance Tips	[16]
View Holder	[8]
DrawAllocation	[8]
WakeLock	[8]
Recycle	[8]
ObsoleteLayoutParam	[8]
Power Save Mode	[5]
Power Awareness	[5]
Enough Resolution	[5]
No Screen Interaction	[5]
Avoid Extraneous Graphics and Animations	[5]
Dynamic Retry Delay	[5]
Race-to-idle	[5]
Reduce Size	[5]
Batch Operations	[5]
Cache	[5]
Decrease Rate	[5]
User Knows Best	[5]
Inform Users	[5]
Manual Sync, On Demand	[5]
Push Over Poll	[5]
WiFi Over Cellular	[5]
Suppress Logs	[5]
Sensor Fusion	[5]
Kill Abnormal Tasks	[5]
Avoid Extraneous Work	[5]

### C. Mobile Applications Environments and Languages

In 2017, a study done by Habchi et al. [17] compared the ratio of energy code smells in iOS and Android mobile applications, concluding that the latter had a higher number of energy code smells in the source code. The study also states that these differences are related to the platform and not to the differentiation in programming language. The main languages for programming iOS mobile applications are Swift and Objective-C while for Android mobile applications are Java and Kotlin. Since we are interested in maximizing the

<sup>1</sup>Available in the JetBrains store: <https://plugins.jetbrains.com/plugin/15637-ecoandroid>.

impact of this project, we focus on Android mobile applications, targeting Java applications.

The most used IDEs for Java development are IntelliJ, Eclipse and NetBeans. In terms of Android application development, the IDEs Android Studio (built on IntelliJ), IntelliJ, and Eclipse are the best choices, being that the first one is the official one for Android development and the one chosen for this project. Note that, even though we focus on Android mobile applications, Android Studio can also be used to develop iOS mobile applications. As long as these applications are written in the Java programming language, the tool that we propose can be used.

#### D. Refactoring for Java Source Code

*Refactoring* is the process of changing the internal structure of a program without changing its external behaviour. It is mainly used to improve code quality and reliability. It has both benefits and risks and it might be difficult to discover when to apply refactoring [18]. As presented in the study by Kim et al. [19], while refactoring can be known to reduce the number of bugs in a program and improve maintainability and reliability, it also has some risks associated. Such risks are, for example, the introduction of regression bugs and the increase of the testing cost.

**Leafactor** [20], [21] is a tool that automatically refactors Android mobile applications source code to reduce energy consumption. Leafactor is an Eclipse plugin while our tool is compatible with both Android Studio and IntelliJ. The 5 refactorings supported by Leafactor are acquired from a previous study [8], by the same authors, about the effect of performance-based practices on mobile application' energy consumption. These patterns listed in table I.

**Chimera** [22] covers 11 energy-greedy code patterns. This paper [22] also compares the energy savings of combinations of refactorings. It uses the Lint<sup>2</sup> for the inspection phase and Autorefactor [23] for the refactoring phase. A new aspect about this project is how broad the evaluation is, inspecting more than 600 mobile applications. It covers the same code smells as Leafactor and the four extra ones.

**AEON** (Automated Android Energy-Efficiency Inspection) [24] is a support framework, compatible with IntelliJ and Android Studio. It automatically detects energy inefficiencies in Android mobile applications and helps developers fixing those inefficiencies. It also supports developers in verifying, refactoring and profiling such inefficiencies.

**EARMO** [25] is an approach that detects and corrects energy-related anti-pattern in mobile applications, while accounting for energy consumption when performing the refactorings. It supports 8 anti-patterns within two categories: Object-oriented specific and Android-specific. The refactoring is achieved by support of refactoring-tool-support of Android Studio and Eclipse. When that was not possible, the changes were applied manually.

**aDoctor** [26], a tool proposed by Palomba et al., is able to identify 15 Android-specific code smells from a catalog

by Reimann et al. [27]. It is built on top of Eclipse Java Development Toolkit (JDK). Later on, *aDoctor* was extended as an Android Studio plugin supporting 5 energy-related refactorings.

A paper [28] by Le Goaër presents a new category in Android lint entitled **Greenness**. This category has 11 checks, which can be viewed as an inspection in Android Studio.

**HOT-PEPPER** [29] is able to detect and correct 3 types of Android-specific code smells. It uses PAPRIKA [30], a static tool analysis for Android apps for the detection and correction of code smells. As a final step, *HOT-PEPPER* uses a tool called NAGA VIPER, to compute energy metrics and evaluate the impact of corrected APKs, being able to inform the developer which APK is the best energy-efficient version, for a given scenario.

TABLE II  
ENERGY-SPECIFIC REFACTORING TOOLS/APPROACHES.

Tool/Approach Name	Refactoring Environment	Paper(s)
Leafactor	Eclipse	[20], [21]
Chimera	-	[22]
AEON	IntelliJ, Android Studio	[24]
EARMO	IntelliJ, Android Studio ,Manually	[25]
aDoctor	Eclipse, Android Studio	[26], [31]
Greenness category	Android Studio (Android lint)	[28]
HOT-PEPPER	-	[29]

### III. ECOANDROID: AN ANDROID STUDIO PLUGIN

#### A. Architecture

EcoAndroid is a publicly-available Android Studio plugin that suggests automated refactorings with the aim of reducing energy consumption of Java android applications. Android Studio is an integrated development environment for Google's Android operating system, built on JetBrains' IntelliJ IDEA. Due to this fact, EcoAndroid plugin is also compatible with IntelliJ. Android Studio is the official IDE for Android app development, making it the best choice for a maximizing the impact of our project. To the best of our knowledge, there are no general-purpose refactoring plugins for Android Studio that can serve as the basis for this project. Thus, to aid in the refactoring of the source code, the *Program Structure Interface* (PSI) [32] of IntelliJ will be used. PSI is a layer of the IntelliJ Platform responsible for parsing files and creating the syntactic and semantic code model. It creates PSI files, that are the root of a structure representing the contents of a file as a hierarchy of elements in a particular programming language. PSI is a read-write representation of the source code as a tree of elements corresponding to the structure of a source file. The PSI can be modified by adding, replacing and deleting PSI elements. These features are what allows the detection of possible energy improvements and the refactoring itself.

EcoAndroid is implemented as extending IntelliJ's functionality. As it is, the functionality is added as an

<sup>2</sup>Lint is a code analysis tool developer.android.com/studio/write/lint.

<extension/> element in the `plugin.xml`.<sup>3</sup> In this extension, there are inspections (where each represent a case the plugin supports). An IntelliJ's plugin can have two type of inspections: a local inspection or a global inspection. As the names suggest, a local inspection looks at only one file while a global inspection looks at a group of files. Due to this, a global inspection does not appear as warning along the source code but needs to be run manually by the user. Since we do not wish to alter a big portion of the source code, every energy pattern is implemented as a local inspection. The results from the inspection can be viewed in two ways: a warning of the source code currently being viewed; or from the results of the IDE's inspection task as a list. In the latter way, the developer can ask to inspect a file, a folder or even the whole project. They can also choose what inspections to ran. The plugin supports a total of five energy patterns, with ten cases among those. Each case is implemented as a local inspection in the plugin. Figure 1 illustrates the process flow of the user interaction between the developer and EcoAndroid. The plugin starts by performing a static analysis, aided by the PSI API. The source code is represented as Abstract Syntax Trees (AST) (actions ① and ②). If a code smell is found, a warning is shown to the developer (action ③) and, if the they wish to do so (action ④), the refactoring, which is also aided by the PSI API, is executed (action ⑤).

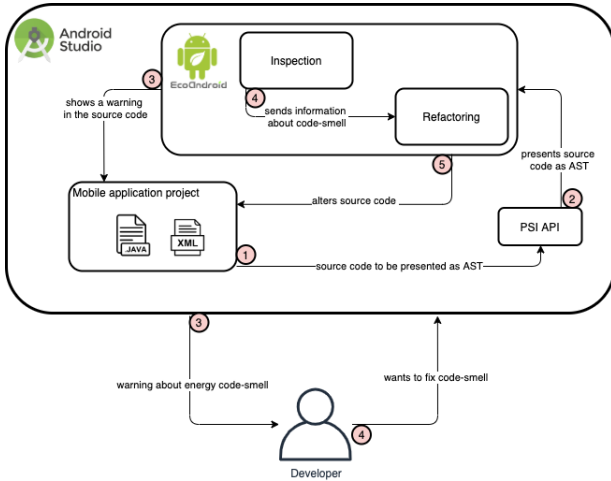


Fig. 1. EcoAndroid detection and refactoring process.

## B. Implementation

EcoAndroid supports a total of 5 energy patterns, divided in 10 separate cases. The energy patterns supported are a subset of the ones presented in the catalog by Cruz et al. [5]. The energy patterns are: *Dynamic Retry Delay*, *Push Over Poll*, *Reduce Size*, *Cache*, and *Avoid Extraneous Graphics and Animations*. However, in some of these patterns more than one case was implemented. EcoAndroid distinguishes two types of warnings: informational ones and non-informational ones.

<sup>3</sup>Plugin.xml is the plugin configuration file which has information about the actions and inspections done by the plugin.

The first type exist due to either the inability to implement the change or because the refactoring implied too many changes to the source code. The catalog [5] presents a list of GitHub commits in which alterations correspond to the application of the energy patterns. During the development of EcoAndroid, the approach taken when supporting an energy pattern was to support the alteration made by the commits shown in Cruz et al.'s catalog. Where possible, every Java source case was covered. Every example shown in the subsections below is available in the GitHub project for the EcoAndroid plugin.

1) *Dynamic Retry Delay*: The objective of the *Dynamic Retry Delay* pattern is to increase the interval between attempts to access a resource, avoiding trying to constantly access a resource that most likely went down. If an attempt to access a resource fails, the time between attempts should be increased, until a certain value, in order to space the access to the resource. If the access is successful, the interval should not be changed. The energy pattern has two cases: *Dynamic Retry Delay* and *Check Network*. The first one is detailed in the next section.

a) *Dynamic Wait Time*: The first case of the Dynamic Retry Delay energy pattern is the entitled, by the plugin, *Dynamic Wait Time*. The interval between threads sleep should grow exponentially and not stay constant, decreasing the change of trying to access a resource that most likely went down.

```

private void startLongPoll(String polledFile,
    int backOffSeconds) {
    pollingTask = new Thread() {
        public void run() {
            long start_time = System.currentTimeMillis();
            long longpoll_timeout = 480;
            int newBackoffSeconds = 0;
            if(backOffSeconds != 0) {
                log.info("Backing off for " +
                    backOffSeconds + " seconds");
                try {
                    Thread.sleep((long)
                        (backOffSeconds * 1000));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            if(System.currentTimeMillis() - start_time <
                longpoll_timeout * 1000) {
                log.info("Longpoll timed out to quick,
                    backing off for 60 seconds");
                newBackoffSeconds = 60;
            }
            else {
                log.info("Longpoll IO exception,
                    restarting backing off {} seconds"
                        + 30);
                newBackoffSeconds = 30;
            }
            startLongPoll(polledFile, newBackoffSeconds);
        }
    };
}

```

Listing 1: Dynamic Wait Time - energy opportunity detected.

In the example shown in listing 1, there is a sleep invocation, using the `backOffSeconds` variable. This variable comes from the parameter of the method `startLongPoll`. As it is a parameter, the inspection looks for method calls of the method `startLongPoll` in the current Java file. As we can observe from the listing, there is a method call which uses

the `newBackOffSeconds` variable to invoke the method. The variables are assigned with constant values, either 30 or 60. With this scenario, the plugin flags this as a problem, showing up as a warning on the variable `backOffSeconds`. Since the code does not already have an informational warning about this pattern added by the plugin, EcoAndroid presents the user with two warnings.

① "EcoAndroid: Dynamic Retry Delay Energy Pattern - information about a new approach to implement it"

```
pollingTask = new Thread () {
    /*
     * TODO EcoAndroid
     * DYNAMIC RETRY DELAY ENERGY PATTERN INFO WARNING
     * Another way to implement a mechanism that manages
     * the execution of tasks and
     * their retrying, if said task fails
     * This approach uses the android.work package
     * If you wish to know more about this topic,
     * read the following information:
     * https://developer.android.com/topic/libraries/
     * architecture/workmanager/how-to/define-work
     */
    public void run() { ... }
```

Listing 2: Dynamic Wait Time - energy pattern applied (information about a new approach to implement it).

The informational warning given to the developer does not alter any source code, only adding a comment with a link to further explain how to use the `WorkRequest` class instead of using `Thread` class.

② "EcoAndroid: Dynamic Retry Delay Energy Pattern - switching to a dynamic wait time between resource attempts case"

```
private void startLongPoll(String polledFile,
    int backOffSeconds) {
    pollingTask = new Thread () {
        int accessAttempts = 0;
        public void run() {
            if(System.currentTimeMillis() - start_time
                < longpoll_timeout * 1000) {
                log.info("Longpoll timed out to quick,
                    backing off for 60 seconds");
                accessAttempts++;
            }
            else {
                log.info("Longpoll IO exception,
                    restarting backing off {} seconds"
                        + 30);
                accessAttempts++;
            }
            newBackoffSeconds = (int) (60.0 *
                (Math.pow(2.0, (double) accessAttempts)
                    - 1.0));
            startLongPoll(polledFile, newBackoffSeconds);
        }
    };
}
```

Listing 3: Dynamic Wait Time - energy pattern applied (switching to a dynamic wait time between resource attempts case).

The second option presented to the developer alters the source code, changing every static variable assignment to a dynamic one. It starts by creating a variable entitled `accessAttempts`, initialized at 0. As the name suggests, the variable holds the number of access attempts to a resource. Then every static assignment done to the variable that puts the

thread to sleep, in this case it is `newBackoffSeconds`, is altered to an incremental assignment of `accessAttempts` variable. At last, the number of access attempts is altered to a value of time with an upper bound. Listing 3 represents the application of the *Dynamic Wait Time* pattern, which applies the alterations described.

2) *Push Over Poll*: A *push notification* establishes and maintains a connection with the server over the Internet and that allows the server to send data to the application when something has actually changed on the server. On the other hand, *Polling* is the continuous checking of other programs or devices by one program or device to see what state they are in, usually to see whether they are still connected or want to communicate. The goal of this energy pattern is to use push notifications instead of actively querying resources, such as polling. This transformation is specifically beneficial when there is not a significant number of notifications, as shown by Dinh and Boonkrong [33], who compare battery usage between these two techniques. If there is not a significant number of notifications coming in, pushing notifications will be a better choice since it's not always actively querying resources. If there is always notifications coming in, the difference between these two mechanisms is not as significant. This energy pattern has one case: *Informational Warning FCM*.

3) *Reduce Size*: The goal of the pattern *Reduce Size* is to reduce the size of the data being transferred as much as possible, therefore reducing the energy being used in the transfer. The change to be made consists in transforming/compressing the data being transmitted, whenever a data transfer occurs. This energy pattern has one case: *GZIP Compression*.

4) *Cache*: The goal of the *Cache* pattern is to store data that is being used frequently, which means a lower energy consumption since it reduces the amount of code executed. This energy pattern has five cases: *Check Metadata*, *Check Layout Size*, *SSL Session Caching*, *Passive Provider Location* and *URL Caching*.

5) *Avoid Extraneous Graphics and Animations*: Displaying graphics and animations are resources with an high energy consumption. The intent of the *Avoid Extraneous Graphics and Animations*'s pattern is to reduce the usage of this resources as much as possible. For example, on the usage of any resource with an high energy consumption that doesn't have a direct impact on the user experience. However, knowing when to apply this pattern is a challenge since it is difficult to know exactly when a resource is strictly needed or when the resource does not have a direct impact on the user experience. The energy pattern has one case: *Dirty Rendering*.

#### IV. EVALUATION

A possible way of evaluating the effectiveness of EcoAndroid is to measure the consumption of energy before and after a proposed refactoring. However, due to the complexity of directly measuring or estimating the energy a mobile application consumes [9], we follow a different approach. Since the energy patterns applied are retrieved from research papers who verified their reliability, we argue that measuring

the energy consumed after refactoring is not strictly necessary, since a decrease of energy consumption is almost guaranteed. The evaluation is thus divided into three phases:

**First Phase.** We measure how many refactorings EcoAndroid suggests for a realistic set of mobile Java applications and how many of those are false positives. We call this phase the *Objective Evaluation*.

**Second Phase.** Based on the results obtained in the first phase of the evaluation, we send the proposed changes to the maintainers of each mobile application (as pull requests). The goal is to obtain feedback from the application developers, but also, to measure the number of proposals accepted. We call this phase the *Subjective Evaluation*, since results depend on the appreciation of the mobile applications' maintainers.

**Third Phase.** The final phase of the evaluation consists in a user study, which aims to assess the usability of EcoAndroid. In the user study, we focus on the most relevant energy patterns (e.g. we only consider patterns that effectively change the code, rather than just adding annotations).

#### A. Mobile Applications Analyzed

For the evaluation of EcoAndroid, it is required to identify a set of mobile applications on which EcoAndroid is used to detect possible improvements in terms of energy consumption. We used Android mobile applications retrieved from *F-droid* [34], an alternative app store that catalogs over 2000 mobile applications that are Free and Open Source Software (FOSS). We retrieved meta-information about all the F-Droid applications<sup>4</sup> and we filtered and ordered them before being used for the evaluation.

- The source code of the application is available in GitHub;
- The GitHub project is not archived;
- The GitHub project has had a commit since 2018;
- The source code of the application is written in Java.

The mobile applications were then sorted by the following order:

- 1<sup>st</sup> Percentage of Pull Requests accepted;
- 2<sup>nd</sup> Date of Last Commit;
- 3<sup>rd</sup> Total merged Pull Requests;
- 4<sup>th</sup> Number of GitHub Stars;
- 5<sup>th</sup> Number of GitHub. Watchers

The first three criteria were chosen to increase our chances of having feedback from developers. Our intuition is that maintainers of projects that accept more pull requests might be more open to discuss our proposals. The last two criteria were chosen to maximize impact by selecting popular projects. After filtering and ordering the mobile applications, the top 100 applications were used in the evaluation process. The first phase of the evaluation consisted in determining how many refactorings are suggested by EcoAndroid for the top 100 mobile applications retrieved from the filtered and ordered

dataset. For this, we executed EcoAndroid in batch mode, since doing it manually for 100 applications would be too time-consuming. Table III presents the results. The lines with a gray background refer to the refactorings which introduce `//TODO's` into the source code. This first phase happened in three stages: we first processed the top twenty mobile applications, then the following twenty, and then the remaining sixty applications. Between each stage, we incorporated any relevant feedback received from developers. For example, errors resulting in false positives were fixed and no longer a problem in the following stages.

Two feedback from developers were labeled as errors of the plugin and fixed. On the pull requests in stage 1, a pull request to the Second Screen app with the *Check Network* energy pattern, the developer said that the project did not declare permission to use the internet so the refactoring did not make sense. This was fixed and no longer was a problem in the following stages. Another pull requests from stage 1, to the Hacs mobile application with the *Check Metadata* energy pattern, the developer answered that the refactoring could break some notification. This bug was fixed and is no longer a problem in the following stages.

TABLE III  
NUMBER OF ENERGY OPPORTUNITIES DETECTED BY ECOANDROID.

Energy Patterns	Case	Refactorings
Dynamic Retry Delay	Dynamic Wait Time	0
	Check Network	5
Push Over Poll	Info Warning FCM	8
Reduce Size	GZIP Compression	14
Cache	Check Metadata	7
	SSL Session Caching	10
	Check Layout size	0
	Passive Provider	11
	Location	
	URL Caching	40
Avoid Extraneous Graphics and Animations	Dirty Rendering	0
Total		95

A total of 95 refactoring opportunities were found in the 7441 Java files, giving an average of one refactoring per  $78.33 \approx 78$  files. Since, in average, the source code of a mobile application inspected has 74.41 Java files, this means an average of around  $0.95 \approx 1$  refactorings per project. This is the case with most projects.

*Case Analysis:* The case with the most refactorings is *URL Caching* with 42.1% of the occurrences. It is then followed by *Check Metadata* (14.7%), *Passive Provider Location* (11.6%), *SSL Session Caching* (10.5%), *Push Over Poll* (8.4%), and *Check Network* (5.3%). EcoAndroid found no opportunities for applying refactorings related to the cases *Dynamic Wait Time*, *Check Layout Size* and *Dirty Rendering*. The combination of patterns with the highest number of associated refactorings is *URL Caching* with *GZIP Compression* with nine projects being affected. This is expected

<sup>4</sup>Collection date: 25 June 2020

since they both look for an invocation of the method *URLConnection#openConnection()* as a first step. The next two combinations with the most occurrences are *URL Caching* with *SSL Session Caching* and *URL Caching* with *Passive Provider Location*, both affecting three mobile applications. With two occurrences, the combination *Check Network* and *URL Caching* is next. With only one occurrence are the combinations: *Info Warning FCM* with *URL Caching*, *Info Warning FCM* and *GZIP Compression*, *Check Network* and *Check Metadata*, *Check Network* and *SSL Session Caching*, *Check Metadata* and *Passive Provider Location*, *Check Metadata* and *URL Caching* and the last one is *Check Metadata* and *GZIP Compression*.

### B. Second Phase: EcoAndroid Refactorings Submitted to Project Maintainers

In the second phase of the evaluation, we sent the refactorings obtained in the first phase to the maintainers of the affected projects. We excluded cases presenting only informational warnings (that introduces `//TODO's` into the source code). Therefore, we did not consider the cases *InfoWarning FCM*, *URL Caching*, and the cases with an informational solution for *Passive Provider Location* (Possible switch to *Passive Provider Location*). We then created pull requests<sup>5</sup> to the original GitHub projects for the remaining refactorings.

A total of 31 pull requests were created, covering 42 refactorings. We received 25 responses (17 were unanswered). Out of the pull requests with feedback from the applications' developers, 20 were accepted and 5 were rejected. This represents an overall acceptance rate of 46.62%. However, when considering only pull requests with feedback from developers, the acceptance rate was of 80%. Out of the five pull requests rejected, two were due the refactoring not saving energy in those cases (one in the pattern *Cache - SSL Session Caching* and *Reduce Size - GZIP Compression*), two were because the class did not use internet but was later fixed (in the pattern *Dynamic Retry Delay - Check Network*) and last one was because the refactoring could break notifications, but was later fixed (in the pattern *Cache - Check Metadata*).

In feedback received from a pull request to the mobile application *Hendroid* related to the case *Check Network*, which had two instances of this pattern, the developer stated that their application depended on another mobile application, changing the target project of the pull request to the mobile application *Hentoid*. While inspecting the new target project, there was one fewer refactoring, altering the number of *Check Network* refactorings to four instead of five. Moreover, in feedback from a pull request related to the case *Check Metadata* to the app *SecondScreen*, a developer suggested creating a pull request to a sister mobile application — *Taskbar* on position #581 in our ordered mobile applications list — adding another refactoring associated with the pattern. In feedback for a pull request related to the case *CheckMetadata* to the mobile application

*ZimLx*, the developer suggested that a pull request to another project would be more efficient, adding another refactoring associated with the pattern to the app *Omega*. This new project was not part of our mobile application list.

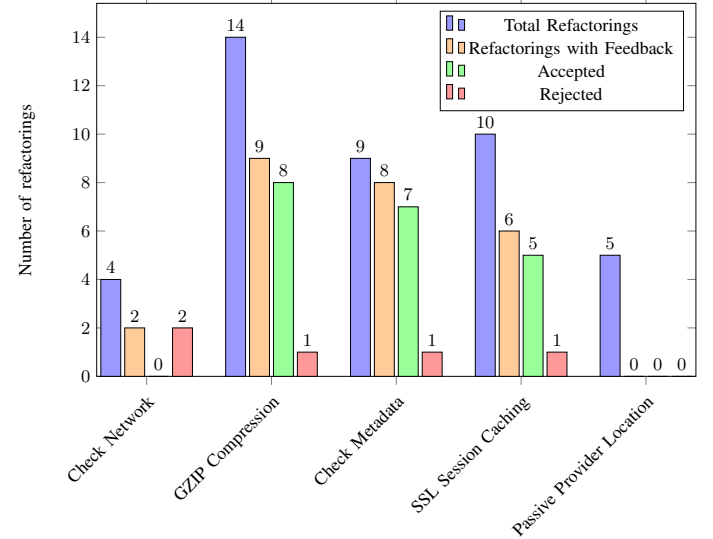


Fig. 2. Number of refactorings proposed by EcoAndroid for each pattern and statistics on the pull requests sent.

Figure 2 presents the number of refactorings sent and the answers given by the maintainers. By observing the bar chart, we can see that the results are mostly positive. The energy pattern with the highest percentage of acceptance is *Check Metadata* with 78%, followed by *GZIP Compression* (57%) and *SSL Session Caching* (50%). The other two energy patterns did not have any accepted pull requests (nor feedback from the maintainers). When considering the percentage of rejections, the energy pattern with the highest value is *Check Network* with 50%, followed by *Check Metadata* (11.1%), *SSL Session Caching* (10%), and *GZIP Compression* (7.14%). It should be noted that we only obtained responses for 60% of the pull requests.

### C. Third Phase: User Study

The final phase of the evaluation was a *user study* to validate the usefulness and usability of EcoAndroid. We wished to answer two research questions:

**RQ1:** Is it easier and/or quicker to apply energy patterns when using EcoAndroid?

**RQ2:** Is EcoAndroid usable for developers?

1) *Structure and Setup:* We divided the user study into two parts that considered different energy patterns. Out of the 10 energy patterns supported by EcoAndroid, only the ones that did not insert `//TODO's` into the source code and that had any occurrences in the first phase of the evaluation were considered. This left us with 5 energy patterns to examine.

<sup>5</sup>For a real and full example, see <https://github.com/farmerbb/Taskbar/pull/138>.

Due to the complexity in understanding the changes required in a short amount of time, the *Check Network* energy pattern was excluded. This left us with 4 energy patterns, two per part. The first part covers the *Cache - Check Metadata* and *Reduce Size - GZIP Compression* energy patterns and the second part covers *Cache - SSL Session Caching* and *Cache - Passive Provider Location* energy patterns. For each energy pattern, a GitHub project was chosen to be used in the study: from the projects with occurrences identified in the first phase, we chose the one with most GitHub stars. The number of participants was defined considering the work by J. Nielsen's on usability and user tests [35], which states that a sufficient number for a usability test is five. We decided to set the number of participants per part to six, due to the need to divide evenly between two groups the users. This means that our user study had a total number of twelve participants. Out of the twelve participants (10 females and 2 males), 10 are computer science master students and 2 are software professionals, with a bachelor degree in computer science. The tasks were performed using Android Studio (version 4.1.1) on a MacBook Air with macOS Catalina (version 10.15.7). Due to the imposed COVID-19 social distancing restrictions, users participated remotely using Zoom's remote control feature. For each part, the users were divided into two distinct groups: a test group and a control group. Both groups had access to the same system and the same version of Android studio. However, the test group was given access to the EcoAndroid plugin while the control group was not. For the convenience of all the participants, access to the catalog from where the energy patterns were mainly retrieved from [5] and to specific Android documentation for each energy pattern was given.

2) *Tasks and Participants*: Participants were first given 10 minutes to read through a short document explaining in detail the tasks in the user study, with brief explanations of the energy patterns involved in the task and with an example of the pattern being used. Then, to apply both energy patterns, participants were given a maximum of 45 minutes to detect in the selected project where to apply the refactoring and to actually apply it. If in the first 10 minutes of this part, participants could not detect where the energy pattern was to be applied, they could ask for hints to help them figure it out. In the last 5 minutes, participants were asked to answer a questionnaire to better understand their experience in this study.

3) *Results*: We collected information during the execution of the tasks proposed and at the end, by asking participants to fill in a questionnaire. During the execution of the tasks, we measured whether or not participants were able to detect where energy patterns could be applied and whether they could change the code correctly, counting the time to perform each step separately. Tables IV and V present the data obtained for both parts of the study.

TABLE IV  
USER STUDY RESULTS: PART 1.

Group	Task 1		Task 2	
	Time to detect (min)	Time to solve (min)	Time to detect (min)	Time to solve (min)
Test	9.67	5.3	5	2.33
Control	10.67	8.33	2.67	4

*Part 1*: In the first pattern, only 1 participant was able to detect the problem (in both groups). In the test group, two participants were able to solve the problem. In the control, all three participants were able to solve the problem. In the second pattern, every participant was able to detect and solve the problem. Out of the three participants in the test group, only two used the plugin to execute the first task and all used the plugin to execute the second task. While the plugin was accessible during the test, it was not mandatory to apply the pattern with it. With this, one participant in the test group solved attempted to solve problem manually. As it was expected, the time to solve the problem in the test group is shorter than the time in the control group since no manual coding had to be done.

TABLE V  
USER STUDY RESULTS: PART 2.

Group	Task 1		Task 2	
	Time to detect (min)	Time to solve (min)	Time to detect (min)	Time to solve (min)
Test	7.67	1	4.67	1
Control	5	1.33	6.67	1.67

*Part 2*: Every participant was able to detect and solve the problem in both groups. Since the alterations to apply both energy patterns do not entail as many alterations as in the first part, the difference between the time to solve the pattern is not as significant as before. However, in average, the control group requires more time than the test group, which suggests that the plugin can save time to developers.

*Questionnaire*: Regarding the plugin, every participant stated that the comments added by EcoAndroid were highly necessary in order to understand the changes made to the source code. With a classification from 1 (strongly disagree) to 5 (strongly agree), when asked if a web link to documentation further explaining the refactoring would be helpful (for example, to the documentation of the class) the average of answers was 4.33. Also with the same classification, the answer to whether the plugin adds enough comments to the source code, the answer given the users was 3.66. As mentioned, the participants of the study had available a document with descriptions and documentation of classes which objected were altered during the refactoring process. During conversation at the end

of the study, three participants mentioned that the document had a significant impact in the understanding and application of the task. Regarding the complexity of the energy patterns, the *Check Metadata* pattern was the hardest to understand, which can be seen by the percentage of participants that were able to detect and solve this problem. It is also the energy pattern with the highest time to both detect and solve the task in both groups. Nearly every participant reported that the hardest part of the user test was understanding the conditions which the energy pattern should be applied.

One of the main disadvantages reported is the fact that warnings are easily missed. This is not necessarily a problem from the plugin, since it uses the IDE system for warnings; moreover, this problem might have been exacerbated by the participants being only looking for mistakes, rather than being actively coding during the task. Another disadvantage identified was the potential lack of comments added in the refactoring for a clearer understanding of the change. The main advantages reported by the participants are the quickness of the alterations done and how the tool is integrated in the coding environment (the developer does not need to run anything to see the results from the inspection, only needing the plugin installed).

#### D. Answers to Research Questions

**RQ1: Is it easier and/or quicker to apply energy patterns when using EcoAndroid?:** In telling the participants that they had EcoAndroid at their disposal, a significant number of the participants lost time in the detection part of the task, trying to find warnings along the source code instead of looking for the right place to implement the energy pattern. This can be seen by the fact that, generally, the test group took more time than the control group in detecting where to apply the first pattern. As reported in the questionnaire, most participants felt that the warnings could be easily missed; this could have contributed to increase the detection time. However, on the second task, the participants were more familiar with the environment and the control group took longer to detect where to apply the patterns. When comparing the solving times between the two groups, the test group was always quicker, which indicates that EcoAndroid saves time. The application with the energy pattern consists of clicking on the warning presented to the user. This would mean the time to solve the problem is close to 0. This does not happen since some participants, even if they used EcoAndroid, tried to solve the pattern manually to see the differences between their implementations and the one given by the plugin. In conclusion, using the EcoAndroid is, in average, fast and always easier to apply energy patterns. When the developer is accustomed to the way EcoAndroid presents their warnings, the detection time of the problem is faster.

**RQ2: Is EcoAndroid usable for developers?:** Out of six participants of the test group, four were able to detect the energy pattern and five were able to apply the refactoring. It should be noted that participants that were not able to finish the tasks were all applying the same energy pattern — *Cache - Check Metadata*, which had the lowest level of

understanding by the participants. Regarding questions about the information present in the warning name, information present in the comments and amount of comments, the average feedback was mostly positive. From the feedback obtained, participants would like EcoAndroid to introduce more comments. When asked if they would recommend EcoAndroid to Java developers, the average answer was 4.83 (using the same classification as before). Most participants felt that the plugin was ready to use.

#### V. CONCLUSION

EcoAndroid is an Android Studio plugin capable of detecting and correcting energy-related code smell, supporting a total of ten refactorings, over five energy patterns, and where two are informational warnings. When using the plugin to inspect a set of mobile applications, it found refactoring opportunities in 35% of the set, having an average of one refactoring per 78 files. When evaluating EcoAndroid by sending pull requests, the feedback given by mobile applications developers was mostly positive. Out of forty two pull requests sent where twenty five have feedback, only five were rejected. This concludes an 80% acceptance rate in pull requests with response and an 46.62% acceptance rate when including all pull requests sent.

We proposed to answer a set of research questions that are answered along the chapters. Next is a summarized answer for each question.

**RQ1: What energy patterns are already known?:** Table I presents a summary of energy patterns already documented, specifying the paper they are reported in.

**RQ2: What are the most relevant energy patterns to support?:** The patterns selected should not need input from the user in order to save energy, should not imply big alterations to the source code and should not include a new functionality. The energy patterns are present in the catalog by Cruz et al. [5].

**RQ3: Are there existing tools that automatically apply energy patterns to the source code of mobile applications?:** Table II lists examples of energy-specific refactoring tools. The differences between EcoAndroid and these tools are the IDE chosen to implement the plugin on - our plugin is compatible with Android Studio, the official IDE for Android development, and IntelliJ, and the energy patterns it chose to support. Another different is the way it executes the inspection and refactoring, through the PSI API, since none of these tools chose this path.

**RQ4: What are the challenges in automatically applying energy patterns?:** In some cases, since this is a refactoring tool and the alterations performed by it should never be too extensive, we face the challenge of not being able to wise to apply the pattern. The approach chosen was to, in these cases, add a comment before the method where the code smell is located explaining the change, usually with a web link for the documentation supporting it. We may also not be able to apply the pattern ourselves, in the case of the *Push Over Poll* energy pattern since the registration of the class in Firebase

is needed. To fix the problem, the same approach as before is used: adding a comment before the method.

Some suggestions for future work include supporting the remaining energy patterns of the catalog [5], running the EcoAndroid with an energy profiling tool to verify the energy savings existence. Other possible work is adding Kotlin support for the energy patterns in question (the Kotlin language is supported by the PSI API).

## REFERENCES

- [1] A. Ribeiro and J. F. Ferreira, "EcoAndroid: An Android Studio plugin for developing energy-efficient Java mobile applications," 2021, submitted to publication.
- [2] C. Wilke, S. Richly, S. Götz, C. Piechnick, and U. Aßmann, "Energy consumption and efficiency in mobile applications: A user feedback study," in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. IEEE, 2013, pp. 134–141.
- [3] "Forbes so you think we're reducing fossil fuel use? think again," <https://www.forbes.com/sites/jamesconca/2019/07/20/so-you-think-were-reducing-fossil-fuel-think-again/>, accessed: 2020-12-01.
- [4] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*. ACM, 2009, pp. 280–293.
- [5] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, pp. 1–27, 2019.
- [6] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for energy efficiency: a reflection on the state of the art," in *Proceedings of the Fourth International Workshop on Green and Sustainable Software*. IEEE Press, 2015, pp. 29–35.
- [7] M. Gottschalk, J. Jelschen, and A. Winter, "Saving energy on mobile devices by refactoring," in *EnviroInfo*, 2014, pp. 437–444.
- [8] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 46–57.
- [9] R. W. Ahmad, A. Gani, S. H. A. Hamid, F. Xia, and M. Shiraz, "A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues," *Journal of Network and Computer Applications*, vol. 58, pp. 42–59, 2015.
- [10] C. Seo, S. Malek, and N. Medvidovic, "An energy consumption framework for distributed java-based systems," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 421–424.
- [11] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, 1999, pp. 2–10.
- [12] Y.-F. Chung, C.-Y. Lin, and C.-T. King, "Aneprof: Energy profiling for android java virtual machine and applications," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 2011, pp. 372–379.
- [13] L. Cruz and R. Abreu, "Emaas: energy measurements as a service for mobile applications," in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*. IEEE Press, 2019, pp. 101–104.
- [14] K. S. Banerjee and E. Agu, "Powerspy: fine-grained software energy profiling for mobile devices," in *2005 International Conference on Wireless Networks, Communications and Mobile Computing*, vol. 2. IEEE, 2005, pp. 1136–1141.
- [15] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 92–101.
- [16] D. Li and W. G. Halfond, "An investigation into energy-saving programming practices for android smartphone app development," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 2014, pp. 46–53.
- [17] S. Habchi, G. Hecht, R. Rouvoy, and N. Moha, "Code smells in ios apps: How do they compare to android?" in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 110–121.
- [18] K. Stroggylos and D. Spinellis, "Refactoring—does it improve software quality?" in *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 10–10.
- [19] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 50.
- [20] L. Cruz, R. Abreu, and J.-N. Rouvignac, "Leafactor: Improving energy efficiency of android apps via automatic refactoring," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 205–206.
- [21] L. Cruz and R. Abreu, "Using automatic refactoring to improve energy efficiency of android apps," *arXiv preprint arXiv:1803.05889*, 2018.
- [22] M. Couto, J. Saraiva, and J. P. Fernandes, "Energy refactorings for android in the large and in the wild," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 217–228.
- [23] "Autorefactor," <http://autorefactor.org/>, accessed: 2020-12-01.
- [24] "Aeon: Automated android energy-efficiency inspection," <https://plugins.jetbrains.com/plugin/7444-aeon-automated-android-energy-efficiency-inspection>, accessed: 2020-12-01.
- [25] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: An energy-aware refactoring approach for mobile apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, 2017.
- [26] E. Iannone, F. Pecorelli, D. Di Nucci, F. Palomba, and A. De Lucia, "Refactoring android-specific energy smells: A plugin for android studio," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 451–455.
- [27] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," in *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung—MMSM*, vol. 2014, 2014.
- [28] O. Le Goaër, "Enforcing green code with android lint."
- [29] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 115–126.
- [30] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of android applications along their evolution (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 236–247.
- [31] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 487–491.
- [32] "Program structure interface (psi)," [https://www.jetbrains.org/intellij/sdk/docs/basics/architectural\\_overview/psi.html](https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html), accessed: 2020-12-01.
- [33] P. C. Dinh and S. Boonkrong, "The comparison of impacts to android phone battery between polling data and pushing data," in *IISRO Multi-Conferences Proceeding. Thailand*, 2013, pp. 84–89.
- [34] "F-droid," <https://f-droid.org>, accessed: 2020-12-01.
- [35] "How many test users in a usability study," <https://www.nngroup.com/articles/how-many-test-users/>, accessed: 2020-12-01.