TÉCNICO LISBOA

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

Ana Sofia Gonçalves Ribeiro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. João Fernando Peixoto Ferreira

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Rui Filipe Lima Maranhão de Abreu

January 2021

Acknowledgments

I would first like to thank my supervisor João F. Ferreira for presenting the theme and for all the support and guidance during this project.

To the friends Instituto Superior Técnico has given me, Inês(s), Mafalda and Rita, thank you for the friendship in the last five years (more like four and I am sorry about that). Without you five, college would not have been the same and would have been much more tedious.

To the rest of my friends, thank you for the tremendous encouragement through this project. To Elisabete, Sérgio and Margarida, thank you for a friendship that turned into a family I know I will take for the rest of my life. To Joana Pinto, who has been my friend since fifth grade or seventh grade, we will actually never know. To Bruno, my person, who has been my best friend for the last 17 years, thank you for never doubting my abilities to achieve something even in all the times I never believed I could.

To my family, thank you for your support and understanding all these years, especially in the last one. To Bela, thank you for all you have done and keep doing for me. To my sisters, Joana and Sara, thank you for dealing with me at my worst. Even if you think my thesis is only turning on the power save mode on your phone. Finally, to my mother and father, thank you for being the best role models a daughter could ask for and for trying to guide me to the best possible future. Even if I didn't always see it myself.

To each and every one of you – Thank you.

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.

Keywords

Sustainable Software, Green Software, Energy Consumption, Energy Patterns, Code Smells, Refactoring.

Resumo

Os dispositivos móveis tornaram-se indispensáveis no nosso dia a dia e reduzir a energia consumida por eles tornou-se essencial nos últimos anos. Por razões económicas e ambientais, além de aprimorar a experiência do utilizador, estender a duração da bateria tornou-se um requisito não funcional com o qual os programadores se devem preocupar. No entanto, desenvolver aplicações móveis com baixo consumo de energia não é uma tarefa trivial. Para resolver esse problema, apresentamos EcoAndroid, um plugin para Android Studio que aplica automaticamente um conjunto de padrões de energia a código-fonte Java. Actualmente, o plugin suporta dez casos diferentes de refactorações relacionadas ao consumo de energia, relativos a cinco padrões de energia retirados da literatura. Usámos o EcoAndroid para analisar 100 aplicações móveis Java do F-Droid e descobrimos que 35 dos projetos tinham um total de 95 *code smells* de energia detectados pelo plugin. Usámos também o EcoAndroid para refactorizar automaticamente todos os *code smells* identificados. Enviámos as 42 refactorizações que introduziram correcções de código (e não apenas avisos informativos) como *pull requests* para os responsáveis dos respectivos projetos. De um total de 42 *pull requests*, recebemos respostas para 25 deles (59,5 %); desses, 20 (80 %) foram aceites e incorporados nos projetos originais. No total, contribuímos para melhorar a eficiência energética de 12 aplicaçãoes móveis Android diferentes. Estes resultados, juntamente com os resultados obtidos num estudo de usabilidade com 12 participantes, mostram que o EcoAndroid é útil, utilizável, e as alterações propostas pela ferramenta são facilmente aceites pelos programadores.

Palavras Chave

Software Sustentável, Software Verde, Consumo de Energia, Padrões de Energia, *Code Smells*, Refactorização.

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

Contents

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 [1] 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.

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 [2]. These resources are also finite and, by saving energy, we are contributing to more affordable energy for future generations. According to the World Health Organization (WHO) [3], air pollution makes seven million casualties word wide every year (9 out 10 people breath air containing an excessive level of pollutants). Given that the main cause of air pollution is the burning of fossil fuels, saving energy has an impact not only on the environment but also on our health.

Economically, saving the energy one consumes will decrease one's utility bills. Moreover, a generalized increase in energy consumption might lead to corporation investments to fulfill the energy demand. Subsequently, this can lead to higher energy prices and more expensive utility bills.

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 (for example, 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.

1.1 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 available mobile platforms (Android or iOS) and which languages are the most used in each platform (to understand how we can maximize our impact); 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[1] 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.

A user study with 12 participants was done to check EcoAndroid's usability. The results show that using the plugin reduces the time required to identify and fix energy-related code smells. In average, there was a saving of 1.43 minutes. Every participant stated that they found EcoAndroid to be usable and the alterations made by it understandable.

Research Questions. 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?

[1] Available in the JetBrains store: https://plugins.jetbrains.com/plugin/15637-ecoandroid.

Summary. 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 20 code smells;
- A study of the most common code smells in 100 Android mobile applications, when considering 5 energy-related patterns.

Illustrative Example An example of a contribution made by EcoAndroid is the application of the *Cache - Check Metadata* energy pattern in the Android mobile application Taskbar[2], 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 283 stars and it has been downloaded more than 500,000 times from the app store Google Play.

Listing 1 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. Listing 2 shows the source code after the refactoring automatically performed by EcoAndroid. We submitted these changes as a pull request that was accepted and merged by the maintainers.[3]

```
public final class TaskerConditionReceiver extends BroadcastReceiver {
    @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)) { ... }
        ...
    }
}
```

Listing 1: Taskbar Example - code smell detected.

[2]Taskbar (Google Play): https://play.google.com/store/apps/details?id=com.farmerbb.taskbar.
[3]Pull Request: https://github.com/farmerbb/Taskbar/pull/138.

```
public final class TaskerConditionReceiver extends BroadcastReceiver {
    private Bundle lastbundle = null;

    @Override
    public void onReceive(Context context, Intent intent) {
        if(U.isExternalAccessDisabled(context)) return;
        if (lastbundle.equals(intent.getBundleExtra(com.twofortyfouram.locale.api.Intent.EXTRA_BUNDLE))) {
                // bundle hasn't changed: we can safely return
                return;
        }
        updateValues(intent);
        ...
        if(PluginBundleManager.isBundleValid(lastbundle)) { ... }
        ...
    }

    private void updateValues(Intent intent) {
         lastbundle =
            intent.getBundleExtra(com.twofortyfouram.locale.api.Intent.EXTRA_BUNDLE);
     }
}
```

Listing 2: Taskbar Example - energy pattern applied.

1.2 Thesis Outline

This thesis is is organized as follows: Chapter 2 (Background and Related Work) overviews work previously done in the area of energy and Java source code refactoring, such as tools that already exist. It covers three main topics: energy consumption and energy profiling, energy patterns for mobile applications, and refactoring of Java source code. Chapter 3 (EcoAndroid: An Android Studio Plugin) describes the specifications of the tool created and the energy patterns the tool supports. Chapter 4 (Evaluation) shows the results of evaluating the plugin created, including a user study to assess the usability of the tool. Chapter 5 (Conclusion) summarizes the work done in this thesis and possible future work opportunities, such as extensions to the tool.

2

Background and Related Work

Contents

This chapter presents work previously done in the context of this thesis. First, Section 2.1 presents available energy profiling tools. Section 2.2 lists energy patterns already documented, focusing on mobile applications. Section 2.3 provides a summary of the most common mobile application coding environments (for example, the most popular IDEs) and the programming languages most used for each platform. At last, Section 2.4 presents Java refactoring tools, with a subsection for energy-specific refactoring tools.

2.1 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 schemes exploit a software module to collect mobile component's power usage statistics to construct power models to estimate the application's energy consumption. Hardware-based schemes use external hardware equipment, which are expensive, labor-intensive, and non-scalable compared with software-based solutions.

Seo et al. [10] developed a framework to estimate the energy consumed in Java-based software systems, fitting into the software-based category. The framework takes a component-based development perspective, which makes it well suited for distributed and embedded applications. It allows to estimate the software system's energy consumption at system construction-time and refine it at run-time.

Powerscope [11] is an energy profiling tool that falls into the hardware-based category. It maps energy consumption to program structure in two stages: the data collection stage and the analysis stage. The first phase's objective is to sample the power consumption and the system activity. The second phase's objective is to create an energy profile of the data gathered from the previous stage.

Aneprof (Android Energy Profiler) [12] is a real-measurement-based power profiling tool for Android, belonging to the hardware-based category. It can obtain function-level power distribution and distinguish power usage among threads, Java methods and JVM services. The paper that describes this tool states that their biggest contribution is the ability of separating Java methods and events inside Dalvik[1] virtual machines [12].

EMaaS (Energy Measurements as a Service for Mobile Applications) [13] is a system that measures energy from a mobile application reliably, fitting into the hardware-based category. It is a peer-to-peer cloud-based system that delivers energy measurements as a service for mobile applications. This system addresses the issues of power monitor tools being too complex and the need for reliability energy models to be continuously updated with new data, combining estimations from an energy model.

[1]Dalvik is a discontinued process virtual machine in Android operating system that executes applications written for Android.

PowerSpy [14] is a fine-grained power tool for the Windows operation system. It falls into the hardware-based category. It works by going through two phases: an event tracking and an analysis stage phase. In the event tracking phase, the mobile application that is to be profiled is executed while monitoring the CPU Time, I/O activity and the energy consumption. In the analysis stage phase, the data from the previous phase is analyzed in order to understand how different parts of the system behave.

eLens [15] is a technique used to estimate energy consumption for Android mobile applications, with an accuracy of 90%, belonging to the software-based category. It merges two ideas: program analysis (to determine the paths traversed and track energy-related information during an execution) and per -instruction energy modeling (that enables eLens to obtain fine-grained estimates of application energy). eLens can be integrated into an IDE, such as Eclipse.

Table 2.1 presents a summary of the energy profiling frameworks discussed.

Framework Name	Category	Paper
Seo et al.	Software-Based	[10]
Powerscope	Hardware-Based	[11]
Aneprof	Hardware-Based	[12]
EMaas	Hardware-Based	[13]
PowerSpy	Hardware-Based	[14]
eLens	Software-Based	[15]

Table 2.1: Energy profiling frameworks.

2.2 Energy Patterns for Mobile Applications.

An *Energy Pattern* describes, formally, the alterations needed to reduce the energy consumed by a device. The study by Pinto et al. [6] analyzes research papers published on top software engineering conferences and identifies 11 opportunities to refactor code with the goal of reducing energy consumption. The study also mentions that 6 of these are related to mobile applications.[2]

- *User Interfaces*: Darker colors, such as black, require less energy to display than lighter ones, such as white;
- *CPU Offloading*: To offload CPU intensive computations from a mobile device to the cloud reduces battery usage;
- *HTTP Requests*: HTTP request is the most energy-consuming operation of the network;
- *Software Piracy*: the most commonly used approach for preventing piracy is code obfuscation, which is likely to impact the mobile application energy usage;
- *I/O Operations*: I/O utilities contribute significantly to the energy consumption of a mobile application;

[2]The description for each energy opportunity is retrieved from the research paper "Refactoring for Energy Efficiency: A Reflection on the State of the Art" [6] by Pinto et al.

- *Continuously Running App*: Modern mobile applications are continuously-running, periodically sending and receiving data from servers. Such cumulatively behavior can greatly impact battery usage.

Gottschalk et al. [7][3] presents a study about the energy savings on mobile devices by refactoring the source code according to five energy patterns. The study presents the energy savings of using the discussed energy patterns. The energy savings are measured with three different techniques: *file-based*, *energy profiling* and *delta-b*. When using *delta-b*[4], the Data Transfer pattern presents a 14% saving, Third-Party Advertising a 24.4% saving[5], Binding Resources Too Early an 18.1% saving, and Backlight a 26.85% saving[6]. On the other hand, the Statement Change presents a 10.2% energy consumption increase.

- *Third-Party Advertising*: This concerns integrated code parts within apps which display advertisements during operation. Thereby, advertisements do not have an influence on apps' functionality, but might consume energy through 3G or WiFi connections. If advertisements are deleted, programmers might have to change their business model, but the main functionality remains unaffected;
- *Binding Resources Too Early*: Refers to hardware components, such as WiFi and GPS, which are switched on by apps at an early stage when they are not yet needed by the app or user;
- *Statement Change*: Describes alternative programming statements, such as if and switch, which can be substituted with each other, because they have the same functionality, but potential different energy consumption;
- *Backlight*: Refers to the background color of an app. For different screen technologies (e.g. Super LCD and Super AMOLED) the energy consumption can vary for different background colors;
- *Data Transfer*: Refers to loading data from a server via a network connection, instead of reading prefetched data from the app's storage.

Li et al. [16][7] presented a study about the energy-savings of three programming practices, specific for Android mobile application development. They performed scenarios, with and without these practices, and measured the energy consumption of each scenario.

- *HTTP Request*: HTTP requests are one of the most important among many methods for accessing the Internet. The study concluded that the bundle of small HTTP requests is a good practice for a more energy-efficient app;

[3]The description of each energy pattern is retrieved from the research paper "Saving Energy on Mobile Devices by Refactoring" [7] by Gottschalk et al.
[4]The *delta-B* technique calculates the energy consumption through the battery level, which can be read out with the Android BatteryManager API.
[5]Two mobile applications were used for the experiment; the average was considered.
[6]Two mobile devices were used for the experiment; the average was considered.
[7]The description of each programming practice is retrieved from the research paper "An Investigation into Energy-Saving Programming Practices for Android Smartphone App Development" [16] by Li et al.

- *Use of Memory*: Higher memory usage only slightly increases the average energy consumption of each access. They concluded that allocating a larger cache to reduce the number of accesses to the network could be considered a good practice;
- *Performance Tips*: The idea was to verify if improving run-time performance would also work for reducing energy consumption. They concluded that most performance oriented best practices regarding array lengths, static invocations, and field access also work for reducing energy.

Cruz et al. [8] performed a study to confirm whether or not 8 performance-based practices had an impact on the energy consumed by an Android mobile application, using 6 applications. The study concluded that 5 out of the 8 should be considered for a more energy-efficient mobile applications:[8]

- *ViewHolder*: This pattern is used to make a smoother scroll in List Views, with no lags;
- *DrawAllocation*: It is a bad practice to allocate objects during drawing operations since it can create noticeable lags. The recommended fix is allocating objects upfront and reusing them for each drawing operation;
- *WakeLock*: Wake locks can be used to prevent the screen or the CPU from entering a sleep state. If an application fails to release a wake lock, or uses it without being strictly necessary, it can drain the battery of the device;
- *Recycle*: Collections implemented using singleton resources should be released so that calls to different collection objects can efficiently use these same resources;
- *ObsoleteLayoutParam*: During development, UI views might be refactored several times. In this process, some parameters might be left unchanged even when they have no effect in the view. This causes useless attribute processing at runtime.

In 2019, Cruz et al. [5] presented a catalog of energy patterns that was created based on energy-related changes from developers, both on Android and iOS. These patterns present solutions for energy consumption problems, for example waiting for a WiFi while on a cellular connection to perform a transfer, or giving the user dark mode to reduce the impact of the screen on the energy consumption. The catalog presents 22 energy patterns divided into 5 categories below:[9]

1. Low Power Related

(a) *Power Save Mode*: Provide an energy efficient mode in which user experience can drop for the sake of better energy usage.

[8]The description of each energy pattern is retrieved from the research paper "Performance-based Guidelines for Energy Efficient Mobile Applications" [8].by Cruz et al.

[9]The description of each energy pattern is retrieved from the research paper "Catalog of Energy Patterns for Mobile Applications" [5] by Cruz et al.

(b) *Power Awareness*: Have a different behavior when the device is connected/disconnected to a power station or has different battery levels.

2. Screen Related

(a) *Dark UI Colors*: Provide a dark UI color theme to save battery on devices with AMOLED13 screens

(b) *Enough Resolution*: Collect or provide high accuracy data only when strictly necessary.

(c) *No Screen Interaction*: Whenever possible allow interaction without using the display.

(d) *Avoid Extraneous Graphics and Animations*: Avoid performing tasks that are either not visible, do not have a direct impact on the user experience to the user or quickly become obsolete.

3. Reduce the Number of Accesses/Transfers

(a) *Dynamic Retry Delay*: Whenever an attempt to access a resource fails, increase the time interval before retrying to access the same resource.

(b) *Race-to-idle*: Release resources or services as soon as possible (such as wake locks, screen).

(c) *Open Only When Necessary*: Open/start resources/services only when they are strictly necessary.

(d) *Reduce Size*: When transmitting data, reduce its size as much as possible.

(e) *Batch Operations*: Batch multiple operations, instead of putting the device into an active state many times.

(f) *Cache*: Avoid performing unnecessary operations by using cache mechanisms.

(g) *Decrease Rate*: Increase time between syncs/sensor reads as much as possible.

4. User Related

(a) *User Knows Best*: Allow users to enable/disable certain features in order to save energy.

(b) *Inform Users*: Let the user know if the app is doing any battery intensive operation.

(c) *Manual Sync, On Demand*: Perform tasks exclusively when requested by the user.

5. Reduce Power Greedy Operations

(a) *Push Over Poll*: Use push notifications to receive updates from resources, instead of actively querying resources.

(b) *WiFi Over Cellular*: Delay or disable heavy data connections until the device is connected to a WiFi network.

(c) *Suppress Logs*: Avoid using intensive logging. Previous work has found that logging activity at rates above one message per second significantly reduces energy efficiency.

(d) *Sensor Fusion*: Use data from low power sensors to infer whether new data needs to be collected from high power sensors.

(e) *Kill Abnormal Tasks*: Provide means of interrupting energy greedy operations (e.g., using timeouts, or users input).

(f) *Avoid Extraneous Work*: Graphics and animations are really important to improve the user experience. However, they can also be battery intensive — use them with moderation.

Table 2.2 has a summarized list of every energy pattern, with the paper it is mentioned in, cited in this section.

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

Table 2.2: Energy patterns summary.

Since the goal of this thesis is to implement a refactoring tool capable of automatically applying energy patterns, not every energy pattern detailed in this section is eligible to be supported. The energy patterns chosen should have the following characteristics:

Criterion 1. The energy improvement is achieved without user input (such as providing a low power mode for the user to choose from (*User Knows Best*) or informing the user of any battery intensive operation (*Inform Users*)).

Criterion 2. The transformation of applying the energy pattern does not alter a big portion of the source code. The end result of this project is for the tool to be used by developers and they might be reluctant to automatically apply sub substantial changes on their source code.

Criterion 3. A new functionality is not added to the mobile application (such as creating a *Power Save Mode*).

2.3 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 impact of this project, we focused 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.

The act of refactoring Java code can be done directly on the Java source code form or on the compiled byte code. Since the goal of this project is to support programmers during the development phase to identify opportunities to optimize energy consumption, our focus is on refactoring Java source code.

2.4 Refactoring of 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.

Refactoring Java code is such a popular activity that well-known IDEs, such as Eclipse [20] and IntelliJ [21], have automatic refactoring support. There are several reasons to apply refactoring to a

program. For example, in an object-oriented language like the Java, a common problem is the strong coupling between classes derived from inheritance. Kegel et al. [22] present a solution that refactors inheritance with delegation.

There exist a few tools and libraries that address the needs of developers to perform automated refactoring. In the remainder of this section, we describe those which we find more relevant to our goals. We start with more general tools and finish with tools focused on energy-related problems.

AutoRefactor [23] is an open-source Eclipse plugin to automatically refactor Java source code.

Walkomd [24] is an open-source tool to resolve Java coding style issues. It can be added to the built process, through Maven or Gradle.

Facebook pfff [25] is a set of tools to perform various activities, among which refactoring source code. This tool has support for Java.

Kadabra [26] is a Java-to-Java compilation tool for code instrumentation and transformations controlled by the Lara language. It uses **SPOON** [27], an open-source library that enables refactoring for the Java language.

Code-Imp [28] is an automated search-based refactoring tool for the Java language. It implements over 25 software quality metrics and 14 design-level transformations. It is not supported by an IDE and it is only reachable from the command line.

Faultbuster [29] is an automated code smell refactoring tool set, independent from the programming language. Its core element is a refactoring framework which identifies and restructures critical parts from the viewpoint of refactoring. The refactorings are based on algorithms, supporting a total of 40 Java coding issues. It provides plugins for popular IDEs (Eclipse, Netbeans and IntelliJ IDEA) in order to reach developers. The plugin retrieves the problems from the refactoring framework and shows a list of the problems to the developer.

AsyncDroid [30] is an Eclipse plugin which performs automated refactoring to transform existing improperly-used async constructs into correct constructs. This code smell could result in memory leaks, lost results and wasted energy.

Refactorty [31] is a tool for the detection and correction of code smells. It is based on the Eclipse Modeling Framework (EMF).[10] Refactory either uses IncQuery [32] patterns to query models for certain structures or uses metrics-based calculations. It also generates quick fixes for possible refactorings which then can be executed to resolve particular quality smells in each development phase.

JIAD (Java based Intent Aspect Detector) [33] is a tool that performs automated refactoring by applying design patterns in the code. Each design pattern has a set of rules that aids in the detection of potential changes. If a part of the source code satisfies the rules, an intent-aspect is created and, with the design pattern name in question, it is transferred to an automatic transformation tool.

[10]EMF is an Eclipse-based modeling framework and code generation facility for building tools and other applications based on a structured data model.

Other recent work is an algorithm to perform automated refactoring to the **Null Object design pattern** [34]. It consists of, first, finding the possible candidates for fields that may not have been initialized and, secondly, refactoring the code by applying the Null Object design pattern.

Energy-specific refactoring tools. The concern for reducing the energy consumed by devices has grown in recent years and the number of refactoring tools targeting energy-related code smells has grown along side this concern. Next, we summarize 7 refactoring tools/approaches, targeting energy code smells.

Leafactor [35, 36] is a tool that automatically refactors Android mobile applications source code to reduce energy consumption. The main differences between Leafactor and the tool created in the context of this thesis are the set of energy refactoring it supports and the IDE chosen for the plugin. 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. This study is presented in Section 2.2 (Energy Patterns for Mobile Applications) and is listed in Table 2.2.

Chimera [37] covers 11 energy-greedy code patterns. The paper presenting Chimera [37] also compares the energy savings of combinations of refactorings. It uses the Lint[11] 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:[12]

- *HashMap Usage*: The usage of HashMap is discouraged since there is ArrayMap, a more energy-efficient data structure. The alteration consists in switching HashMap to ArrayMap;
- *Excessive Method Calls*: Method calls can have an impact on performance since it usually involves pushing arguments to the call stack, storing the return value in the appropriate processor's register, and cleaning the stack afterwards. Removing method calls inside loops that can be extracted from them can help performance;
- *Member Ignoring Method*: The issue here is the existence of a non-static method that could be static. The methods that qualify for this change should not access any class fields, should not directly invoke non-static methods and should not be overriding methods;
- *Resource Leak*: This issue covers three energy-greedy code patterns, all particular cases of Wakelock. In this issue, the cases considered are the cases of Sensor, Camera and Media resources, which differ from Wakelock in the way they are released.

AEON (Automated Android Energy-Efficiency Inspection) [38] is a support framework, compatible

[11] *Lint* is a code analysis tool that checks an Android project source files for potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization: `developer.android.com/studio/write/lint`.

[12] The description of the refactorings are retrieved from the paper "Energy Refactorings for Android in the Large and in the Wild" [37] by Couto et al.

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 [39] 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.[13] The refactoring is achieved with support from the refactoring-tool-support of Android Studio and Eclipse. When that was not possible, the changes were applied manually.Anti-patterns considered include:

- *Blob*: A large class that absorbs most of the functionality of the system with very low cohesion between its constituents;
- *Lazy Class*: Small classes with low complexity that do not justify their existence in the system;
- *Long-parameter list*: 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;
- *Refuse Bequest*: A subclass uses only a very limited functionality of the parent class;
- *Speculative Generality*: There is an abstract class created to anticipate further features, but it is only extended by one class adding extra complexity to the design;
- *Binding Resources too early*: Refers to the initialization of high-energy-consumption components of the device, e.g., GPS, WiFi before they can be used;
- *HashMap usage*: From API 19, Android platform provides ArrayMap 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;
- *Private getters and setters*: 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 that translates this call to a virtual method called, which is up to seven times slower than direct field access.

aDoctor [40], a tool proposed by Palomba et al., is able to identify 15 Android-specific code smells from a catalog by Reimann et al. [31]. 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 [41]:[14]

- *Durable WakeLock*: To avoid unnecessary battery consumption, an idle Android device goes on standby. When an app needs to keep the CPU active to complete some background work, the Android API provides "wake-locks" that can be acquired to keep the device awake;

[13]The description of the anti-patterns is retrieved from the paper "EARMO: An Energy-Aware Refactoring Approach for Mobile Apps" [39] by Morales et al.

[14]The description of the refactorings are retrieved from the paper "Refactoring Android-specific Energy Smells: A Plugin for Android Studio" [40] by Iannone et al.

- *Inefficient Data Structure*: A HashMap's key type parameter can be any Object subclass, typically primitive types wrapper classes, like Integer. Almost all method calls on a HashMap let the Android RunTime (ART) to apply the autoboxing continuously and unboxing (the automatic two-way conversion between primitive type with their corresponding wrappers), that determines a non-trivial computational overhead;
- *Internal Setter*: Setter methods are a fundamental component of Object-Oriented programming. They usually accept a single argument that is assigned to an instance variable. A non-static method of the same class that calls a setter of this kind (i.e., with only a single assignment) makes a useless computational effort because it has the access rights to make a direct assignment on that property, possibly causing an energy loss;
- *Leaking Thread*: The Android Runtime (ART) treats an active Thread instance as a Garbage Collector (GC) root, meaning that its memory cannot be reclaimed. Whenever a Thread is stopped (by calling stop() or interrupt()), it ceases to be treated as a GC root, becoming eligible for garbage collection;
- *Member Ignoring Method*: According to the Java Memory Model, a static method is faster than its equivalent non-static one: mainly because the caller object this reference is not passed to static methods, so the reference resolution does not take place. A static method does not access any internal properties (i.e., instance variables and non-static methods). Therefore, if a non-static method does not access any internal properties of its belonging class, it should be set as a static one.

A paper by Le Goaër presents a new category in Android lint entitled **Greenness** [42]. This category has 11 checks[15], which can be viewed as inspections in Android Studio:

- *Everlasting Service*: If someone calls Context#startService() then the system will retrieve the service (creating it and calling its onCreate() method if needed) and then call its onStartCommand(Intent, int, int) method with the arguments supplied by the client. The service will at this point continue running until Context#stopService() or Service#stopSelf() is called. Failing to call any of these methods leads to a serious energy leak;
- *Dark UI*: Developers are allowed to apply native themes for their app, or derive new ones from the latter. This decision has a significant impact on energy consumption since displaying dark colors is particularly beneficial for mobile devices with (AM)OLED screens;
- *Battery-Efficient Location*: Location awareness is one of the most popular features used by apps. The fused location provider is one of the location APIs in Google Play services which combines signals from GPS, WiFi, and cell networks, as well as accelerometer, gyroscope, magnetometer and

[15]The descriptions of the checks are retrieved from the paper "Enforcing Green Code With Android Lint" [42] by Le Goaër.

other sensors. It is officially recommended to maximize battery life;

- *Sensor Leak*: Most Android-powered devices have built- in sensors that measure motion, orientation, and various environmental conditions. The common point of all these sensors is that they are expensive while in use. Their common bug is to let the sensor unnecessarily process data when the app enters an idle state, typically when paused or stopped;
- *Sensor Coalesce*: With SensorManager#registerListener(SensorEven tListener, Sensor, int) the events are delivered as soon as possible. Instead, SensorManager#registerListener(SensorEventListener, Sensor, int, int maxReportLatencyUs) allows events to stay temporarily in the hardware FIFO (queue) before being delivered. The events can be stored in the hardware FIFO up to maxReport-LatencyUs microseconds. Once one of the events in the FIFO needs to be reported, all of the events in the FIFO are reported sequentially. Setting maxReportLatencyUs to a positive value allows to reduce the number of interrupts the AP (Application Processor) receives, hence reducing power consumption, as the AP can switch to a lower power state while the sensor is capturing the data;
- *Bluetooth Low-Energy*: In contrast to classic Bluetooth, Bluetooth Low Energy (BLE) is designed to provide significantly lower power consumption. Its purpose is to save energy on both paired devices but very few developers are aware of this alternative API;
- *Internet In The Loop*: Opening and closing internet connection continuously is extremely battery-inefficient since HTTP exchange is the most consuming operation of the network [14]. This bug typically occurs when one obtain a new HttpURLConnection by calling URL.openConnection() within a loop control structure (while, for, do-while, for-each);
- *Durable Wake Lock*: A wake lock is a mechanism to indicate that your application needs to have the device stay on. The general principle is to obtain a wake lock, acquire it and finally release it. Hence, the challenge here is to release the lock as soon as possible to avoid running down the device's battery excessively;
- *Uncompressed Data Transmission*: Transmitting a file over a network infrastructure without compressing it consumes more energy than with compression;
- *Rigid Alarm*: Applications are strongly discouraged from using exact alarms unnecessarily as they reduce the OS's ability to minimize battery use;
- *Service at Boot-time*: Services are long-living operations, as components of the apps. However, they can be started in isolation each time the device is next started, without the user's acknowledgement. This technique should be discouraged because the accumulation of these silent services results in excessive battery depletion that remains unexplained from the end-user's point of view.

HOT-PEPPER [43] is able to detect and correct 3 types of Android-specific code smells. It uses **PAPRIKA** [44], 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[16], being able to inform the developer which APK is the most energy-efficient version, for a given scenario. The three code smells considered are:

- *Internal Getter/Setter*: Occurs when a field is accessed, within the declaring class, through a getter and/or a setter. This indirect access to the field may decrease the performance of the app;
- *Member Ignoring Method*: This method does not access an object attribute or is not a constructor, it is recommended to use a static method in order to increase performance. The static method invocations are about 15%–20% faster than dynamic invocations;
- *HashMap Usage*: ArrayMap and SimpleArrayMap as replacements of the standard Java HashMap. They are supposed to be more memory-efficient.

Figure 2.1 presents a summary of the 7 energy-specific refactoring tools discussed.

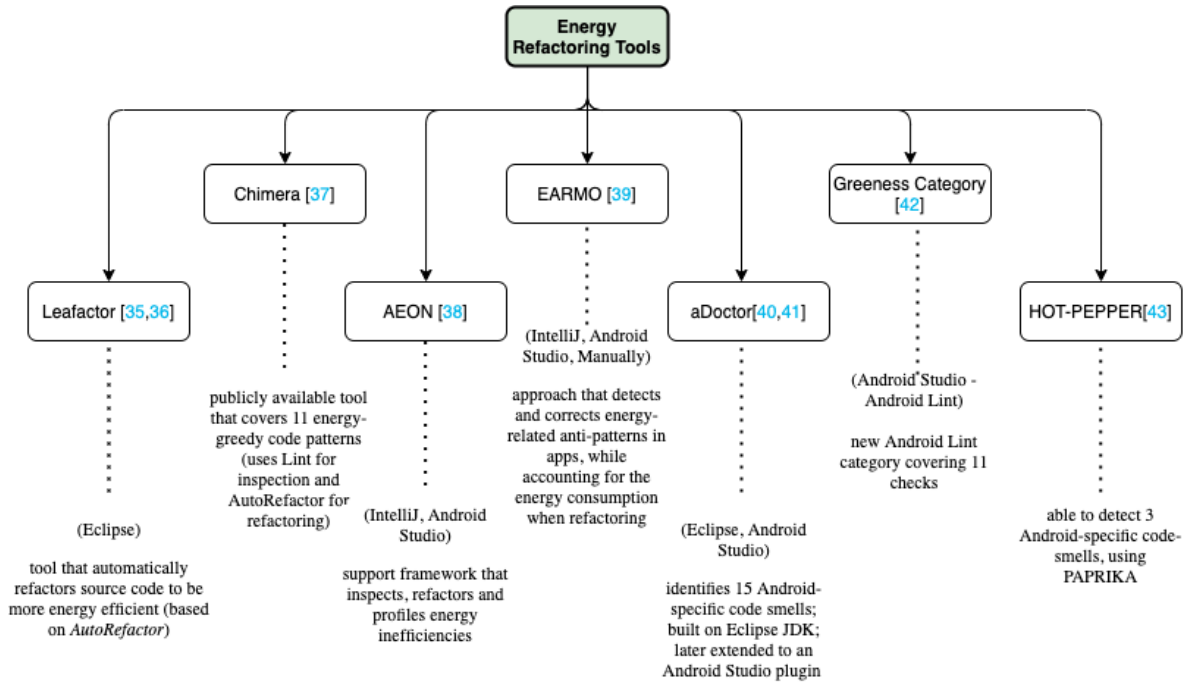

Figure 2.1: Energy-related refactoring tools.

[16]Android Application Package is the package file format used by the Android operating system.

3

EcoAndroid: An Android Studio Plugin

Contents

This chapter presents EcoAndroid, the Android Studio plugin that we developed. Section 3.1 lists functional and non-functional requirements, and Section 3.2 describes the architecture and the technology choices made. Section 3.3 lists every energy pattern supported by EcoAndroid, showing an example for each one.

3.1 Requirements

As described and justified in the previous chapter, our goal is to develop an Android Studio plugin that can automatically refactor Java Android mobile applications so that they follow certain energy patterns. This section presents requirements that the plugin must fulfill.

3.1.1 Functional Requirements

Process language elements Since the goal is to suggest improvements to the user code, we first need a way to extract any element of the code that is relevant for the analysis, such as method calls and method implementations. An IDE that provides the creation of plugins will usually provide some sort of model to access and manipulate the elements in the code, which is the case of Android Studio (the IDE selected for this project).

Find element scope For almost every pattern supported, it is necessary to know the scope where the element is situated in (e.g., knowing how the assignments to variables used to invoke a parameter are being done). Android Studio not only provides a way to access and manipulate the elements, but also a form of seeing the relationships between the elements through abstract trees.

Create new elements The objective of the project is to perform automated refactoring, making it necessary to alter, create or even delete elements in the source code. The IDE selected has a feature allowing us to perform every change necessary.

Display warnings and messages to the user As a result of the inspection process, warnings and messages should be displayed to the user in the exact place where the problem lays. Android Studio IDE displays warnings in the source code as yellow highlights in the section of the problem.

Put the user in control The user should have the option to apply the proposed refactorings. If the problem appears as a warning in the source code, the user can chose whether to apply the alterations or not.

Allow the user to reverse refactorings The developer could wish to reverse the refactoring for some reason. The environment chosen gives the user the option by using CTRL+Z or CMD+Z, depending on the operating system.

3.1.2 Non-Functional Requirements

Integrate with an IDE The tool being integrated with a popular IDE has the potential to reach an higher number of users. It also makes it easier for developers to use the tool, since it is part of an IDE they are already using.

Extendable It should be possible to extend the tool so that it supports more patterns in the future. This is important, since the community is still exploring new ways of writing energy-efficient mobile applications.

Execute the tool in batch mode Even though the envisaged tool is interactive and integrated into the IDE, it is desirable to run the tool in batch mode so that it can easily be executed on a large number of projects. This is particularly useful to evaluate new features.

Give information to the user when the refactoring is too complex In some cases, the alterations needed to fix the code smell might be too complex for a refactoring tool to apply. One solution is to give information to the user, in the form of comments, so that users can perform the changes themselves if they wish to do so. The insertion of comments is possible in Android Studio, making this feature possible.

Display all the warnings in a set of files Android Studio allows the user to inspect, one inspection or a group of them, more than one file at the time. This makes it possible, for example, for the user to inspect their whole project at once.

3.2 Architecture

EcoAndroid is a publicly-available Android Studio plugin[1] 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, making EcoAndroid also compatible with IntelliJ. Android Studio is the official IDE for Android app development, making it the best choice for 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 implement the refactorings of the source code, the *Program Stucture*

[1] Available in the JetBrains store: https://plugins.jetbrains.com/plugin/15637-ecoandroid.

Interface (PSI) [45] of IntelliJ is 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. The most common way to inspect code using PSI is with a top-down navigation approach using a *visitor*, being the method used by EcoAndroid. Since IntelliJ is an IDE for Java, and being PSI an layer of it, one of the languages supported in this layer is the Java coding language. However, through the *Custom Language Support* feature of the IDE, it is possible to extend the PSI to another language.

EcoAndroid is implemented as extending IntelliJ's functionality: the functionality is added as an `<extension/>` element in the plugin file `plugin.xml`.[2] Our functionality is implemented through inspections where each one represents 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 suggests, 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 in 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 associated with the source code currently being viewed; or a list of results of the IDE's inspection task. 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 run. The plugin supports a total of 5 energy patterns, with a total of 10 cases. Each case is implemented as one local inspection in the plugin.

Plugin Structure Figure 3.1 shows EcoAndroid's structure. It is composed of 10 inspections, each representing one case supported by EcoAndroid. Each case has always two classes associated with it: an inspection class and a quick-fix class. The inspection class is responsible for detecting the code smell while the quick-fix class is responsible for fixing the code smell.

[2] `Plugin.xml` is the plugin configuration file which has information about the actions and inspections done by the plugin.

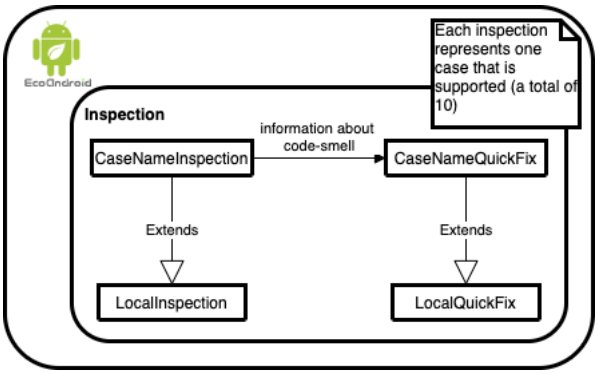

Figure 3.1: EcoAndroid structure.

User interaction process Figure 3.2 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 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)). A developer that wishes to use the plugin can run it in two ways: either by opening the file they wish to inspect or by running the inspection on a file, package or project. While in the first option, the warning simply appears in the source code. The second option actually shows a list of the warnings making it easier for the developer to see every problem. It is also possible for the developer to apply every refactoring automatically (*Analyze* | *CodeCleanup*), although this approach was not taken when evaluating EcoAndroid.

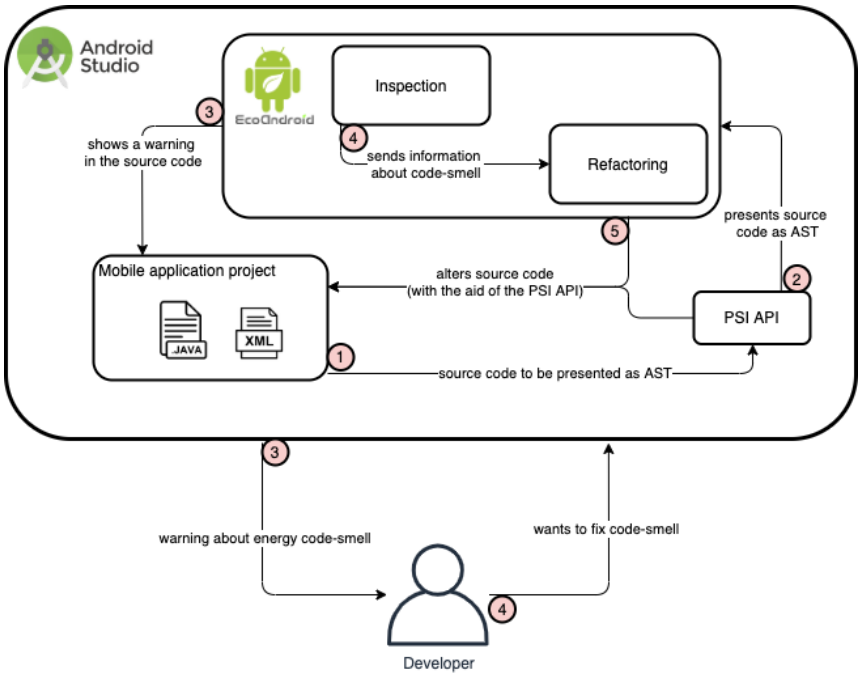

Figure 3.2: EcoAndroid detection and refactoring process.

Methodology for supporting a new inspection. At the time of writing, EcoAndroid supports 10 cases. However, the plugins can be easily extended. The addition of new case, more precisely a local inspection, follows these steps:

Step 1. Include the inspection in the `plugin.xml` file by adding a new `<localInspection/>` element. (see Listing 3);

Step 2. Create the inspection file, a subclass of `LocalInspectionTool`.[3] This class must be the one registered in the previous step. (see Listing 4);

Step 3. Create the quick-fix file which will perform the refactoring, a subclass of `LocalQuickFix`.[4] This is the class responsible for refactoring the source code, previously linked to the warning on the inspection class. (see Listing 5).

Listings 3, 4 and 5 exemplify the addition of a refactoring — in this case, the refactoring associated with the pattern *Dynamic Wait Time*, one of the cases supported by EcoAndroid. Listing 3 shows how an inspection is added to the plugin configuration file, as an extension element. Listings 4 and 5 show an example of how the inspection class, previously linked to the refactoring, sends the code smell information to a class which deals with handling the refactoring process.

```
<idea-plugin>
  ...
  <extensions defaultExtensionNs="com.intellij">
    <localInspection
            language="JAVA"
            displayName="EcoAndroid: Dynamic Retry Delay Energy Pattern -
                switching to a dynamic wait time between resource attempts"
            groupPath="Java"
            groupBundle="messages.InspectionsBundle"
            groupKey="group.names.probable.bugs"
            enabledByDefault="true"
            level="WARNING"
            implementationClass="DynamicRetryDelay.DynamicWaitTime.DynamicWaitTimeInspection"/>
    ...
  </extensions>
  ...
</idea-plugin>
```

Listing 3: Supporting an inspection example - `plugin.xml`.

```
public class DynamicWaitTimeInspection extends LocalInspectionTool {
    private DynamicWaitTimeQuickFix dynamicWaitTimeQuickFix;

    public PsiElementVisitor buildVisitor(@NotNull ProblemsHolder holder,
    boolean isOnTheFly) {
        return new JavaElementVisitor() {
        ...
        holder.registerProblem(timeVariable, DESCRIPTION_TEMPLATE_DYNAMIC_WAIT_TIME, dynamicWaitTimeQuickFix);
        ...
}
```

Listing 4: Supporting an inspection example - local inspection sub class.

[3]Fully qualified class name: `com.intellij.codeInspection.LocalInspectionTool`.
[4]Fully qualified class name: `com.intellij.codeInspection.LocalQuickFix`.

```
public class DynamicWaitTimeQuickFix implements LocalQuickFix { ... }
```

Listing 5: Supporting an inspection example - local quick-fix sub class.

3.3 Implementation

EcoAndroid supports a total of 5 energy patterns, divided in 10 separate cases, as represented in Figure 3.3. The energy patterns supported are a subset of the ones presented in the catalog by Cruz et al. [5], which follow the criteria presented in Section 2.2. Out of the energy patterns that satisfy those criteria, a set of 5 was chosen, taking into consideration how easy it is to support them. The energy patterns are: *Dynamic Retry Delay*, *Push Over Poll*, *Reduce Size*, *Cache*, and *Avoid Extraneous Graphics and Animations*. For some of these patterns, more than one case was implemented, as Figure 3.3 shows. The cases with a gray background represent informational cases, which are refactorings that do not alter any source code, opting to only inform the developer of the change by adding a `//TODO` comment. This type of warning 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.

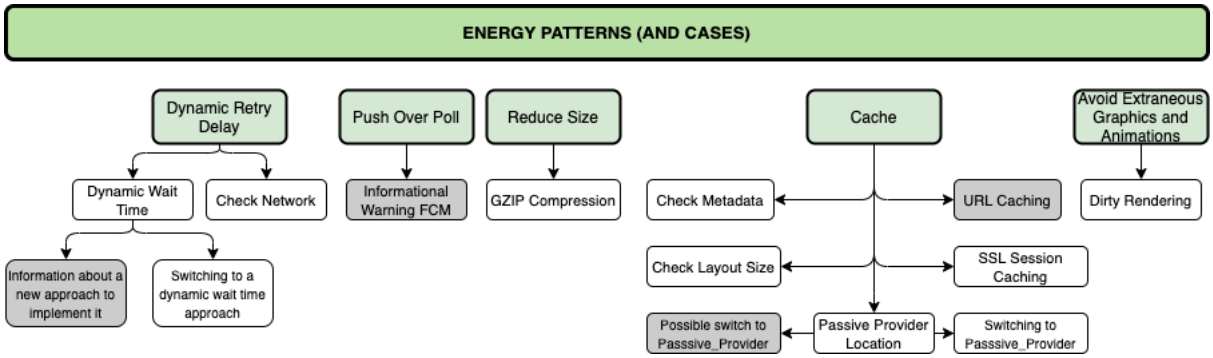

Figure 3.3: Energy patterns supported by EcoAndroid. Cases with gray background represent informational cases, where no source code is altered.

Warnings and comments. The messages displayed in EcoAndroid's warnings follow a systematic template, which is presented in the following box. The *description template* is the text shown to user about the description of the problem found. The *apply fix popup* is the description of the action performed by the quick-fix class — i.e., by the refactoring process.

```
Description Template : EcoAndroid: *ENERGY PATTERN NAME* [*ENERGY PATTERN CASE NAME*] can be applied
Apply Fix popup : Apply pattern *ENERGY PATTERN NAME* [*ENERGY PATTERN CASE NAME*]
```

Figure 3.4 shows an example of the messages displayed by the plugin. The case represented is *Dynamic Wait Time*.

Figure 3.4: EcoAndroid warning information example.

Even though it will be omitted in most of the following code listings, every refactoring performed by the plugin inserts a comment summarizing the change and listing the files affected before the method where the energy problem is detected (see the box below). Comments will be shown when describing Information Warnings, e.g. the one regarding the Dynamic Wait Time case and the Push Over Poll case.

```
/*
* EcoAndroid: *ENERGY PATTERN NAME* ENERGY PATTERN APPLIED
* The goal is to *ENERGY PATTERN DESCRIPTION*
* Application changed java file *JAVA FILE* [and xml file *XML FILE*]
*/
```

Figure 3.5 shows an example of the comment added by EcoAndroid to explain the alterations made. The case exemplified is *Dynamic Wait Time*.

```
/*
 * EcoAndroid: DYNAMIC RETRY DELAY ENERGY PATTERN APPLIED
 * Switching the wait time from constant to dynamic
 * Application changed java file "DynamicWaitTime.java"
 */
```

Figure 3.5: EcoAndroid comment example.

We present the 5 energy patterns in the next 5 subsections. For each pattern, we list the considered cases, and for each case, we describe the case, the implemented inspection, and the implemented refactoring.

3.3.1 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 catalog has two Java examples of the application of this pattern. However, only one case was supported because the second one implied too many alterations to the source code. A Kotlin example from the catalog was also translated to Java, as it was the example given in the project document.

Dynamic Retry Delay cases

- Dynamic Wait Time
- Check Network

3.3.1.1 Dynamic Wait Time

We call the first case of the Dynamic Retry Delay energy pattern *Dynamic Wait Time*. The idea is to grow the interval between thread sleeps exponentially, instead of keeping it constant (thus decreasing the chance of trying to access a resource that most likely went down). The example presented in Listing 6 was retrieved from the mobile application Simple Task[5] and translated from Kotlin.

```
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 6: Dynamic Wait Time - code smell detected.

[5]Commit: https://github.com/mpcjanssen/simpletask-android/commit/1b674be880439bb176c285509efbd96629270f70.

Dynamic Wait Time inspection

Step 1. Look for `Thread#sleep`[6] method calls;

Step 2. Verify if a variable is being used to invoke the method;

Step 3. Determine the origin of the variable;

`Parameter`: check every method call to the method the parameter belongs to and verify how the assignments were made. If they are being done statically, continue the inspection;

`Variable`: either a local or a global variable. If the assignments to the variable are being done statically, continue the inspection.

Step 4. If an informational warning is not already present in the code, two warnings are presented to the developer: an informational warning (Listing 7) and a warning whose refactoring alters the source code (Listing 8).

In the example shown in Listing 6, 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. The user then has the choice to apply two different solutions: the first is based on the android.work package (and EcoAndroid only introduces information as a comment); the second refactors the code so that a dynamic wait time is used.

(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 7: 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 com-

[6]Fully qualified class name: `java.lang.Thread`.

ment with a link to further explain how to use the `WorkRequest`[7] 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 8: Dynamic Wait Time - energy pattern applied (switching to a dynamic wait time between resource attempts case).

Dynamic Wait Time refactoring (switching to a dynamic wait time between resource attempts case)

Step 1. Create the `accessAttempts` variable;

Step 2. Alter every dynamic assignment of the variable used to put the thread to sleep to an incremental assignment of the `accessAttempts` variable;

Step 3. Compute a value to put the thread to sleep.

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 named `accessAttempts`, initialized with 0. As the name suggests, the variable holds the number of access attempts to a resource. Afterwards, every `newBackOffSeconds` assignment to a constant value is altered to an incremental assignment of the `accessAttempts`. At last, the number of access attempts is altered to a value of time with an upper bound. Listing 8 represents the application of the *Dynamic Wait Time* pattern, which applies the alterations described.

3.3.1.2 Check Network

The second case implemented is called *Check Network*. This refactoring is retrieved from the mobile application Episodes.[8] The method to apply this energy pattern was deprecated so the refactoring was

[7] Fully qualified class name: `androidx.work.WorkRequest`.
[8] Commit: https://github.com/jamienicol/episodes/commit/e004294d1b2a4d05c49157f1ddec32c0071345a5.

adapted to work with the last version of the Android library. The *Check Network* case's idea is to avoid processing of a request without an active network connection. Whenever there is a request to process, the application should verify if there is network connection before processing it. This prevents the processing from being stuck on unresponsive internet method calls.

```
public static class Service extends IntentService {
        ...
       @Override
        protected void onHandleIntent(Intent intent) { ... }
        ...
}
```

Listing 9: Check Network - code smell detected.

Check Network inspection

Step 1. Search for implementations overriding the method `BroadcastReceiver#onHandleIntent`[9]. The `onHandleIntent` method is invoked on the worker thread with a request to process;

Step 2. Verify if the body of the method, or any method it invokes, `onHandleIntent` does not invoke the method `getActiveNetwork` stating an intention to verify if there is an active network;

Step 3. Check if the `AndroidManifest.xml`[10] of the mobile application has permission to use internet.

Exceptions: If the `getActiveNetwork`'s method call is located deeper into the method calls, it will show a warning to the developer.

```
public static class Service extends IntentService {
        ...
        @Override
        protected void onHandleIntent(Intent intent) {
            if (hasActiveNetwork()){...
            else {
                NetworkStateReceiver networkStateReceiver = new NetworkStateReceiver();
                ConnectivityManager connectivityManager = (ConnectivityManager)
                   getApplicationContext().getSystemService(Context.CONNECTIVITY_SERVICE);
                networkStateReceiver.enable(getApplicationContext());
                networkStateReceiver.setService(this);
                connectivityManager.registerDefaultNetworkCallback(networkStateReceiver);
            }
        }

        protected boolean hasActiveNetwork() {
            final ConnectivityManager connManager = (ConnectivityManager)
                getApplicationContext().getSystemService(Context.CONNECTIVITY_SERVICE);
            Network activeNetwork = connManager.getActiveNetwork();
            return (activeNetwork != null);
        }
        ...
}
```

Listing 10: Check Network - energy pattern applied (`onHandleIntent` and `hasActiveNetwork` methods).

[9] Fully qualified class name: `android.content.BroadcastReceiver`.

[10] `AndroidManifest.xml` describes essential information about your app to the Android build tools, the Android operating system, and Google Play.

```
public static class Service extends IntentService {
        ...
        public class NetworkStateReceiver extends ConnectivityManager.NetworkCallback {
            private Service service;
            public void setService(Service newService) { service = newService; }

            @Override
            public void onAvailable(Network network) { ... }

            // EcoAndroid: Method to  "turn on" this class
            public void enable(Context context) {
                ConnectivityManager connectivityManager = (ConnectivityManager)
                   context.getSystemService(Context.CONNECTIVITY_SERVICE;
                connectivityManager.registerDefaultNetworkCallback(this);
            }

            // EcoAndroid: Method to  "turn off" this class
            public void disable(Context context) {
                ConnectivityManager connectivityManager = (ConnectivityManager)
                   context.getSystemService(Context.CONNECTIVITY_SERVICE;
                connectivityManager.unregisterNetworkCallback(this);
            }
        }
        ...
}
```

Listing 11: Check Network - energy pattern applied (`NetworkStateReceiver` class).

```
public void onAvailable(Network network) {
    // EcoAndroid: If there is an active network connection, this method will
    // "turn off" this class and arrange to process the request
    if (service.hasActiveNetwork()) {
        Context context = getApplicationContext();
        disable(context);
        final AlarmManager alarmManager = (AlarmManager)  context.getSystemService(Context.ALARM_SERVICE);
        final Intent innerIntent = new Intent(context, Service.class);
        final PendingIntent pendingIntent = PendingIntent.getService(context, 0, innerIntent, 0);
        SharedPreferences preferences = context.getSharedPreferences(context.getPackageName(), Context.MODE_PRIVATE);
        boolean autoRefreshEnabled = preferences.getBoolean("pref_auto_refresh_enabled", false);
        final String hours = preferences.getString("pref_auto_refresh_period", "0");
        long hoursLong = Long.parseLong(hours) * 60 * 60 * 1000;
        if (autoRefreshEnabled && hoursLong != 0) {
            final long alarmTime = preferences.getLong("last_auto_refresh_time", 0) + hoursLong;
            alarmManager.set(AlarmManager.RTC, alarmTime, pendingIntent);
        } else {
            alarmManager.cancel(pendingIntent);
        }
```

Listing 12: Check Network - energy pattern applied (`onAvailable` method).

Check Network refactoring

Step 1. Create a method to check if there is an active network connection;

Step 2. Split the `onHandleIntent`'s code path between having a network connection or not;

Step 3. In the case of not having an active network connection, create a class to deal with it and wait for a connection;

Step 4. Create `NetworkStateReceiver` class with three methods:

`onAvailable`: schedule the processing of the request and invoke the `disable` method;

`enable`: register the class in the connectivity manager;

`disable`: unregister the class in the connectivity manager.

The first two alterations done to the source code are the creation of the a method that checks if there is an active network connection — `hasActiveNetwork()` — and the creation of an if statement in the `onHandleIntent`'s method body that distinguishes the code path whether there is an connection or not — `if (hasActiveNetwork())`. The second alteration done to code is the definition of the class `NetworkStateReceiver`. This class is of subtype `NetworkCallback`[11] and implements three methods: `onAvailable`, `enable` and `disable`. The `onAvailable` method, whose code is presented in Listing 12, is called when a framework connects and a new network is ready to be used. The `enable` method starts up the class, registering the callback in the connectivity manager. The `disable` method shuts down the class, unregistering the callback in the connectivity manager. Both of these methods implementations are presented in Listing 11.

3.3.2 Push Over Poll

A *push notification* establishes and maintains a connection with a server over the Internet and 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 [46], 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 is not always actively querying resources. If there are frequent notifications coming in, the difference between these two mechanisms is not as significant. The catalog did not have any example of this energy pattern. In order to support this pattern, a common way of doing polling, described in the Android documentation, is flagged as a code smell.

Push Over Poll cases

- Informational Warning FCM

3.3.2.1 Informational Warning FCM

According to the Android documentation [47], *Firebase Cloud Messaging* (FCM) is the mechanism to use when choosing push notifications. Since FCM requires developers to register their applications, EcoAndroid only creates an informational warning instead of performing code changes. When using a polling mechanism, Android documentation[12] states that there are three common ways to define background work:

[11] Fully qualified class name: `android.net.ConnectivyManager.NetworkCallback`.
[12] Guide to background processing: https://developer.android.com/guide/background.

- Registering a broadcast receiver in the manifest file;
- Scheduling a repeating alarm using `AlarmManager`[13];
- Scheduling a background task using either `WorkManager`[14] or `JobScheduler`[15].

Currently, the plugin detects the second approach as a polling mechanism. This case deals with objects of the `AlarmManager` setting repeating alarms, as seen in Listing 13.

```
public void setAlarm(Context context) {
    AlarmManager am =( AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
    Intent i = new Intent(context, Alarm.class);
    PendingIntent pi = PendingIntent.getBroadcast(context, 0, i, 0);
    am.setRepeating(AlarmManager.RTC_WAKEUP, System.currentTimeMillis(), 1000 * 60 * 10, pi);
}
```

Listing 13: Informational Warning FCM - code smell detected.

Informational Warning FCM inspection

Step 1. Search for invocations of the method `AlarmManager#setRepeating`;

Step 2. Verify if an EcoAndroid informational comment is not already before the method.

```
/*
 * TODO EcoAndroid
 * PUSH OVER POLL ENERGY PATTERN INFO WARNING
 * An alternative to a polling service is a to use push notifications
 * One way to implement them is to use Firebase Cloud Messaging
 * FCM uses an API and works with Android Studio 1.4 or higher with Gradle projects
 * If you wish to know more about this topic, read the following information:
 * https://firebase.google.com/docs/cloud-messaging/android/client
 */
public void setAlarm(Context context) { ... }
```

Listing 14: Informational Warning FCM - energy pattern applied.

Informational Warning FCM refactoring

Step. Insert EcoAndroid informational comment.

3.3.3 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. The catalog did not have any Java examples that we could use, so the example described in the *Example* section of the *Reduce Size* energy pattern was implemented.

[13]Fully qualified class name: `android.app.AlarmManager`.
[14]Fully qualified class name: `androidx.work.WorkManager`.
[15]Fully qualified class name: `android.app.job.JobScheduler`.

Reduce Size cases

- GZIP Compression

3.3.3.1 GZIP Compression

In the case of *GZIP Compression* of the *Reduce Size*'s energy pattern, the intention is to request that any given response from an URL Connection is compressed by the GZIP scheme.

```
HttpURLConnection con = (HttpURLConnection) url.openConnection();
System.out.println("Length : " + con.getContentLength());
Reader reader = new InputStreamReader(con.getInputStream());
```

Listing 15: GZIP Compression - code smell detected.

GZIP Compression inspection

Step 1. Search for an opened `HttpUrlConnection`[16];

Step 2. Determine whether the connection is receiving a stream;

Step 3. Verify if the input stream is being requested as compressed by the GZIP scheme. If it isn't, a warning is presented by the plugin.

Exceptions: if the invocation of the method `openConnection)` happens inside the declaration of a resource in a *try-with-resources* statement, the plugin will not show a warning. This is because the alterations for this pattern imply an if statement, which can not be added to the resource declaration portion of the *try-with-resources* statement.

As we can see from the example shown in Listing 15, there is an open connection receiving an input stream that is not requested to be compressed. When going through all the steps necessary by the inspection, EcoAndroid shows a warning in the source code.

```
HttpURLConnection con = (HttpURLConnection) url.openConnection();
con.setRequestProperty("Accept-Encoding", "gzip");
System.out.println("Length : " + con.getContentLength());
Reader reader;
if ("gzip".equals(con.getContentEncoding())) {
    reader = new InputStreamReader(new GZIPInputStream(con.getInputStream()));
} else {
    reader = new InputStreamReader(con.getInputStream());
}
```

Listing 16: GZIP Compression - energy pattern applied.

GZIP Compression refactoring

Step 1. Invoke `URLConnection#setRequestProperty` method to accept stream compressed by the GZIP scheme;

[16]Fully qualified class name: `java.net.URLConnection`.

Step 2. Create an if statement to correctly receive the input stream, whether it comes compressed or not.

3.3.4 Cache

The goal of the *Cache* pattern is to store data that is being used frequently.This leads to a lower energy consumption since it reduces the amount of code executed and a lower number of accesses to retrieve information. The catalog has six Java implementations of the Cache energy pattern. Four of these are supported by the EcoAndroid while two are not. One is due to altering involving SQL files, which is not supported by the PSI API, and the other one is due to changing more than one file — the case created an variable to store the last value of a variable (similar to Check Metadata), but over two class files.

Cache cases

- Check Metadata
- Check Layout Size
- SSL Session Caching
- Passive Provider Location
- URL Caching

3.3.4.1 Check Metadata

In the specific subcase of *Check Metadata*, the intention is to check the data received by the method `BroadcastReceiver#onReceive`. On the mehod `onReceive`, the idea is to verify the data retrieved from the parameter `intent` and, if nothing changed since the last time the method was performed, no code is executed. If something changes, then the required actions can be taken. The transformation *Check Metadata* is based on a commit to the mobile application GadgetBridge[17].

```
public class MusicPlaybackReceiver extends BroadcastReceiver { ...
    @Override
    public void onReceive (Context context, Intent intent) {
        String artist = intent.getStringExtra("artist");
        String album = intent.getStringExtra("album");
        String track = intent.getStringExtra("track");
        MusicSpec musicSpec = new MusicSpec();
        LOG.info("Current track: " + artist + ", " + album + ", " + track);
        musicSpec.artist = artist;
        musicSpec.album = album;
        musicSpec.track = track;
    }
}
```

Listing 17: Check Metadata - code smell detected.

[17] Commit: https://github.com/Freeyourgadget/Gadgetbridge/commit/d1a62968f69ebb7dd6dd7cb273a06aa0761681a4.

Check Metadata inspection

Step 1. Search for implementations that override the `BroadcastReceiver#onReceive`[18] method;

Step 2. Determine if the method retrieves information from the parameter `intent` and saves it in a local variable;

Step 3. Verify if the variables are being checked before executing the method. If not, a warning is shown.

Exceptions: there exist a few scenarios where this pattern should not be applied. For example, when the `onReceive` method invokes methods from either the class `NotificationManager`[19] or `DownloadManager`[20].

In Listing 17, the implementation of the method `onReceive` is saving information from the parameter `intent` in three local variables: `artist`, `album` and `track`. Since it is not verifying any of these variables before executing the method, the plugin detects a problem and shows a warning.

```
public class MusicPlaybackReceiver extends BroadcastReceiver {
    private String lasttrack = null;
    private String lastalbum = null;
    private String lastartist = null;
    ...
    @Override
    public void onReceive(Context context, Intent intent) {
        if (lastartist.equals(intent.getStringExtra("artist")) && lastalbum.equals(intent.getStringExtra("album")) &&
        lasttrack.equals(intent.getStringExtra("track"))) {
            // EcoAndroid: nothing has changed; we can safely return
            return;
        }
        updateValues(intent);
        MusicSpec musicSpec = new MusicSpec();
        LOG.info("Current track: " + lastartist + ", " + lastalbum + ", " + lasttrack);
        musicSpec.artist = lastartist;
        musicSpec.album = lastalbum;
        musicSpec.track = lasttrack;
    }
    private void updateValues(Intent intent) {
        lastartist = intent.getStringExtra("artist");
        lastalbum = intent.getStringExtra("album");
        lasttrack = intent.getStringExtra("track");
    }
}
```

Listing 18: Check Metadata - energy pattern applied.

Check Metadata refactoring

Step 1. Create variables to store the previous values of the data retrieved from the parameter `intent`;

Step 2. Add an if statement to verify if there were any alterations since the last execution of the method;

Step 3. Create auxiliary method to update storing values, `updateValues(Intent)`.

Step 4. Alter every reference to old variable to new variable, (e.g. `track`→`lasttrack`).

[18] Fully qualified class name: `android.content.BroadcastReceiver`.
[19] Fully qualified class name: `android.app.NotificationManager`.
[20] Fully qualified class name: `android.app.DownloadManager`.

3.3.4.2 Check Layout Size

The *Check Layout Size* case aims to prevent execution of code that resets the size of a view, when the view's measures are at 0. This case, present in the catalog, was retrieved from the Shattered-Pixel-Dungeon mobile application[21].

```
private void updateDisplaySize() {
    float dispWidth = view.getMeasuredWidth();
    float dispHeight = view.getMeasuredHeight();
    float dispRatio = dispWidth / (float) dispHeight;
    float renderWidth = dispRatio > 1 ? PixelScene.MIN_WIDTH_L : PixelScene.MIN_WIDTH_P;
    float renderHeight = dispRatio > 1 ? PixelScene.MIN_HEIGHT_L : PixelScene.MIN_HEIGHT_P;
    if (powerSaver()){
        int maxZoom = (int) Math.min(dispWidth/renderWidth, dispHeight/renderHeight);
        renderWidth *= Math.max(2, Math.round(1f + maxZoom * 0.4f));
        renderHeight *= Math.max(2, Math.round(1f + maxZoom * 0.4f));
        final int finalW = Math.round(renderWidth);
        final int finalH = Math.round(renderHeight);
        if (finalW != width  finalH != height){
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    view.getHolder().setFixedSize(finalW, finalH);
                }
        ...
}
```

Listing 19: Check Layout Size - code smell detected.

Check Layout Size inspection

Step 1. Search method calls to `View#getMeasuredWidth`[22] or `View#getMeasuredHeight`;

Step 2. Check if the values returned from the method calls are being used to invoke the method `SurfaceHolder#setFixedSize`;

Step 3. Verify if the values from the first invocation are not verified to being zero along the method's body.

In Listing 19, we show an invocation to the method `getMeasuredWidth` and `getMeasuredHeight`, each saved in a local variable, respectively. Those local variables are then used to invoke the method `setFixedSize` through other local variables (`dispthWidth`→`renderWidth`→`finalW` and `dispthHeight`→`renderHeight`→`finalH`).

```
private void updateDisplaySize() {
    if (view.getMeasuredWidth() == 0  view.getMeasuredHeight() == 0) {
        return;
    }
    ...
}
```

Listing 20: Check Layout Size - energy pattern applied.

[21] Commit: https://github.com/00-Evan/shattered-pixel-dungeon/commit/5491f315f6080f6ce9883638c5a7339c19a14828.
[22] Fully qualified class name: `android.view.View`.

Check Layout Size refactoring

Step. Create an if statement before the `getMeasuredWidth` or `getMeasuredHeight` method calls that checks whether if one of the view's measures are zero.

3.3.4.3 SSL Session Caching

The idea behind the *SSL Session Caching* case is to increase the cache size to a value as large as possible. Li et al. [16] performed a study about the energy consumption of increasing the cache size. They concluded that even if allocating more memory means an higher energy consumption, the difference is not as big and could even be worth if it means making fewer accesses to the network. In order to achieve this, every time the inspection finds an initialization of an SSLContext, it verifies if the size of the cache size is not already set to 0 (which means the cache size is as big as it can be). If it does not, it adds the call to set the cache size to 0.

```
SSLContext context = SSLContext.getInstance("TLS");
context.init(keyManagers, trustManagers, null);
```

Listing 21: SSL Session Caching - code smell detected.

SSL Session Caching inspection

Step 1. Find an initialization of an `SSLContext`[23];

Step 2. Verify if the context being initialized is not already altering the cache size to 0.

```
SSLContext context = SSLContext.getInstance("TLS");
context.init(keyManagers, trustManagers, null);
SSLSessionContext sslSessionContext = context.getServerSessionContext();
int sessionCacheSize = sslSessionContext.getSessionCacheSize();
if (sessionCacheSize > 0) {
    // EcoAndroid: the next line makes the cache size of an SSL Session unlimited
    sslSessionContext.setSessionCacheSize(0);
}
```

Listing 22: SSL Session Caching - energy pattern applied.

SSL Session Caching refactoring

Step 1. Create a local variable called `sslSessionContext`, which holds the server session context;

Step 2. Create a local variable named `sessionCacheSize`, which has the size of the cache;

Step 3. Insert an if statement to make the cache's size unlimited, if it is not already.

[23] Fully qualified class name: `javax.net.SSLContext`.

3.3.4.4 Passive Provider Location

The `LocationManager`[24] class job is to provide access to the system location services. It has three different type of providers: `GPS_PROVIDER`, `NETWORK_PROVIDER` and `PASSIVE_PROVIDER`. This last one can be used to passively receive location updates when other applications or services request them without actually requesting the locations yourself. This provider will only return locations generated by other provider, leading to a lower energy consumption of the mobile application. This case is called *Passive Provider Location* and it is based on the transformation found in the mobile application K9[25]. The transformation changes the parameter `provider` in `requestLocationUpdates` method calls of the `LocationManager` class. The *Passive Provider Location* presents the developer with two solutions, depending on certain specifications:

(1) EcoAndroid: Cache - possible switch to `PASSIVE_PROVIDER`

```
private void setLocationManager() { ...
    if(locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER)) {
        locationManager.requestLocationUpdates( LocationManager.NETWORK_PROVIDER) , 60000, 50, locationListener);
    }
}
```

Listing 23: Passive Provider Location - code smell detected (possible switch to $PASSIVE\_PROVIDER$).

(2) EcoAndroid: Cache - switching to `PASSIVE_PROVIDER`

```
locationManager.requestLocationUpdates( LocationManager.NETWORK_PROVIDER) , 60000, 50, locationListener);
```

Listing 24: Passive Provider Location - code smell detected (switching to $PASSIVE\_PROVIDER$).

Passive Provider Location inspection

Step 1. Search for `LocationManager#requestLocationUpdates`;

Step 2. Verify the first parameter given:

`PASSIVE_PROVIDER`: this provider is the one that consumes less energy out of the three. If the method is being invoked with it, the inspection stops here;

`anything else`: if the method is being invoked with any of these, the inspection continues.

Step 3. Verify if the `requestLocationUpdate`'s invocation is done inside the condition of an if statement. If not, the inspection stops here;

Step 4. Inspect if, before this method call, exists an explicit intention for a desired provider;

`No explicit intention`: a warning, whose solution alters the source code, can be shown to the developer (Listing 23);

[24] Fully qualified class name: `android.location.LocationManager`.
[25] Commit: https://github.com/k9mail/k-9/commit/43c38a047feedda4720af5bfbc188a33f8dfaced.

`Explicit intention`: examine if there is not already an EcoAndroid informational comment before the method. An explicit intention is said to be found if there is an invocation to `LocationManager#isProviderEnabled` prior to the `LocationManager#requestLocationUpdates` invocation. If not, an informational warning can be shown. (Listing 24).

```
/*
 * TODO EcoAndroid
 * CACHE ENERGY PATTERN INFO WARNING
 * Another type of provider for LocationManager is PASSIVE_PROVIDER
 * This provider uses a cache mechanism to retrieve the location,
which consumes less energy then the other options
 * This approach uses the android.location package
 * If you wish to know more about this topic, read the following information:
 * https://developer.android.com/reference/android/location
/LocationManager#PASSIVE_PROVIDER
 */
private void setLocationManager() { ... }
```

Listing 25: Passive Provider Location - energy pattern applied (possible switch to $PASSIVE\_PROVIDER$).

```
/*
* EcoAndroid: This next piece of code presents two ways to implement a location
manager that spends less energy.
* 1 - Switching to PassiveProvider
* 2 - Using the criteria class to get the best provider for the needs requested
(with the need for POWER_LOW)
* The second option has been giving "priority". However, the goal is for the
programmer to chose the one which fits bets.
* If you wish to know more, please read:
https://developer.android.com/reference/android/location/LocationManager
*/
boolean flagEcoAndroid = false;
if (flagEcoAndroid) {
    Criteria criteria = new Criteria();
    criteria.setPowerRequirement(Criteria.POWER_LOW);
    String lm = locationManager.getBestProvider(criteria, true);
    locationManager.requestLocationUpdates(lm, 60000, 50, locationListener);
} else {
    locationManager.requestLocationUpdates(LocationManager.PASSIVE_PROVIDER,
    60000, 50, locationListener);
}
```

Listing 26: Passive Provider Location - energy pattern applied (switching to $PASSIVE\_PROVIDER$).

The mobile application from which this case originated from only altered the type of provider when requesting a location update. However, during one of the evaluation phases presented in Section 4.4, we learned about another way to save energy: if the user does not wish to use `PASSIVE_PROVIDER`, the user can set a requirement for `POWER_LOW`, as presented in Listing 26. In the option that alters the source code, the `AndroidManifest.xml` file needs to have permission to `ACCESS_FINE_LOCATION` (Listing 27).

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Listing 27: Passive Provider Location - `AndroidManifest.xml`.

Passive Provider Location refactoring

Step 1. Create a local variable called `flagEcoAndroid`;

Step 2. Create an if statement for both possible solutions given to the user:

`Criter.POWER_LOW`: ask for a provider with a criteria for low power consumption;

`PASSIVE_PROVIDER`: use this provider directly.

Step 3. Insert a comment explaining the alterations and both options given to the user.

3.3.4.5 URL Caching

The idea behind the *URL Caching* case is to prevent processing an URL response that was already processed. The transformation is based on the Android documentation about avoiding redundant downloads.[26]

```
public class HttpConnect{
    ...
    HttpsURLConnection urlCon = (HttpsURLConnection)urlObj.openConnection();
    ...
}
```

Listing 28: URL Caching - code smell detected.

URL Caching inspection

Step 1. Search for an opened `URL`[27] connection;

Step 2. Verify if said connection is retrieving the time the connection was last modified.

Exceptions: the inspection does not flag cases where the URL connection is open on the expression of a return statement.

```
public class HttpConnect{
    static long lastUpdateTime = 0;
    ...
    HttpsURLConnection urlCon = (HttpsURLConnection) urlObj.openConnection();
    long currentTime = System.currentTimeMillis();
    long lastModified = urlCon.getHeaderFieldDate("Last-Modified", currentTime);
    if (lastModified < lastUpdateTime) {
        // TODO EcoAndroid: Skip Update
    } else {
        lastUpdateTime += lastModified;
    }
    ...
}
```

Listing 29: URL Caching - energy pattern applied.

[26] Avoid redundant downloads: https://developer.android.com/training/efficient-downloads/redundant_redundant.

[27] Fully qualified class name: `java.net.URL`.

URL Caching refactoring

Step 1. Create static global variable called `lastUpdateTime`;

Step 2. Create local variable called `currentTime`;

Step 3. Create local variable called `lastModified`;

Step 4. Create an if statement to differentiate the code path for when the url connection has changed since it was last processed from when it was not.

The refactoring starts by getting the current time and the time the url connection object was last modified. Then, a global variable is created in the class entitled `lastUpdateTime`, intialized to 0. The last modification consists in adding an if statement to distinguish both code paths: if the connection was modified since last time or not.

3.3.5 Avoid Extraneous Graphics and Animations

Displaying graphics and animations leads to 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 does not have a direct impact on the user experience). 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 catalog from which the refactorings are based on had one Java example of an implementation of this pattern. However, since the alteration consisted of too many changes to the source code to be eligible for our refactoring tool, the example was not supported. In order to support this energy pattern, a refactoring to alter the rendering mode was created.

Avoid Extraneous Graphics and Animations cases

- Dirty Rendering

3.3.5.1 Dirty Rendering

The `GLSurfaceView` class has two types of rendering modes that can be chosen. One where the render is called repeatedly to the render scene, `RENDERMODE_CONTINUOUSLY` and one where the render is only called when the surface is created and when it is requested, `RENDERMODE_WHEN_DIRTY`. The default render mode is the first one. The *Dirty Rendering* case changes the render mode to `RENDERMODE_WHEN_DIRTY`.

```
mRenderer = new DemoRenderer(this);
mGLSurfaceView.setEGLConfigChooser(8, 8, 8, 8, 16, 0);
mGLSurfaceView.setRenderer(mRenderer);
mGLSurfaceView.setRenderMode( GLSurfaceView.RENDERMODE_CONTINUOUSLY) ;
```

Listing 30: Dirty Rendering - code smell detected.

Dirty Rendering inspection

Step 1. Search for method calls to the method `GLSurfaceView#setRenderMode`[28];

Step 2. Verify the type of rendering. If it is not on `RENDERMODE_WHEN_DIRTY`, a warning is shown.

```
mRenderer = new DemoRenderer(this);
mGLSurfaceView.setEGLConfigChooser(8, 8, 8, 8, 16, 0);
mGLSurfaceView.setRenderer(mRenderer);
mGLSurfaceView.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
```

Listing 31: Dirty Rendering - energy pattern applied.

Dirty Rendering refactoring

Step. Switch to `RENDERMODE_WHEN_DIRTY` the parameter used to invoke the `setRenderMode` method.

[28] Fully qualified class name: `android.opengl.GLSurfaceView`.

4

Evaluation

Contents

This chapter describes the evaluation of EcoAndroid and the results obtained. Section 4.1 gives an overview of the evaluation process and describes its different phases. Section 4.2 details which mobile applications were used for the evaluation and how they were retrieved and selected. Section 4.3 provides quantitative data on the number of energy-efficiency improvement opportunities detected by EcoAndroid. The results can be used to understand whether certain energy patterns are underused. Section 4.4 presents a list of refactorings suggested by EcoAndroid that were applied and integrated into real-world Java mobile applications. Finally, Section 4.5 presents a user study performed to assess the usability of EcoAndroid.

4.1 Overview

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, as illustrated in Figure 4.1:

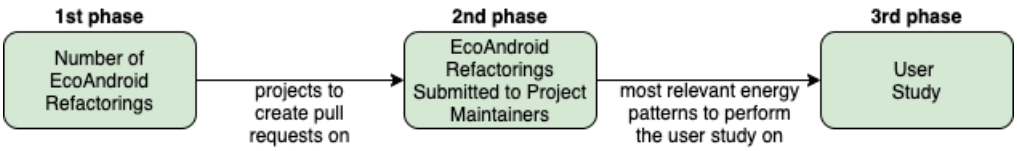

Figure 4.1: Three evaluation phases.

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).

4.2 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* [48], 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[1] 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 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 where the last commit made to the GitHub project was done more than 2 years ago. The mobile applications were then sorted by the following order:

1st Percentage of Pull Requests accepted;
2nd Date of Last Commit;
3rd Total merged Pull Requests;
4th Number of GitHub Stars;
5th 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 2 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. Table 4.1 summarizes the main characteristics of the GitHub projects, considering both processed and inspected projects. Table 4.2 summarizes the number of Java files and lines of code (LoC) processed by EcoAndroid. These numbers allow us to measure the average number of energy-related problems per number of files and LoC. Table A.1 (page 75) lists the top 100 applications considered in this evaluation.

[1] Collection date: 25 June 2020

		GitHub Watchers	GitHub Stars	GitHub Forks	GitHub Contributors	Merged PRs	Closed PRs	% PRs Accepted
All	Min	0	0	0	0	0	0	0
Apps	Mean	18.53	185.37	65.27	10.21	39.71	48.20	0.61
	Max	1692	25630	8716	317	2603	3009	1
Top	Min	0	0	0	1	1	1	1
100	Mean	6.3	38.44	11.61	8.43	5.88	5.88	1
	Max	29	616	181	317	60	60	1

Table 4.1: F-Droid mobile applications characteristics.

	Java Files	Java LoC
Min	13	0
Mean	74.41	14685.98
Max	722	196057
Total	7441	1468597

Table 4.2: Characteristics of the top 100 mobile applications considered.

4.3 First Phase: Number of EcoAndroid Refactorings

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 4.3 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 is no longer a problem in the following stages.

In two instances, feedback from developers pointed to errors of the plugin and these were fixed. On a pull request to the Second Screen application regarding 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 it was no longer a problem in the following stages. Another pull request to the Hacs mobile application regarding the *Check Metadata* energy pattern, the developer answered that the refactoring could break some notifications. It was indeed a bug that was fixed and it was no longer a problem in the following stages.

Energy Patterns	Case	Refactorings	Projects affected
Dynamic Retry Delay	Dynamic Wait Time	0	0
	Check Network	5	4
Push Over Poll	Info Warning FCM	8	3
Reduce Size	GZIP Compression	14	11
Cache	Check Metadata	7	6
	SSL Session Caching	10	7
	Check Layout size	0	0
	Passive Provider Location	11	6
	URL Caching	40	19
Avoid Extraneous Graphics and Animations	Dirty Rendering	0	0
Total		95	35

Table 4.3: Number of energy opportunities detected by EcoAndroid.

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*.

While reviewing the refactorings related to the *Check Metadata* energy pattern for the first two stages, we noticed that for invocations of methods from the the class `NotificationManager` or `DownloadManager`, EcoAndroid was proposing refactorings that affected negatively the notifications shown to the user. There were two occurrences of this problem, one in each of the first two stages of the objective phase. We corrected this problem before moving into stage 3.

Moreover, for refactorings related to the *Passive Provider Location* case, six occurrences represented scenarios where the developer is clearly stating an intention to use a specific provider. In these cases, instead of altering the source code, an informational warning is shown to the user for the possibility of altering provider. Note that since the plugin will show an informational warning to the user, these are counted as refactorings in Table 4.3. The only difference is that these cases will not be used in the second evaluation phase.

Regarding refactorings related to the *GZIP Compression* case, two occurrences were scenarios where the method invocations happened in the resource section of the *try-with-resources* statement. Since the alterations for this pattern imply an if statement, which can not be added to the resource declaration portion, these cases were flagged as false positives.

The combination of patterns with the highest number of associated refactorings is *URL Caching* with

GZIP Compression with 9 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 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*.

4.4 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 to the original GitHub projects for the remaining refactorings. The table in the Appendix B summarizes the pull requests sent and the responses from the developers (page 78). For each project for which EcoAndroid proposed refactorings, we followed the following steps:

Step 1. We forked the project's GitHub repository;

Step 2. We created a new branch named EcoAndroid that contained all the refactorings automatically applied by EcoAndroid;

Step 3. We created a pull request to the original GitHub repository.

All the pull requests sent by us followed a template, which is present in the following box.[2]

```
This improves the energy efficiency of [*APPLICATION NAME*] by applying the [*ENERGY PATTERN
NAME*] Energy Pattern for mobile applications.

The energy pattern was applied in [*JAVA FILE(S) ALTERED NAME(S)*].
The general idea is [*GENERAL DESCRIPTION*].
In particular, [*SPECIFIC ALTERATIONS DESCRIPTION*].
```

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

[2]For a real and full example, see https://github.com/farmerbb/Taskbar/pull/138.

and 5 were rejected. This represents an overall acceptance rate of 80% (or 46.62% when considering all pull requests sent). Out of the 5 pull requests rejected, two rejections were due to the refactoring not saving energy in those cases (one regarding the pattern *SSL Session Caching* and the other *GZIP Compression*), two were because the class did not use internet (this was regarding the pattern *Check Network* and it was later fixed), and one was because the refactoring could break notifications (this was regarding the pattern *Check Metadata* and it was later fixed).

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 less refactoring, altering the number of *Check Network* refactorings to 4. 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.

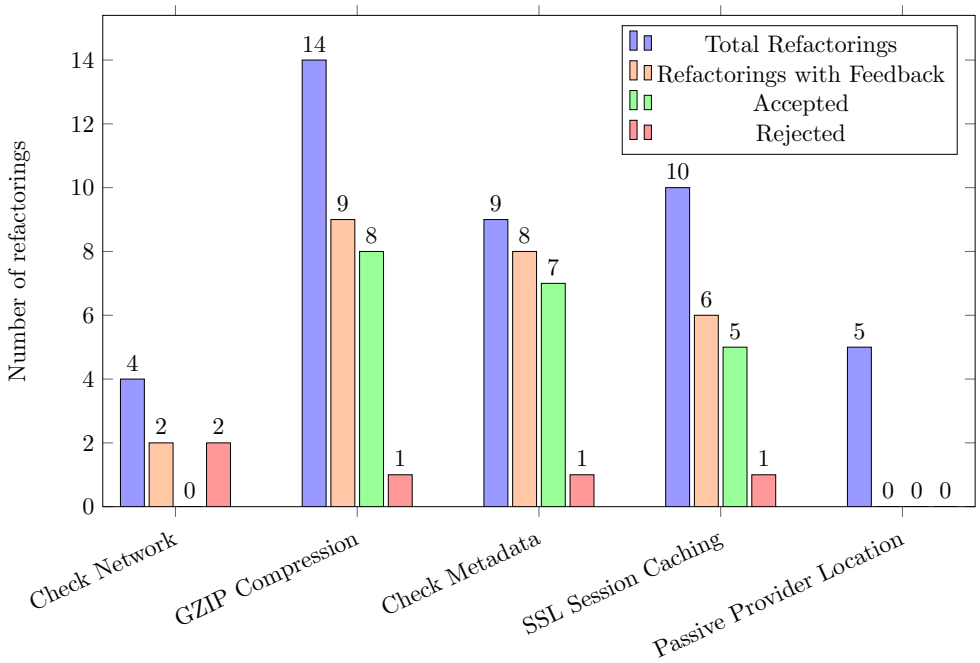

Figure 4.2: Number of refactorings proposed by EcoAndroid for each pattern and statistics on the pull requests sent.

Figure 4.2 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.

Table 4.4 summarizes the main GitHub project characteristics which merged our pull requests. Note that every project component average has a higher value than the average for mobile applications inspected (compare with Table 4.1).

	GitHub Watchers	GitHub Stars	GitHub Forks	GitHub Contributors	Merged PRs	Closed PRs	% PRs Accepted
Min	1	0	1	2	1	1	0.93
Mean	9.15	83	18.46	6.23	18.54	18.85	0.99
Max	22	207	61	12	60	60	1

Table 4.4: Mobile applications characteristics with EcoAndroid merged pull requests.

4.5 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?

4.5.1 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 five energy patterns to examine. 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 four 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.

Table 4.5 details each mobile application used and the number of stars it has.[3]

Energy Pattern	Application Name	Stars
Cache - Check Metadata	Taskbar	207⋆
Reduce Size - GZIP Compression	Tracker-Control-Android	114⋆
Cache - SSL Session Caching	Download Navi	157⋆
Cache - Passive Provider Location	Trekarta	50⋆

Table 4.5: User study mobile applications.

The number of participants was defined considering the work by J. Nielsen's on usability and user tests [49], which states that a sufficient number for a usability test is 5. We decided to set the number of participants per part to 6, 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. 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.

Table 4.6 details, for each part, the number of participants per group and every mobile application considered (with the corresponding stars).

Part	Test Group	Control Group	Energy Pattern	Projects considered
1	3 participants	3 participants	Cache - Check Metadata	TimeTable (4⋆)
				SecondScreen (95⋆)
				ZimLx (136⋆)
				Taskbar (207⋆)
			Reduce Size - GZIP Compression	Audinaut (78⋆)
				Onpc (42⋆)
				Klingon-Assistant-Android (0⋆)
				Tracker-Control-Android (70⋆)
				Vanilla-music-lyrics-search (11⋆)
				DokuwikiAndroid (18⋆)
2	3 participants	3 participants	Cache - SSL Session Caching	Twire (82⋆)
				Inwallet (10⋆)
				donwload-navi (157⋆)
				akvo-rsr-up (3⋆)
			Cache - Passive Provider Location	OpenTopoMap (18⋆)
				Trekarta (50⋆)
				Acastus (12⋆)

Table 4.6: User study overview.

[3]The number of stars is the one the app had when the mobile applications where analyzed (25 June 2020).

4.5.2 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 were 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.

Table 4.7 presents relevant participant characteristics for this study, presenting their knowledge on Java, on developing mobile applications, and on Android Studio. 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.

Group	Occupation (%)		Android Studio experience (%)	Mobile application development experience (%)	Energy-efficient app development experience (1-5)	Java knowledge (1-5)
	MSc Student	Software Professional				
Test	83.33	16.67	50	16.67	1.67	3.5
Control	83.33	16.67	50	66.67	2.17	3.5

Table 4.7: Relevant characteristics of participants.

Appendix C (page 79) (User Study Documents) contains the documents shown to the participants in each part, with the links presented to them. Appendix D (page 85) (User Study Questionnaires) contains the questionnaires given to the participants, for both parts.

4.5.3 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 4.8 and 4.9 present the data obtained for both parts of the study.

Group	Task 1 [Check Metadata]				Task 2 [GZIP Compression]			
	Detected problem (%)	Detection time (min)	Solved problem (%)	Solving time (min)	Detected problem (%)	Detection time (min)	Solved problem (%)	Solving time (min)
Test	33.3	9.67	66.67	5.3	100	5	100	2.33
Control	33.3	10.67	100	8.33	100	2.67	100	4

Table 4.8: User study results: part 1.

Regarding Part 1 of the study, out of the three participants in the test group, only 2 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. In fact, one participant in the test group attempted to solve the 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.

Group	Task 1 [SSL Session Caching]				Task 2 [Passive Provider Location]			
	Detected problem (%)	Detection time (min)	Solved problem (%)	Solving time (min)	Detected problem (%)	Detection time (min)	Solved problem (%)	Solving time (min)
Test	100	7.67	100	1	100	4.67	100	1
Control	100	5	100	1.33	100	6.67	100	1.67

Table 4.9: User study results: part 2.

Regarding Part 2, every participant in the test group used the plugin to do both tasks. 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. In average, the control group requires more time than the test group, which suggests that the plugin can save time to developers.

Questionnaire. The final task performed by all the participants, of both the test and control groups, was to fill in a questionnaire to further understand their experience. 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 were altered during the refactoring process. During a 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 under which the energy pattern should be applied. One of the most common complaints among the users was that the warnings shown by EcoAndroid were easily missed, making it sometimes more difficult for the test group to find the location of the problem since they were looking for a warning. Unfortunately, this is not an alteration achievable (or even desirable) to the plugin since the presentation of the warnings is done by IDE itself, in both IntelliJ and Android Studio. One solution is to run the inspection task on the file or project since it presents the result as list. This was not done by any of the test group participants.

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).

Tables 4.10 and 4.11 present the answers given by participants of the user study. The score scale goes from 1 (Strongly disagree) to 5 (Strongly agree), except for the question "EcoAndroid inserts the enough comments explaining the change." where the scale is from 1 (Too few comments) and 5 (Too many comments).

Question	**Score**
The plugin is ready to use.	4.83
The comments that EcoAndroid adds to the source code explaining the changes performed are necessary to understand how the code changed.	4
A link to further documentation that helps understand the change would be helpful.	4.33
The energy pattern name is descriptive enough. [e.g. Cache - Check Metadata].	4
EcoAndroid inserts the enough comments explaining the change.	3.67
Overall, I consider EcoAndroid to be a useful plugin.	5
Overall, I would recommend that Android Java developers use EcoAndroid to assist them in creating more energy-efficient applications.	4.83

Table 4.10: Questionnaire answers - test group.

Question	Score
The description provided for the first pattern was clear.	4
The description provided for the second pattern was clear.	4.5

Table 4.11: Questionnaire answers - control group.

4.5.4 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 makes it 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.

5

Conclusions

Contents

In this chapter we discuss the contributions made and possible improvements of EcoAndroid. Section 5.1 describes the main contributions and whether or not the objectives proposed were met. Section 5.2 lists possible future work that could be done to further extend the plugin.

5.1 Achievements

The objective proposed by this thesis was to create a developer tool to aid in coding more energy-efficient mobile applications. The result was EcoAndroid. Compatible with both IntelliJ and Android Studio, EcoAndroid supports a total of 10 refactorings, over five energy patterns, and where two represent informational transformations. When using the plugin to inspect a set of 100 mobile applications, it found refactoring opportunities in 35 of them, having an average of one refactoring per 78 files. When evaluating EcoAndroid by sending pull requests to open-source GitHub projects, the feedback given by developers was mostly positive. Out of 42 pull requests sent, where 25 had feedback, only 5 were rejected. This represents an 80% acceptance rate in pull requests with response and an 46.62% acceptance rate when including all pull requests sent. When evaluating EcoAndroid through an user study, we verified that the plugin represented a saving of 1.43 minutes when fixing a code smell. While this may not seem substantial, we must note that participants in the control group had documentation regarding the energy patterns at their disposal. Every participant claimed that they find EcoAndroid usable and the alterations made by it understandable.

In Section 1.1 we proposed to answer four research questions, which were addressed throughout the document. Here we present a summary of each answer.

RQ1: What energy patterns are already known by the software engineering community? A total of 37 energy patterns are detailed in Section 2.2 (Energy Patterns for Mobile Applications). The energy patterns revolve around handling HTTP requests more efficiently, choosing darker backgrounds, continuously executing a method that does not need to be running at all times, among other practices. Specific examples are providing darker colors to the user, waiting for a WiFi connection before processing a non-urgent request, compress data before transmitting, switching from a energy greedy operation to a non-greedy one (implementing push notifications instead of a polling mechanism) and informing the user of a high energy consumption task being done. Moreover, according to the evaluation performed, there is a significant difference in EcoAndroid refactorings for each case supported (for example, EcoAndroid detected 14 opportunities regarding the pattern *Reduce Size - GZIP Compression*, but 0 opportunities regarding the pattern *Dynamic Retry Delay - Dynamic Wait Time*). This might be because some of these more frequently identified cases are not as familiar to developers, suggesting that more documentation on these cases would be helpful.

RQ2: What are the most relevant energy patterns to support? Since the goal of this project was to implement a refactoring tool, we defined criteria requiring that 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. This project focused on the energy patterns present in the catalog by Cruz et al. [5]. In this set of patterns, the ones we consider more relevant to implement in a refactoring tool were chosen, coming to a subset of five energy patterns: *Dynamic Retry Delay*, *Push over Poll*, *Reduce Size*, *Cache* and *Avoid Extraneous Graphics and Animations*.

RQ3: Are there existing tools that automatically apply energy patterns to the source code of mobile applications? As listed in Section 2.4 (Refactoring of Java Source Code), there are existing refactoring tools aimed at decreasing the energy consumed by mobile devices. Examples are Leafactor [35, 36], AEON [38] and Chimera [37]. The main 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 supported. Another relevant difference is the usability check performed. EcoAndroid was tested with 12 participants over four different cases, giving us a better understanding of how the alterations are perceived by real users.

RQ4: What are the challenges in automatically applying energy patterns? Both the inspections and refactorings implemented by EcoAndroid are performed with the aid of the PSI API, which is a layer of IntelliJ, making all the process dependent on the IDE (but still allowing for batch command-line execution). PSI enables plugin developers to handle the code in a structured way, presenting the elements as trees. In some cases, the alterations required might be considered substantial and, given that this is a refactoring tool and the alterations performed by it should never be too extensive, we face the challenge of determining when the pattern should be applied automatically. In these cases, the approach taken by EcoAndroid is to add a `//TODO`, i.e. an informational comment, before the method where the code smell is located explaining the change, usually with a web link for the documentation supporting it. In some cases it is possible to apply patterns completely automatically, but in other cases it is not possible.For example, in the case of the *Push Over Poll* energy pattern it is impossible since the registration of the class in Firebase is needed. To fix the problem, the same approach as before is used: adding a `//TODO` comment before the method.

5.2 Plugin Limitations and Future Work

As mentioned before, reducing the energy consumed by our mobile devices has become a growing concern in recent years. While EcoAndroid is able to help developers in coding more energy-efficient source code, it can still be improved. Some suggestions for future work include:

- **Inspecting the rest of the cases for the typical polling mechanisms:** as explained in Section 3.3.2 (Push Over Poll), Android documentation reports that there are three typical ways of defining background work. Currently, EcoAndroid only supports the inspection of one them. Future work could be supporting the other two approaches;

- **Supporting the remaining energy patterns of the catalog:** the catalog [5] we based our energy patterns has a total of twenty two (22) energy code smells. EcoAndroid supports five of these. Future work could be supporting the rest of the energy patterns present in the catalog;

- **Supporting the other energy patterns:** Section 2.2 lists various energy patterns, some of which are already considered by automated refactoring tools and others which do not have any refactoring support. Future work could support some of these energy patterns discussed that are not present in the catalog.

- **Implementing the feedback from the user study:** a straightforward conclusion from the user study is the need to add more comments to better understand the alterations made to the source code. However, some developers may not like a high number of comments. An approach to dealing with this problem could be having a link for the EcoAndroid GitHub project homepage and having there detailed explanations and links for the documentation for the class whose objects are altered. Future work could be supporting this feature;

- **Supporting energy patterns for the Kotlin language:** the inspection and refactoring are both aided by IntelliJ's PSI API. We chose to support these energy patterns for the Java language but the API is also available for the Kotlin language. Future work could be supporting these energy patterns for Kotlin;

- **Reducing the number of possible false positives:** the evaluation processed 100 mobile applications, but had a total of 1615 F-droid applications that could be inspected. Since false positives were found during the processing of the apps, future work could be repeating the evaluation process for the remaining data set;

- **Running EcoAndroid on an energy profiling tool:** measuring the energy consumed by a mobile device can be a difficult task, which is why it was not done in the evaluation of EcoAndroid. Even if complicated, integrating EcoAndroid with an energy profiling tool could be interesting to check the energy savings the energy patterns represent. ANEPROF [12] could be a good option since it is a real-measurement-based power profiling tool specific for Android, which EcoAndroid is focused on.

Bibliography

[1] 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.

[2] "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.

[3] "WHO air pollution," https://www.who.int/health-topics/air-pollution#tab=tab_1, 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] M. Petito, "Eclipse refactoring," *http://people. clarkson. edu/˜ dhou/courses/EE564-s07/Eclipse-Refactoring. pdf*, vol. 5, p. 2010, 2007.

[21] A. Rani and H. Kaur, "Refactoring methods and tools," *International Journal*, vol. 2, no. 12, 2012.

[22] H. Kegel and F. Steimann, "Systematically refactoring inheritance to delegation in java," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 431–440.

[23] "Autorefactor," http://autorefactor.org/, accessed: 2020-12-01.

[24] "Walkmod," http://walkmod.com/, accessed: 2020-12-01.

[25] "Facebook pfff," http://github.com/facebook/pfff/, accessed: 2020-12-01.

[26] "Kadabra," http://specs.fe.up.pt/tools/kadabra/, accessed: 2020-12-01.

[27] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[28] I. H. Moghadam and M. Ó Cinnéide, "Code-imp: a tool for automated search-based refactoring," in *Proceedings of the 4th Workshop on Refactoring Tools*. ACM, 2011, pp. 41–44.

[29] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Faultbuster: An automatic code smell refactoring toolset," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 253–258.

[30] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of android asynchronous programming (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 224–235.

[31] 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.

[32] G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró, "Integrating efficient model queries in state-of-the-art emf tools," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2012, pp. 1–8.

[33] J. Rajesh and D. Janakiram, "Jiad: a tool to infer design patterns in refactoring," in *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM, 2004, pp. 227–237.

[34] M. A. G. Gaitani, V. E. Zafeiris, N. Diamantidis, and E. A. Giakoumakis, "Automated refactoring to the null object design pattern," *Information and Software Technology*, vol. 59, pp. 33–52, 2015.

[35] 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.

[36] L. Cruz and R. Abreu, "Using automatic refactoring to improve energy efficiency of android apps," *arXiv preprint arXiv:1803.05889*, 2018.

[37] 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.

[38] "Aeon: Automated android energy-efficiency inspection," https://plugins.jetbrains.com/plugin/7444-aeon-automated-android-energy-efficiency-inspection, accessed: 2020-12-01.

[39] 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.

[40] 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.

[41] 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.

[42] O. Le Goaër, "Enforcing green code with android lint."

[43] 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.

[44] 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.

[45] "Program structure interface (psi)," https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html, accessed: 2020-12-01.

[46] 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.

[47] "Use firebase cloud messaging as an alternative to polling," https://developer.android.com/training/efficient-downloads/regular_updates, accessed: 2020-09-30.

[48] "F-droid," https://f-droid.org, accessed: 2020-12-01.

[49] "How many test users in a usability study," https://www.nngroup.com/articles/how-many-test-users/, accessed: 2020-12-01.

A

Top 100 F-Droid Applications

Top 100 F-Droid Applications Analyzed by EcoAndroid			
1	asdoi/TimeTable	51	js-labs/WalkieTalkie
2	92lleo/WhatsappWebToGo	52	TachibanaGeneralLaboratories/download-navi
3	ktt-ol/hacs	53	k3b/LosslessJpgCrop
4	mkulesh/microMathematics	54	nvllsvm/Audinaut
5	Rudloff/openvegemap-cordova	55	tobykurien/BatteryFu
6	pla1/FediPhoto	56	bradand/XMouse
7	mkulesh/onpc	57	knirirr/BeeCount
8	jfcolom/rosary	58	SecUSo/privacy-friendly-pain-diary
9	otakuhqz/ZimLX	59	btcontract/lnwallet
10	Kestutis-Z/World-Weather	60	devgianlu/PretendYoureXyzzyAndroid
11	dslul/openboard	61	tengusw/share_to_clipboard
12	senzhk/ADBKeyBoard	62	raatmarien/chibe
13	De7vID/klingon-assistant-android	63	andreynovikov/trekarta
14	agateau/pixelwheels	64	ivpn/android-app
15	Nonononoki/Hendroid	65	sanbeg/flashlight
16	farmerbb/SecondScreen	66	SecUSo/privacy-friendly-tape-measure
17	nelenkov/cryptfs-password-manager	67	AdrienPoupa/AttestationDeplacement
18	Willena/OpenDNSUpdater	68	beegee-tokyo/disaster-radio-android
19	dougkeen/BartRunnerAndroid	69	vanilla-music/vanilla-music-cover-fetch
20	NathanielMotus/Cavevin	70	wseemann/RoMote
21	Rudloff/lineageos-updater-shortcut	71	chaosdorf/meteroid
22	thetwom/toc2	72	billthefarmer/scope
23	OxfordHCC/tracker-control-android	73	billthefarmer/shorty
24	DoubleGremlin181/WhatsApp-Twitch-Stickers	74	billthefarmer/crossword
25	h0chi/next-companion	75	phikal/ReGeX
26	OpenArchive/Save-app-android	76	xavierfreyburger/tempus-romanum
27	rnauber/xskat-android	77	SecUSo/privacy-friendly-pin-mnemonic
28	vanilla-music/vanilla-music-lyrics-search	78	JohnLines/mediclog
29	fabienli/DokuwikiAndroid	79	wistein/TourCount
30	TeamNewPipe/NewPipe-legacy	80	wistein/TransektCount
31	rmst/yoke-android	81	phrogg/DSAAssistant
32	termux/termux-float	82	phrogg/BatteryCalibrator
33	matejdro/PebbleDialer-Android	83	rascarlo/ArchPackages
34	Pygmalion69/OpenTopoMapViewer	84	developerfromjokela/motioneye-client
35	Perflyst/Twire	85	Harvie/NorthDog
36	Abdallah-Abdelazim/mynotes-app	86	Stypox/mastercom-workbook
37	Alcidauk/CineLog	87	akvo/akvo-rsr-up
38	seguri/Lock	88	SecUSo/privacy-friendly-werewolf
39	ruleant/getback_gps	89	GittyMac/MoClock
40	benjaminaigner/aiproute	90	forrestguice/SuntimesCalendars
41	Catfriend1/syncthing-android-fdroid	91	btimofeev/instead-launcher-android
42	hoihei/Silectric	92	freshollie/UsbGps4Droid
43	arnowelzel/periodical	93	gjedeer/Acastus
44	ajh3/NoSurfForReddit	94	HappyPeng2x/SumatoraDictionary
45	fistons/TinyTinyFeed	95	CorvetteCole/GotoSleep
46	Tortel/SysLog	96	punksta/volume_control_android
47	mcastillof/FakeTraveler	97	cetoolbox/cetoolbox
48	apcro/leafpicrevived	98	quaddy-services/DynamicNightLight
49	fei0316/snapstreak-alarm	99	tmarzeion/drawable-notepad
50	devgianlu/DNSHero	100	KeikaiLauncher/KeikaiLauncher

Table A.1: Mobile applications considered in the evaluation. These are the top 100 applications when sorting by descending order according to the criteria: percentage of pull requests accepted, date of last commit, total merged pull requests, number of GitHub stars, and number of GitHub watchers.

B

State of Pull Requests

Application Name	Energy Pattern Applied	Status	Reason for rejection
Timetable	CheckMetadata	Accepted	-
SecondScreen	CheckNetwork	Rejected	Class did not declare android.permission.INTERNET
SecondScreen	CheckMetadata	Accepted	-
Hacs	CheckMetadata	Rejected	Refactoring could break some notification
ZimLX	CheckMetadata	Accepted	-
ZimLX	CheckMetadata	Accepted	-
Twire	SSLSessionCaching	Accepted	-
Twire	SSLSessionCaching	Accepted	-
Twire	SSLSessionCaching	Accepted	-
OpenTopoMapViewer	PassiverProviderLocation	No response	-
OpenTopoMapViewer	PassiverProviderLocation	No response	-
GetBack GPS	PassiverProviderLocation	No response	-
Taskbar	CheckMetadata	Accepted	-
NewPipe	CheckNetwork	No response	-
NewPipe	SSLSessionCaching	No response	-
NewPipe	SSLSessionCaching	No response	-
Audinaut	CheckNetwork	Accepted	-
Fistons	SSLSessionCaching	No response	-
Trekarta	PassiverProviderLocation	No response	-
Trekarta	CheckMetadata	No response	-
Inwallet	SSLSessionCaching	Accepted	-
Download Navi	SSLSessionCaching	Accepted	
Syncthing-Fork	SSLSessionCaching	Rejected	Refactoring does not effect energy consumption
Akvo RSR Up	SSLSessionCaching	No response	-
Acastus Photon	PassiverProviderLocation	No response	
RoMote	CheckNetwork	No response	-
Omega	CheckMetadata	Accepted	-
Hentoid	CheckNetwork	Rejected	Class does not use network
Onpc Cordova	GZIP Compression	Accepted	-
Rosary Cordova	GZIP Compression	No response	-
World-Weather	GZIP Compression	No response	-
Klingon-Assistant-Android	GZIP Compression	Accepted	-
Tracker-Control-Android	GZIP Compression	Accepted	-
Tracker-Control-Android	GZIP Compression	Accepted	-
Tracker-Control-Android	GZIP Compression	Accepted	-
Tracker-Control-Android	GZIP Compression	Accepted	-
Vanilla-music-lyrics-search	GZIP Compression	Accepted	-
BatteryFu	GZIP Compression	No response	-
Trekarta	GZIP Compression	No response	-
TourCount	GZIP Compression	Rejected	Refactoring does not effect energy consumption
DokuwikiAndroid	GZIP Compression	Accepted	-
Cordova-Android	GZIP Compression	No response	-

Table B.1: State of pull requests.

C

User Study Documents

C.1 Part 1

C.1.1 Test Group

EcoAndroid: User Study

Thank you very much for participating in this user study on the application of energy-related patterns in mobile applications. We will focus on Android applications written in Java and will ask you to undertake two programming tasks. The goal is to apply certain energy patterns to the code that we will present to you so that it becomes more energy-efficient.

Please note that there are many ways of approaching and solving the tasks proposed. If you feel unable to solve any task just let us know. It might be possible to give you a hint that will help you proceed.

This user study lasts for about one hour. Near the end, we will ask you to fill in a questionnaire.

Resources

You will have access to the IDE Android Studio, where two projects will be already open. To help you in

solving the tasks proposed, you can access the catalogue Energy Patterns for Mobile Apps. In the tasks below, we give additional links that might be helpful.

You will also have access to the plugin EcoAndroid that could be helpful when applying the energy patterns. The plugin inspects the source code looking for energy code problems and, later, refactors the source code accordingly.

Task 1: Apply the Pattern Cache

The goal of Task 1 is to apply the pattern Cache in the project Taskbar, which is already open in Android Studio. We propose that you follow these steps:

1. Read the description of the pattern Cache in the catalog. The commit listed in the catalog might be helpful in understanding how this pattern is normally applied. Also, the documentation for the class BroadcastReceiver may be useful.

2. Find an opportunity to apply the pattern in the project Taskbar. If after 10 minutes you do not know where to apply it, you can ask for a hint.

3. Apply the pattern. You can compile the code to check whether your changes are accepted, but you do not need to use time running the application.

Task 2: Apply the Pattern Reduce Size

The goal of Task 2 is to apply the pattern Reduce Size in the project Tracker-Control-Android, which is already open in Android Studio. We propose that you follow these steps:

1. Read the description of the pattern Reduce Size in the catalog. The following example may be useful. Also, the documentation for the class URLConnection may be useful.

2. Find an opportunity to apply the pattern in the project Tracker-Control-Android. If after 10 minutes you do not know where to apply it, you can ask for a hint.

3. Apply the pattern. You can compile the code to check whether your changes are accepted, but you do not need to use time running the application.

Conclusion

Once the two tasks above are concluded, we will ask you to fill in a quick questionnaire.

All the data collected in this study is anonymous and might be used in research publications.

Many thanks for participating!

C.1.2 Control Group

EcoAndroid: User Study

Thank you very much for participating in this user study on the application of energy-related patterns in mobile applications. We will focus on Android applications written in Java and will ask you to undertake two programming tasks. The goal is to apply certain energy patterns to the code that we will present to you so that it becomes more energy-efficient.

Please note that there are many ways of approaching and solving the tasks proposed. If you feel unable to solve any task just let us know. It might be possible to give you a hint that will help you proceed.

This user study lasts for about one hour. Near the end, we will ask you to fill in a questionnaire.

Resources

You will have access to the IDE Android Studio, where two projects will be already open. To help you in solving the tasks proposed, you can access the catalogue Energy Patterns for Mobile Apps. In the tasks below, we give additional links that might be helpful.

Task 1: Apply the Pattern Cache

The goal of Task 1 is to apply the pattern Cache in the project Taskbar, which is already open in Android Studio. We propose that you follow these steps:

1. Read the description of the pattern Cache in the catalog. The commit listed in the catalog might be helpful in understanding how this pattern is normally applied. Also, the documentation for the class BroadcastReceiver may be useful.

2. Find an opportunity to apply the pattern in the project Taskbar. If after 10 minutes you do not know where to apply it, you can ask for a hint.

3. Apply the pattern. You can compile the code to check whether your changes are accepted, but you do not need to use time running the application.

Task 2: Apply the Pattern Reduce Size

The goal of Task 2 is to apply the pattern Reduce Size in the project Tracker-Control-Android, which is already open in Android Studio. We propose that you follow these steps:

1. Read the description of the pattern Reduce Size in the catalog. The following example may be useful. Also, the documentation for the class URLConnection may be useful.

2. Find an opportunity to apply the pattern in the project Tracker-Control-Android. If after 10 minutes you do not know where to apply it, you can ask for a hint.

3. Apply the pattern. You can compile the code to check whether your changes are accepted, but you do not need to use time running the application.

Conclusion

Once the two tasks above are concluded, we will ask you to fill in a quick questionnaire.
All the data collected in this study is anonymous and might be used in research publications.
Many thanks for participating!

C.2 Part 2

C.2.1 Test Group

EcoAndroid: User Study

Thank you very much for participating in this user study on the application of energy-related patterns in mobile applications. We will focus on Android applications written in Java and will ask you to undertake two programming tasks. The goal is to apply certain energy patterns to the code that we will present to you so that it becomes more energy-efficient.
Please note that there are many ways of approaching and solving the tasks proposed. If you feel unable to solve any task just let us know. It might be possible to give you a hint that will help you proceed.
This user study lasts for about one hour. Near the end, we will ask you to fill in a questionnaire.

Resources

You will have access to the IDE Android Studio, where two projects will be already open. To help you in solving the tasks proposed, you can access the catalogue Energy Patterns for Mobile Apps. In the tasks below, we give additional links that might be helpful.
You will also have access to the plugin EcoAndroid that could be helpful when applying the energy patterns. The plugin inspects the source code looking for energy code problems and, later, refactors the source code accordingly.

Task 1: Apply the Pattern Cache - SSL Session Caching

The goal of Task 1 is to apply the pattern Cache in the project Download-Navi, which is already open in Android Studio. We propose that you follow these steps:

1. Read the description of the pattern Cache in the catalog. The following commit may be useful. Also, the documentation for the class SSLSessionContext may be useful.

2. Find an opportunity to apply the pattern in the project Download-Navi. If after 10 minutes you do not know where to apply it, you can ask for a hint.

3. Apply the pattern. You can compile the code to check whether your changes are accepted, but you do not need to use time running the application.

Task 2: Apply the Pattern Cache - Passive Provider Location

The goal of Task 2 is to apply the pattern cache in the project Trekarta, which is already open in Android Studio. We propose that you follow these steps:

1. Read the description of the pattern Cache in the catalog. The commit listed in the catalog might be helpful in understanding how this pattern is normally applied or this other commit. Also, the documentation for the class LocationManager may be useful.

2. Find an opportunity to apply the pattern in the project Trekarta. If after 10 minutes you do not know where to apply it, you can ask for a hint.

3. Apply the pattern. You can compile the code to check whether your changes are accepted, but you do not need to use time running the application.

Conclusion

Once the two tasks above are concluded, we will ask you to fill in a quick questionnaire.
All the data collected in this study is anonymous and might be used in research publications.
Many thanks for participating!

C.2.2 Control Group

EcoAndroid: User Study

Thank you very much for participating in this user study on the application of energy-related patterns in mobile applications. We will focus on Android applications written in Java and will ask you to undertake two programming tasks. The goal is to apply certain energy patterns to the code that we will present to you so that it becomes more energy-efficient.
Please note that there are many ways of approaching and solving the tasks proposed. If you feel unable to solve any task just let us know. It might be possible to give you a hint that will help you proceed.
This user study lasts for about one hour. Near the end, we will ask you to fill in a questionnaire.

Resources

You will have access to the IDE Android Studio, where two projects will be already open. To help you in solving the tasks proposed, you can access the catalogue Energy Patterns for Mobile Apps. In the tasks below, we give additional links that might be helpful.

Task 1: Apply the Pattern Cache - SSL Session Caching

The goal of Task 1 is to apply the pattern Cache in the project Download-Navi, which is already open in Android Studio. We propose that you follow these steps:

1. Read the description of the pattern Cache in the catalog. The following commit may be useful. Also, the documentation for the class SSLSessionContext may be useful.
2. Find an opportunity to apply the pattern in the project Download-Navi. If after 10 minutes you do not know where to apply it, you can ask for a hint.
3. Apply the pattern. You can compile the code to check whether your changes are accepted, but you do not need to use time running the application.

Task 2: Apply the Pattern Cache - Passive Provider Location

The goal of Task 2 is to apply the pattern cache in the project Trekarta, which is already open in Android Studio. We propose that you follow these steps:

1. Read the description of the pattern Cache in the catalog. The commit listed in the catalog might be helpful in understanding how this pattern is normally applied or this other commit. Also, the documentation for the class LocationManager may be useful.
2. Find an opportunity to apply the pattern in the project Trekarta. If after 10 minutes you do not know where to apply it, you can ask for a hint.
3. Apply the pattern. You can compile the code to check whether your changes are accepted, but you do not need to use time running the application.

Conclusion

Once the two tasks above are concluded, we will ask you to fill in a quick questionnaire.
All the data collected in this study is anonymous and might be used in research publications.
Many thanks for participating!

D

User Study Questionnaires

D.1 Test Group

EcoAndroid - User Study

Thank you so much for participating in the EcoAndroid user study! As a last task, we ask you to answer the following questions so we can better understand your experience in the study.
Note: All the data collected in this study is anonymous and might be used in research publications.

Questions about the user

This section is about getting to know more about the users and their knowledge about Java, Android Studio and developing mobile applications.

What is your current occupation?

- ☐ BSc Student
- ☐ MSc Student
- ☐ PhD Student

☐ Software Professional (Not a student)

	← yes				no →
Have you ever used Android Studio before?	☐				☐
Have you ever developed a mobile application?	☐				☐

	← no knowledge				expert →
How do you assess your knowledge of practices for building more energy-efficient mobile applications?	☐	☐	☐	☐	☐
How do you assess your own Java knowledge?	☐	☐	☐	☐	☐

Questions about the user study

This section is about better understanding your experience in this study.

	← strongly disagree				strongly agree →
The plugin EcoAndroid is easy to use.	☐	☐	☐	☐	☐
The comments that EcoAndroid adds to the source code explaining the changes performed are necessary to understand how the code changed.	☐	☐	☐	☐	☐
A link to further documentation that helps understand the change would be helpful	☐	☐	☐	☐	☐
The energy pattern name is descriptive enough. [e.g. Cache - Check Metadata].	☐	☐	☐	☐	☐

	← too few comments				too many comments →
EcoAndroidinserts the enough comments explaining the change.	☐	☐	☐	☐	☐

	← strongly disagree				strongly agree →
Overall, I consider EcoAndroidto be a useful plugin.	☐	☐	☐	☐	☐
Overall, I would recommend that Android Java developers use EcoAndroid to assist them in creating more energy-efficient applications.	☐	☐	☐	☐	☐

What did you like the most about EcoAndroid?_____

What did you like the least about EcoAndroid?_____

Is there anything you would change in EcoAndroid?_____

D.2 Control Group

EcoAndroid - User Study

Thank you so much for participating in the EcoAndroid user study! As a last task, we ask you to answer the following questions so we can better understand your experience in the study.

Note: All the data collected in this study is anonymous and might be used in research publications.

Questions about the user

This section is about getting to know more about the users and their knowledge about Java, Android Studio and developing mobile applications.

What is your current occupation?

- ☐ BSc Student
- ☐ MSc Student
- ☐ PhD Student
- ☐ Software Professional (Not a student)

	← yes				no →
Have you ever used Android Studio before?	☐				☐
Have you ever developed a mobile application?	☐				☐

	← no knowledge				expert →
How do you assess your knowledge of practices for building more energy-efficient mobile applications?	☐	☐	☐	☐	☐
How do you assess your own Java knowledge?	☐	☐	☐	☐	☐

Questions about the user study

This section is about better understanding your experience in this study.

	← strongly disagree				strongly agree →
The description provided for the first pattern was clear.	☐	☐	☐	☐	☐
The description provided for the second pattern was clear.	☐	☐	☐	☐	☐

What did you find easiest about the application of the first energy pattern?_____

What did you find more difficult about the application of the first energy pattern?_____

What did you find easiest about the application of the second energy pattern?_____

What did you find more difficult about the application of the second energy pattern?_____