EcoAndroid: An Android Studio Plugin for Developing Energy-Efficient Java Mobile Applications

Ana Ribeiro* and João F. Ferreira†
INESC-ID & Instituto Superior Técnico
University of Lisbon, Lisbon, Portugal
Email: *anasofiaribeiro@tecnico.ulisboa.pt, †joao@joaoff.com

Alexandra Mendes
University of Beira Interior, Covilhã, Portugal
HASLab, INESC TEC, Porto, Portugal
Email: alexandra@archimendes.com

Abstract—Mobile devices have become indispensable in our daily life and reducing the energy consumed by them has become essential. However, developing energy-efficient mobile applications is not a trivial task. To address this problem, we present EcoAndroid, an Android Studio plugin that automatically applies energy patterns to Java source code. It currently supports ten different cases of energy-related refactorings, divided over five energy patterns taken from the literature. We used EcoAndroid to analyze 100 Java mobile applications (≈1.5M LOC) and we found that 35 of the projects had a total of 95 energy code smells. EcoAndroid was able to automatically refactor all the code smells identified.

Index Terms—Green Software, Energy Consumption, Energy Patterns, Code Smells, Android, Refactoring

I. INTRODUCTION

Mobile devices have become a fundamental accessory in current day-to-day life. They are used as credit cards, work tools, educational helpers, among many other 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. Indeed, Wilke et al. [1] analyzed comments left in the Google Play market place for Android applications and concluded that 18% of the complaints were related to energy problems. The reduction of the energy consumed by mobile devices has thus become an important non-functional requirement [2], [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, this is a complex task since a lot of factors can influence energy consumption, such as: 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, as 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]. An extremely useful resource is Cruz and Abreu’s catalog of energy-related patterns [5]. This 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 far from trivial and can be time-consuming.

To address this problem, we present EcoAndroid, 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, distributed over five energy patterns taken from the literature [5].

We used EcoAndroid to analyze 100 Java projects (≈1.5M LOC) from F-Droid, a Free and Open Source Android App Repository, and we found that 35 of the projects had a total of 95 energy code smells detected by the plugin. EcoAndroid was able to automatically refactor all the code smells identified.

EcoAndroid is an extendable Android Studio plugin, created to assist developers in creating energy-efficient mobile applications. Since it is open source [1] it can also be extended by the research community to explore new techniques that help in the creation of more energy-efficient mobile applications. EcoAndroid is currently available in the JetBrains Marketplace: https://plugins.jetbrains.com/plugin/15637-ecoandroid

Illustrative Example: To illustrate the type of refactoring that EcoAndroid is capable of, we show in Figure 1 an example of a refactoring suggested and applied by EcoAndroid. It consists in the application of the Cache - Check Metadata energy pattern in the Android mobile application Taskbar, which puts a start menu and recent apps tray on top of the screen that is accessible at any time. Taskbar is a popular application: at the time of writing, its GitHub project has 405 stars and it has been downloaded more than 1 million times from the app store Google Play. Figure 1a shows the original source code, where the code smell was detected. There is an opportunity to optimize the energy efficiency of the code by caching the object bundle and only executing the code if the object has changed. Figure 1b shows the source code after the refactoring automatically performed by EcoAndroid.

---

1EcoAndroid Github Repository: https://github.com/sr-lab/EcoAndroid
public final class TaskerConditionReceiver
extends BroadcastReceiver {
    private Bundle lastbundle = null;

    @Override
    public void onReceive(Context context, Intent intent) {
        if (U.isExternalAccessDisabled(context)) return;

        BundleScrubber.scrub(intent);
        final Bundle bundle = intent.getBundleExtra(com.twofortyfouram.locale.api.Intent.EXTRA_BUNDLE);

        if (PluginBundleManager.isBundleValid(bundle)) {
            // bundle hasn't changed: we can safely return
            return;
        }

        updateValues(intent);

        if (PluginBundleManager.isBundleValid(lastbundle)) {
            // bundle hasn't changed: we can safely return
            return;
        }

        lastbundle = intent.getBundleExtra(com.twofortyfouram.locale.api.Intent.EXTRA_BUNDLE);
    }
}

C. Energy-Related Refactoring for Java Source Code

This subsection focuses on refactoring tools that optimize the energy efficiency of Java mobile applications. Table II lists the tools and their respective environment.

**Leafactor** [11], [12] is a tool that automatically refactors Android mobile applications source code to reduce energy consumption. Leafactor is an Eclipse plugin while EcoAndroid is compatible with both Android Studio and IntelliJ. The 5 refactorings supported by Leafactor are acquired from a previous study by the same authors about the effect of performance-based practices on mobile applications’ energy consumption [8]. **Chimera** [13] covers 11 energy-greedy code patterns. It uses Lint[^3] for the inspection phase and Autorefactor [14] for the refactoring phase. A new aspect about this project is how broad the evaluation is, inspecting more than 600 mobile applications. In their paper, Couto et al. [13] also compare the energy savings of combinations of refactorings. **AEON** (Automated Android Energy-Efficiency Inspection) [15] is a support framework, compatible with IntelliJ and Android Studio. It automatically detects energy inefficiencies in Android mobile applications and helps developers fix those inefficiencies. It also supports developers in verifying, refactoring and profiling such inefficiencies. **EARMO** [16] is an approach that detects and corrects energy-related anti-patterns 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 via refactoring-tool-support provided by Android Studio and

[^3]: Lint is a code analysis tool [developer.android.com/studio/write/lint].
<table>
<thead>
<tr>
<th>Tool/Approach Name</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leafactor</td>
<td>Eclipse</td>
</tr>
<tr>
<td>Chimera</td>
<td>Intellij, Android Studio</td>
</tr>
<tr>
<td>AEON</td>
<td>Intellij, Android Studio</td>
</tr>
<tr>
<td>EARMO</td>
<td>Intellij, Manually</td>
</tr>
<tr>
<td>aDoctor</td>
<td>Eclipse, Android Studio</td>
</tr>
<tr>
<td>Greeness category</td>
<td>Android Studio (Android lint)</td>
</tr>
<tr>
<td>HOT-PEPPER</td>
<td>-</td>
</tr>
</tbody>
</table>

**TABLE I: Energy-Specific Refactoring Tools.**

When that is not possible, the changes have to be applied manually. Palomba et al. propose aDoctor [17], a tool that identifies 15 Android-specific code smells from a catalog by Reimann et al. [18]. It is built on top of the Eclipse Java Development Toolkit (JDK). aDoctor was also extended as an Android Studio plugin supporting 5 energy-related refactorings. Le Goaër presents a new category in Android lint entitled Greeness [19]. This category has 11 checks, which can be viewed as an inspection in Android Studio. HOT-PEPPER [20] is able to detect and correct 3 types of Android-specific code smells. It uses PAPRIKA [21], a static analysis tool 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.

When compared to the tools above, EcoAndroid is the first to support refactorings associated with the energy patterns Dynamic Retry Delay, Push Over Poll, Cache, and Avoid Extraneous Graphics and Animations.

### III. ECOANDROID: ARCHITECTURE AND IMPLEMENTATION

This section describes the architecture of EcoAndroid and the cases that were implemented.

**A. Overview**

EcoAndroid is an Android Studio plugin that suggests automated refactorings with the aim of reducing energy consumption of Java android applications. Android Studio is an IDE built on JetBrains’ Intellij IDEA. Due to this fact, the EcoAndroid plugin is also compatible with Intellij. Android Studio was chosen because it is the official IDE for Android app development, making it the best choice for maximizing the impact of our work.

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 Stucture Interface (PSI) of Intellij was used [23]. 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 allow the detection of possible energy improvements and the refactoring itself.

An Intellij’s plugin can have two types 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 show warnings along the source code but needs to be run manually by the user. Since we want EcoAndroid to be a tool that developers use interactively during development, every case that the plugin supports is implemented as a local inspection [2]. The plugin supports a total of five energy patterns, divided in ten separated cases. Each case is implemented as a local inspection in the plugin.

Figure 2 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 an Abstract Syntax Tree (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).

**B. Cases Implemented**

The five energy patterns supported are a subset of the ones presented in the catalog by Cruz and Abreu [5]. The energy patterns are: Dynamic Retry Delay, Push Over Poll, Reduce Size, Cache, and Avoid Extraneous Graphics and Animations. These were chosen because they all seem feasible.

Despite being designed as a tool to be used interactively in the IDE, it is possible to run EcoAndroid in batch mode. This is useful to process many projects, as we have done in our evaluation (see below).
to be implemented with little to none human intervention and also because they are not implemented by any of other existing refactoring tools. As described below, in some of these patterns more than one case was implemented (there are 10 cases in total). EcoAndroid provides two types of warnings: informational ones and non-informational ones. The first type does not have an automated refactoring associated. This is because either i) the suggestion is impossible to implement without further information (e.g. in the case of the Push Over Poll energy pattern, registration of the class in Firebase is needed) or ii) the required changes affect too much code. For these cases, if the developer wishes to follow EcoAndroid’s suggestion and implement the changes manually, the plugin introduces a TODO comment so that the change is listed in the IDE’s TODO window. The second type of warning has an automated refactoring associated and will change the code by applying the identified energy pattern. Figure 4 shows the 10 cases supported by EcoAndroid. Cases with gray background represent informational cases, where no source code is altered. We briefly describe all the cases below, but due to space restrictions, we only discuss in detail two cases: Dynamic Wait Time and GZIP Compression. For more details on how EcoAndroid refactors the code, the reader can consult the entries of the considered patterns on Cruz and Abreu’s catalog [5].

1) Dynamic Retry Delay: The goal 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 spread over time the accesses. If the access is successful, the interval should not be changed. This energy pattern has two cases: Dynamic Retry Delay and Check Network. We describe in detail the first case.

a) Dynamic Wait Time: The first case of the Dynamic Retry Delay energy pattern is named, by the plugin, Dynamic Wait Time. The interval between threads sleep should grow exponentially and not stay constant, decreasing the chance of trying to access a resource that most likely went down. 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, 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. When processing this example, the plugin flags this as a problem, showing up as a warning on the variable backOffSeconds. In this case, EcoAndroid presents the user with two warnings:

• Warning 1: “EcoAndroid: Dynamic Retry Delay Energy Pattern - information about a new approach to implement it”

The informational warning shown does not alter any source code: it only adds a comment with a link that further explains how to use the WorkRequest class instead of using the Thread class. This is shown in Listing 2.

• Warning 2: “EcoAndroid: Dynamic Retry Delay Energy Pattern - 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 the accessAttempts variable. Finally, 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.
Fig. 3: Energy patterns supported by EcoAndroid. Cases with gray background represent informational cases, where no source code is altered.

2) Push Over Poll: A push notification establishes and maintains a connection with a server over the Internet, allowing 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 check 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 [24], 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. Otherwise, the difference between these two mechanisms is not as significant. 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: Graphics and animations are resources associated with an high energy consumption. The intent of the Avoid Extraneous Graphics and Animations’s pattern is to reduce the usage of these resources as much as possible. This is particularly relevant for any resource associated with an high energy consumption that does not 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*. EcoAndroid supports only a simple case: it is able to change the rendering mode from `GLSurfaceView.RENDERMODE_CONTINUOUSLY` to `GLSurfaceView.RENDERMODE_WHEN_DIRTY`.

### IV. Evaluation

Given that the goal of EcoAndroid is to automatically apply a set of energy patterns to Java source code, we measured how many refactorings EcoAndroid suggests for a realistic set of mobile Java applications.

#### A. Mobile Applications Analyzed

We used Android mobile applications retrieved from [F-droid](https://f-droid.org), 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 and we filtered and ordered them before being used for the evaluation. We processed information relative to 2319 mobile applications from which we filtered 1615 applications with the following characteristics:

- 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.

From the F-Droid applications retrieved, there was a total of 474 mobile applications where the main language was not Java, 89 mobile applications where the GitHub project was archived, and 141 mobile applications with the last commit made to the GitHub project more than 2 years ago. The mobile applications were then sorted by the following order: 1) Percentage of Pull Requests accepted; 2) Date of Last Commit; 3) Total merged Pull Requests; 4) Number of GitHub Stars; 5) 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. This corresponds to a total of 7441 Java files and 1,468,597 LOC. Table III summarizes the main characteristics of the GitHub projects, considering both processed and inspected projects.

#### B. EcoAndroid Refactorings

We were interested 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 informational warnings, whose refactorings introduce TODOs into the source code. We divided this evaluation in three stages: we first processed the top twenty mobile applications, then the following twenty, and then the remaining sixty applications.

During the first stage, it became clear that an application of the *Check Network* energy pattern in the Hacs mobile app did not make sense, since the application did not declare permission to use the internet. We updated EcoAndroid accordingly and this was no longer a problem in the following stages. Also during the first stage, an application of the *Check Metadata* energy pattern in the Hacs mobile application could break some notifications of the application. We updated EcoAndroid to consider the case identified and this was no longer a problem in the following stages. We moved to the second stage and then to the third stage, thus analysing a total of 100 applications. In the second and third stages there were no problems identified and no changes to EcoAndroid were required.

A total of 95 refactoring opportunities were found in 35 projects, in a total of 7441 Java files, giving an average of one refactoring per 78.33 ≈ 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 ≈ 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 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 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 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*.

### V. Threats to Validity

There are two main aspects that may affect the validity of our work and findings. First, due to the complexity of EcoAndroid, there may exist implementation bugs. We extensively tested the tool to mitigate this risk. Moreover, since EcoAndroid was able to refactor a substantial amount of real-world code written by multiple people, we are confident in our

---

3 Collection date: 25 June 2020
TABLE II: F-Droid mobile applications characteristics.

<table>
<thead>
<tr>
<th></th>
<th>GitHub Watchers</th>
<th>GitHub Stars</th>
<th>GitHub Forks</th>
<th>GitHub Contributors</th>
<th>Merged PRs</th>
<th>Closed PRs</th>
<th>% PRs Accepted</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
<td>18.53</td>
<td>185.37</td>
<td>65.27</td>
<td>10.21</td>
<td>39.71</td>
<td>48.20</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>1692</td>
<td>25630</td>
<td>8716</td>
<td>317</td>
<td>2603</td>
<td>3009</td>
</tr>
</tbody>
</table>

|                          |                 |              |              |                     |            |            |                |
| Top 100                  | Min             | 0            | 0            | 0                   | 1          | 1          | 1              |
|                          | Mean            | 6.3          | 38.44        | 11.61               | 8.43       | 5.88       | 5.88           |
|                          | Max             | 29           | 616          | 181                 | 317        | 60         | 60             |

TABLE III: Number of energy opportunities detected by EcoAndroid.

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

VI. CONCLUSION

EcoAndroid is an Android Studio plugin that automatically applies a set of energy patterns to Java source code. We used EcoAndroid to analyze 100 Java mobile applications from F-Droid (≈ 1.5M LOC) and we found that 35 of the projects had a total of 95 energy code smells detected by the plugin. EcoAndroid was able to automatically refactor all the code smells identified. These results suggest that EcoAndroid is useful and is capable of applying energy patterns to real-world mobile applications. Since the scripts that collect and process applications from F-Droid are also distributed with EcoAndroid, we argue that our artefacts are valuable assets for driving reproducible research in automated energy optimization of mobile applications.

Summary of Findings: The development of EcoAndroid and its evaluation gave us several insights. The two most important are:

1) There is a significant number of opportunities to apply energy patterns in real-world Java Android applications. When considering 100 applications taken from the F-Droid repository, EcoAndroid found a total of 95 refactoring opportunities in 35 projects. This suggests that there are many Java Android applications that could benefit from the use of EcoAndroid.

2) From the set of patterns considered by EcoAndroid, the cases with more refactorings are URL Caching and Check Metadata, with 42.1% and 14.7% of occurrences, respectively. The combination of patterns with the highest number of associated refactorings is URL Caching with GZIP Compression (9 out of 100 projects).

Future Work: Some suggestions for immediate future work include submitting refactorings proposed by EcoAndroid to project maintainers (we have started this process and preliminary results are positive); performing user studies to assess the usefulness and usability of EcoAndroid; supporting the remaining energy patterns of Cruz and Abreu’s catalog [5]; and performing experiments involving energy profiling. Android Studio’s energy profiler [27] is an obvious choice that can be explored, but ANEPROF [28] could also be a good option since it is a real-measurement-based power profiling tool specific for Android. Another possible direction is to extend EcoAndroid so that it supports applications written in Kotlin (the Kotlin language is supported by the PSI API).
ACKNOWLEDGMENTS

This work has been supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under projects UIDB/50021/2020 and PTDC/CCI-COM/29300/2017.

REFERENCES


