TÉCNICO LISBOA

Automatic Refactoring for Energy Efficiency in Continuous Integration Pipelines

Ricardo José Horta Morais

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu
Member of the Committee: Prof. Pedro Manuel Guerra e Silva Reis dos Santos

September 2020

Acknowledgments

Many things are hard to achieve without the support of others, and they should be recognized for their part, this dissertation was only possible through the support of others, in particular the continuous support that I have been consistently given by my friends, family and mentors.

First, I would like to thank my family for supporting me, not only during my dissertation, but also through all the years of my education, without their support I would not have been able to reached this point.

I would also like to thank my friends, in particular Otelo Magalhães and Artur Esteves for being there when I needed them the most.

I am also grateful for the help of both the JavaParser and Spoon support teams in particular Martin Monperrus whose help was of the most importance.

All the participants of the survey should also be acknowledged in particular those that were not anonymous: João Morais, Artur Esteves, Márcio Pamplona and Filipe Correia. A special thanks to João Morais for helping me review the final submission of this dissertation.

Finally, I want to acknowledge my dissertation supervisors Prof. Rui Maranhão and Prof. Luís Cruz for their continuous support, patience and knowledge that has made this dissertation possible.

Abstract

Contemporary society demands more than is currently possible for battery technology on mobile devices. Developers should meet this necessity by designing mobile applications that take energy efficiency into account.

Energy-conscious practices have yet to proliferate in the mobile development community and are often left behind because developers do not know how to apply them and why they are important, for instance bad energy efficiency in applications tend to lead to bad application reviews and consequently less sales. Moreover, developers are not equipped with tools that help in that regard.

In this work I introduce LeafactorCI, a software solution that assists developers by automatically refactoring energy inefficient anti-patterns on android projects, allowing them to focus on creative work. LeafactorCI stands out because it was designed to be lightweight, adaptable, and simple, to be easily introduced to continuous integration environments. LeafactorCI is evaluated on the GitHub platform with the TravisCI integration which are the most popular Version Control System platform and CI service, respectively.

Keywords

Green mining, mobile, energy efficiency, automated refactoring, continuous integration, anti-patterns.

Resumo

A sociedade contemporânea exige mais do que é atualmente possível da tecnologia de bateria em dispositivos móveis. Os programadores devem atender a essa necessidade projetando aplicativos móveis que levem em consideração a eficiência energética.

As práticas conscientes de energia ainda estão por proliferar na comunidade de desenvolvimento móvel e geralmente são deixadas para trás porque os programadores não sabem como aplicá-las e/ou por que são importantes, por exemplo, a má eficiência energética em aplicativos tende a levar a críticas menos agradáveis a aplicativos e, consequentemente, a menos vendas. Além disso, os programadores não estão equipados com ferramentas que ajudam nesse sentido.

Neste trabalho, apresento o LeafactorCI, uma solução que ajuda os programadores a refatorar automaticamente antipadrões de energia em projetos Android, permitindo que eles se concentrem em trabalho criativo. O LeafactorCI destaca-se pelo fato de ter sido projetado para ser leve, adaptável, simples e para ser facilmente introduzido em ambientes de integração contínua. O LeafactorCI é avaliado na plataforma GitHub com a integração do TravisCI, que são as plataformas mais populares de sistema de controle de versão e de serviço de integração contínua, respectivamente.

Palavras Chave

Continuous Integration, Energy bugs, Energy efficiency, Anti-patterns, GIT, Gradle, Spoon, Android, Java

Contents

List of Figures

Listings

Acronyms

CI	Continuous Integration
FAQ	Frequently Asked Questions
DBMS	Database Management System
GPS	Global Positioning System
UI	User Interface
CPU	Central Processing Unit
ORM	Object-Relational Mapping
AMOLED	Active-Matrix Organic Light Emitting Diode
LCD	Liquid Crystal Display
XML	Extensible Markup Language
JDT	Java Development Tool
IDE	Integrated Development Environment
AST	Abstract Syntax Tree
GUI	Graphical User Interface
LGPL	Lesser General Public License
MIT	Massachusetts Institute of Technology
CeCILL-C	CEA CNRS INRIA Logiciel Libre
VCS	Version Control System
CLI	Command Line Interface
API	Application Programming Interface
UML	Unified Modeling Language
SDK	Software Development Kit

1

Introduction

Contents

1.1 Motivation

In recent years, there has been an increase in efforts by the scientific community to improve the energy efficiency of mobile devices through the improvement of the Application-level [8, 10, 11]. In particular, Cruz and Abreu studied the impact of fixing eight Android performance-related anti-patterns on energy efficiency [10] and concluded that there are five anti-patterns, that do positively influence energy efficiency, specifically: ViewHolder, DrawAllocation, WakeLock, ObsoleteLayoutParam, and Recycle. By exploring this fact, in a later study, Cruz and Abreu introduced Leafactor [11, 15], a refactoring utility that automatically cleanses android projects of four of those anti-patterns.

Because Android developers are in need of an answer to their energy bugs that considers their development practices [11, 25], this dissertation introduces Leafactor as an open-source continuous integration solution that helps them to easily purge energy-efficient anti-patterns in their source code. Unlike other solutions, this dissertation focus on automation and adaptability by releasing a new implementation of Leafactor, called LeafactorCI, published as a Gradle plugin powered by Spoon. The fact that most Continuous Integration (CI) services provide Docker containerization technology means that the execution of Gradle tasks is widely supported. Most Android applications are built on top of Gradle.

A significant number of benefits can be obtained from adopting CI [14, 24, 38, 40]. Vasilescu et al. assessed the effects of continuous integration by gathering data from GitHub [38]. They collected 247 GitHub projects that at some point introduced CI and found that after CI was added, more Pull-Requests from the core developers were accepted, and fewer rejected. In addition, fewer submissions from non-core developers got rejected, suggesting that CI both improves the handling of Pull-Request from insiders as well as outsiders. On the other hand, they found that CI did not decreased user-reported bugs. However, there was a decrease of developer-reported bugs, which suggests that CI is helping developers in that regard.

1.2 Objective

Developers lack tools to properly enhance the energy consumption footprint of their mobile apps and most online resources are oriented on how to improve app performance, which not always translates to improving energy efficiency [25]. To solve this problem, it is proposed a solution that helps them clear energy bugs in android applications and relieve them of energy efficiency concerns. The adaptation to CI practices was also considered as there is a growing number of developers turning to CI [14]. This dissertation objectives will be accomplished by:

- Introducing LeafactorCI, a Gradle based plugin with a smaller footprint and better performance than its predecessor (Leafactor) by re-implementing the refactoring rules using a slimmer and

faster technology.

- Improving usability by providing LeafactorCI as a Gradle plugin, enabling its integration with Android projects and to adapt to several CI scenarios, enhancing its chances to proliferate inside the developer community.
- Suggesting a strategy for delegating refactoring decisions to the developer through the automatic creation of branches containing the fixes, such that they can be merged after manual acceptance.
- Creating a test battery in order to establish the baseline of support and avoid future regressions.
- Documenting the tool and publishing it in an alpha version.
- Evaluating the final solution by answering the following questions by either conducting a user study on a set of volunteers or through analysis and demonstration:
  - Can LeafactorCI be used inside a CI environment?
  - How easy it is to adopt LeafactorCI?

1.3 Contributions

The product of this dissertation is LeafactorCI, a tool to refactor four types of energy bugs (Wake Lock, View Holder, Recycle, and Draw Allocation) that can form in the code of Android Java projects. With LeafactorCI, developers are able to check if these energy bugs are present in their code, and if they are, they are able to fix them automatically. For an even more automatic experience, LeafactorCI was designed to be used inside a CI environment. This dissertation suggests a strategy to bootstrap an automatic refactoring process. LeafactorCI comes out as an open-source project at `https://github.com/TQRG/leafactor-ci` and is launched in its alpha version at `https://plugins.gradle.org/plugin/tqrg.leafactor.ci`. Developers are welcome to contribute to the project and/or use it as a base to conduct further research on other energy bugs.

1.4 Dissertation structure

The rest of this document is organized as follows: The chapter 2 explores the related work; Chapter 3 presents the design and implementation of the solution to the problems exposed in the related work; Chapter 4 is dedicated to the evaluation of the resulting solution; Chapter 5 concludes with final remarks. And finally appendix A presents part of the code of the solution.

2

Related Work

Contents

This section presents the state of the art and reflects on the investigation behind the design and implementation decisions for this work. This section is divided into four distinct sub-sections, each represents a conceptional step, from the fundamental to the supplementary, it is divided in the following way: the Section 2.1 describes the two most prevalent profiling methods used in mobile green mining[1] studies; the Section 2.2 evaluates studies that explore the intricacies of patterns in the source-code of mobile applications and how some of those patterns influence energy efficiency; Section 2.3 features studies that propose tools and methods to automatically refactor out the previously mentioned anti-patterns; and finally Section 2.4 illustrates how CI is positively disrupting how software solutions are developed and lays a motivational foundation for supporting the CI ecosystem.

2.1 Energy Profiling

Studies on mobile green mining rely on different means of profiling the battery consumption. Usually either one of two methods are used: direct hardware measurement; or energy consumption estimation. Each leads to different results. On one hand, direct hardware measurement provides higher certainty in energy consumption as the sensors measurement is very accurate (provided that the probing arrangement is reliable), this method, however, is more expensive, lacks flexibility and practicality due to the set up complexity. On the other hand, despite estimating energy consumption being simpler and less expensive, it will most likely not produce such accurate results, as the models cannot portray every detail of reality.

In this section, a simplified version of Linares-Vásquez et al. [27] approach to classifying energy profiling methods was used along with the following dimensions:

- Apps - number of applications used in the evaluation of the proposed technique;
- Approach - approach used for collecting and estimating energy measurements:
  - Hardware-based profiling (HBP) - physical measurement
  - Power models (PM) - modeling the factors that come into play in energy consumption
  - Android Battery API (ABA) - Android provides an API to access data taken from the embedded sensors in the device
- Profiling granularity - the artifacts that are considered and utilized in the profiling effort
  - Application (A)
  - Flow Path (FP) - determine application paths traversed and track energy-related information during an execution

[1] Methodology of reducing energy costs by optimizing the underlying system

Technique	Apps	Approach	Element	Tool	Year
Sahin et al. [35]	-	HBP	P	H	2012
Pathak et al. [32]	21	PM	S,A,F,APIC	EP	2012
Li and Halfond [25]	6	PM	FP,F,S,A	VL	2013
Linares-Vásquez et al. [27]	55	HBP	APIC	M,H	2014
Li et al. [26]	400	PM	APIC,F,S,A	VL,M,PT	2014
Cruz and Abreu [10]	6	HBP	S,P	H	2017

Table 2.1: Paper classification

  - Function (F)
  - API Calls (APIC) - the call made to the Android SDK
  - Statement (S)
  - Patterns (P)

- Tool - the tool that were used for acquiring the data
  - vLens (VL)
  - PowerTutor (PT)
  - eCalc (EC)
  - eProf (EP)
  - Monsoon power monitor (M)
  - Hardware (H)

Ahmad et al. also classified the state-of-the-art of energy profiling back in 2015. In their work they organized the studies into two energy profiling schemes, Software Based Profiling and Hardware Based Profiling. Figure 2.1 shows how they classified the studies. We, however, use Linares-Vásquez et al. approach because it is simpler, and contains the relevant details.

This work is a follow up of the study done by Cruz and Abreu. Nevertheless it is important to compare it with similar studies. A small set of relevant and diverse papers from 2012 and above were selected, their methods were analysed and classified. The classification can be seen in Table 2.1. The most similar study to [10] (Cruz and Abreu) can be seen in the table to be [27] (Linares-Vásquez et al.) with 3 years apart. They both used Hardware based profiling but evaluated different artifacts, Linares-Vásquez et al. evaluated API calls, and Cruz and Abreu evaluated patterns and statements. Linares-Vásquez et al. considered a vast range of application, 55 to be exact, while Cruz and Abreu considered six.

One example of energy consumption estimation can be shown in [25] where Li and Halfond conducted a small-scale empirical evaluation of commonly suggested energy-saving and performance-enhancing coding practices. Through the use of energy consumption estimation, they concluded that

techniques like bundling network packets up to a certain size and using certain coding practices for reading array length information, accessing class fields, and performing invocations all led to reduced energy consumption. In another empirical study [26] on energy consumption of android applications based on power consumption estimation, Li et al. found that apps spend more than 60% of their energy in idle states and that network is the most energy consuming component, therefore they postulate that optimizing the Application-level alone is not enough. However, a number of studies [9, 10, 35] have demonstrated that there is in fact space for improvement in the Application-level and that any improvement, even if small, is relevant.

In [10], Cruz and Abreu emphasized the particularities of energy profiling and reviewed empirical studies that based their findings on data obtained from tools such as PowerTutor [21], eProf [17], and eCalc [23]. In the same study Cruz and Abreu use a hardware power measuring device to evaluate their work which they introduced in another study [12], justifying that estimation software is usually only compatible with specific smartphone models and Android versions, making evaluation very difficult. Further, they showed that it is possible to improve energy efficiency by up to 5% just by making changes to the Application-level which can equate to a significant quantity of battery life minutes saved.

Linares-Vásquez et al. hinted for a trade-off between design principles and battery saving [27], in particular they expose ”information hiding” as an expensive principle and advice to disregard it by giving direct assess to private fields. They also highlight that using a Database Management System (DBMS) may have a non-negligible impact on the battery consumption [27], and it should be avoided unless strictly necessary.

Pathak et al. point out that some of the energy is consumed by asynchronous mechanisms such as GPS, Wi-Fi, camera, etc. which make it difficult to trace-back energy consumption [32]. They published and used eProf a fine-grain energy profiling tool that can overcome that obstacle. eProf can be used in conjunction with static analysis techniques to develop energy optimizers that automate the process of restructuring app source code to reduce their energy footprint [32].

From what we can gather, even thought those studies use different profiling approaches, all of them point out that there is room for improvement. This work is based on the work done by Cruz and Abreu in [10] which used Hardware Based Profiling to show how specific patterns in the code influence energy consumption. The next section explores how patterns can have an effect on energy consumption.

2.2 Patterns

Classic design patterns, like those identified in the book released by the Gang of Four[18][2], fail to influence energy consumption when introduced in the design phase [35], i.e. it is not enough to simply adopt

[2]The authors of the book Design Patterns: Elements of Reusable Object-Oriented Software (1994) that describe 23 classic object-oriented software design patterns

one or more design patterns in the design phase to improve energy efficiency e.g. Java applications often apply the Model-View-Controller design pattern, which seems to be less energy efficient compared to other forms of organizing the code [27]. Consequently, developers seeking to improve the energy footprint of their mobile applications have to consistently pay attention to bad development practices.

Recently a number of studies have been demonstrating that there is a correlation between code smells[3] and energy efficiency [8, 29] and subsequently exploring this fact, for instance Banerjee and Roychoudhury in [8] used an approach where a design-expression is extracted from a given application. This design-expression represents the application's event graph abstraction. Where there is a non empty intersection between the design-expression and a defect-expression (an expression that represents undesired anti-patterns), a refactoring is done in order to undo the intersection. In [29], Morales et al. led a preliminary study that evaluated if anti-patterns influence energy consumption, additionally if different types of anti-patterns(object-oriented and android specific anti-patterns) influence energy consumption differently and concluded that removing "Binding resources too early", "Private getters" and setters, "Refused Bequest", and "Lazy class" anti-patterns can improve energy efficiency.

In previous work, Cruz and Abreu compiled a guideline for fixing a set of five patterns which form on Android projects and impact energy efficiency. They are:

- **View Holder** - This pattern appears in List Views. When in a List View, the system has to draw each item and the problem arises when the method findViewById is called a number of times, this method is known for being a very expensive method.
- **Draw Allocation** - Allocating objects during a drawing or layout operation is a bad practice. Allocating objects can cause garbage collection operations that will slow down the operation and create a nonsmooth User Interface (UI).
- **Wake Lock** - Wake locks are mechanisms to control the power state of the mobile device. This can be used to wake up the screen or the Central Processing Unit (CPU) when the device is in a sleep state in order to perform tasks. If an application fails to release a wake lock or uses it without being strictly necessary, it can drain the battery.
- **Obsolete Layout Param** - During development, UI views might be refactored several times. In this process, some parameters might be left unchanged even when they have no effect on the view. This causes useless attribute processing at runtime leading to battery consumption.
- **Recycle** - There are collections such as TypedArray that are implemented using singleton resources. The problem occurs when the resource is not released properly, leading to inefficient resource management.

[3] A term popularized by Martin Fowler that designates code patterns that can be found in software that indicate weaknesses in design that may slow down development or increase the risk of bugs or failures in the future

A large scale study by Cruz and Abreu on energy practices of mobile applications [16] inspects 1027 Android project and their respective commits, pull request, and issues to find energy-related changes. Cruz and Abreu delivered a catalog of 22 design patterns to improve energy efficiency. Their work exposes 22 energy-related practices. This study, however, provides no evidence for the benefits of using them.

Verdecchia et al. in a recent study evaluated the energy impact of five code-smells on three Open Source Java Object-Relational Mapping (ORM) based applications [39]. The code-smells were:

- Feature Envy - A method that is placed in the wrong class because it invokes other classes more than its own class, which leads to low cohesion.
- Type Checking - Control flow mechanisms such as 'if' and 'switch' statements that check the type of the variables at runtime instead of using compile time mechanisms.
- Long Method - A method too large that could be divided into smaller ones in order to improve maintainability.
- God Class - A class that controls and centralizes a great part of the architecture leading to maintainability inconvenience.
- Duplicated Code - The same code structure appears in different parts of the code.

Verdecchia et al. showed that all of the above mentioned code-smells, when fixed lead to improvements in energy efficiency of at least one (out of the three) of the sample applications, in particular "Feature Envy" and "Long Method" provided the best results. While fixing "God Class" and "Duplicated Code" did show improvements in the energy efficiency, the fix also impacted performance negatively, which means that there seems to be a trade-off in those two cases. They also noticed that the impact of fixing those code smells depends on multiple factors like the size and the age of the application.

Mannan et al. analysed 20 code-smells, 500 Android and 750 desktop applications and found that there seems to be more code-smells on android applications than desktop applications like shown in Figure 2.2.

The distribution clearly shows that there is a high number of code smells such as the Feature Envy and God Class that Verdecchia et al. shown to improve energy efficiency.

Gottschalk in his thesis [22] described and evaluated five energy bugs thoroughly:

- Third-Party Advertisement - where advertisements are used in applications that do not require internet connection.
- Binding Resources too Early (BRTE) - where resources such as Wi-Fi and Global Positioning System (GPS) are turned on way before they are needed. (This pattern is also explored in [10] by Cruz and Abreu).

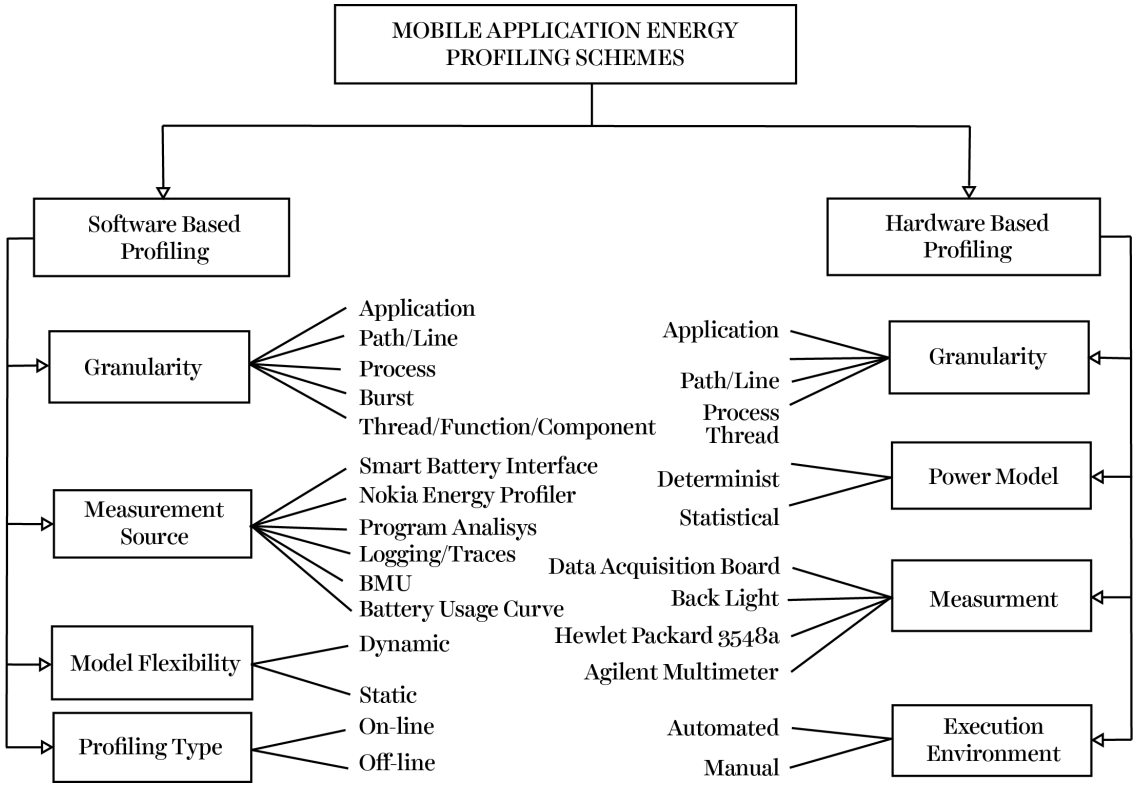

Figure 2.1: Classification introduced by Ahmad et al. in 2015 [7]

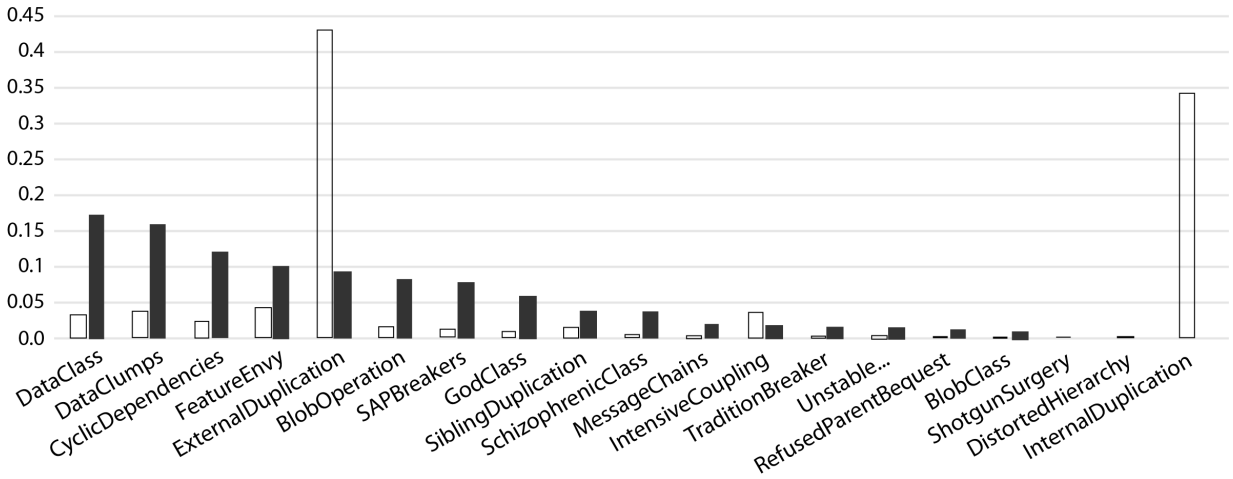

Figure 2.2: Distribution of code-smells in desktop and android applications taken from [28]

- Statement Change - where an 'if' statement can be changed to a 'switch' statement or vice-versa(similar to the Type Checking code-smell mentioned earlier) when it improves readability.
- Backlight - where the background color of the application may have an effect on the energy efficiency on different screen technologies such as Liquid Crystal Display (LCD) and Active-Matrix Organic Light Emitting Diode (AMOLED).
- Data Transfer - where data is loaded from the server instead of the application storage.

Palomba et al. used a lightweight tool called aDoctor to identify 15 Android-specific code smells [31] from the 2014 Reimann et al. catalogue [33] founded on the traditional code-smells from **?** book [**?**], however they did not study the energy impact of those code smells. Much like Leafactor, this study used a similar approach to detecting anti-patterns in the code, they used static analysis of Android projects.

We have seen in this section that patterns can be identified in the code of applications that worsen the consumption of energy. The next section will take a look at how automated refactoring can be used to cleanse those patterns.

2.3 Automated Refactoring

Refactoring techniques can be used to improve software design by altering its structure without changing its functionality [20]. Refactoring focus on the nonfunctional attributes of software, therefore, it can be used to improve the energy efficiency of mobile applications.

Cruz et al. used the technique to fix anti-patterns in the source-code of mobile applications. In earlier work [11], Cruz and Abreu showcased Leafactor, a toolset designed to automatically purge five android specific anti-patterns that negatively affect energy consumption. Leafactor is divided into two engines, one is a Java refactoring engine based on the open-source project AutoRefactor [34] and the other is an Extensible Markup Language (XML) refactoring engine made from scratch to deal with layout related anti-patterns, at the time the latter only takes care of the dead code. Because most of Leafactor was implemented on top of AutoRefactor, which depends on the Eclipse Java Development Tool (JDT) library which makes it, therefore, bound to either be used as an Eclipse Integrated Development Environment (IDE) plugin or as a headless[4] plugin. This is a disadvantage as it restricts its domain and therefore its usefulness.

There are essentially two ways to go about source-code analysis, static analysis and dynamic analysis. In static analysis the source code of the application is analysed usually with a model such as an Abstract Syntax Tree (AST) of it which disregards the actual execution of the application, from there the model is analysed and conclusions are made [22]. In dynamic analysis the application execution is

[4]Headless plugins run in what is known as the headless mode of Eclipse. This mode does not start the graphical interface, hence it is faster and uses less memory. This mode is ideal for interfacing with other tools that are not part of Eclipse.

```
1 public class GpsPrinter extends Activity {
2     public void onCreate() {
3         requestLocationUpdates();
4     }
5     public void onResume() {
6         requestLocationUpdates();
7     }
8 }
```

Figure 2.3: Sample code taken from [22].

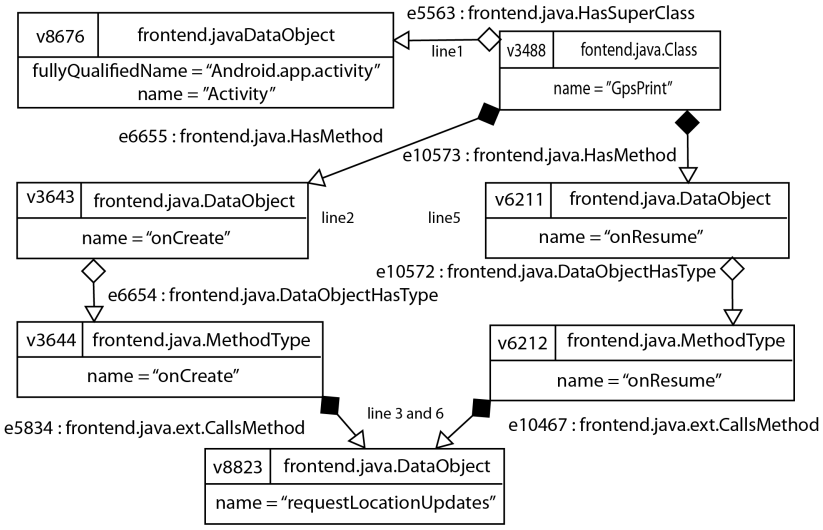

Figure 2.4: TGraph representation taken from [22].

considered which allows for the extrapolation of usage data [22]. Cruz and Abreu use static analysis to identify the patterns in the code.

Gottschalk implemented a refactoring tool called EnergyRefactoring [22] in 2013. It uses a form of static analysis, something called the TGraph which is generated from the original application to create a tree-like representation of the code, then with a query language called GReQL[5] a query is made to identify the specific pattern and finally with JGraLab API[6] and the result of the query a transformation is made to remove the pattern. Figure 2.3 is an example of code taken from [22] and Figure 2.4 demonstrates its representation in TGraphs.

Morales et al. published a refactoring approach titled EARMO [29]. This approach balances between two conflicting objectives: design quality; and energy efficiency. They claim that this approach enabled them to achieve a remarkable extension in battery life. Unlike Cruz et al. that selected a set of anti-patterns that affect energy efficiency negatively, Morales et al. focused on multi-objective optimization, where the main goal is to improve the quality of the code while controlling the energy efficiency. This approach takes into account all sorts of code-smells(object-oriented code-smells as well as mobile specific code smells), but in particular code-smells that do affect energy consumption, and for a very simple

[5]https://bit.ly/2FMdN9Q
[6]https://bit.ly/2CwOxBk

reason: refactoring such code-smells could lead to worse energy efficiency. While this approach seems to be promising, it has yet to, as far as we know, materialize as a publicly available tool.

This section exposed some of the approaches currently being used to tackle this problem. This work will use the static analysis approach for pattern identification and AST to represent the code structure and apply the refactorings. AST's are simple and well studied, they go hand in hand with static analysis and there are a great number of tools available. The next section dwells into CI and explores how a tool can be made such that it is adaptable to CI environments.

2.4 Continuous Integration

Martin Fowler defined Continuous Integration as "a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly" [19]. Following this definition it can establish that this practice involves people contributing frequently with the expectation of reaching a series of competitive advantages. Ståhl and Bosch studied process related differences in CI practices [37], they advocate that CI is an umbrella for a variety of practices and that those practices should be taken into consideration as those may not represent CI as a whole. They analyzed the literature and categorized the claims that were made and found that there are currently no consensus on what CI is as a single homogeneous practice and that simply stating that a project is using CI is often insufficient information. Hence, they present a model that exerts finer granularity which effectively distinguishes CI practices from one another, the model helps to understand which of the practices better relates to the goals. The most relevant question in the context of this project are:

1. Build duration - The build frequency needs to be considered. Is it three times a week, every day? It matters.
2. Build triggering - The mechanism used to trigger a build should be explicit. Is it on commit, scheduled, manual, etc?
3. Integration frequency - How often developers integrated their work.

These points should be taken into considerations when adopting any CI tool in order for it to be used effectively. For example, if a project is only built every 6 months it may not be worth investing in the automation.

Testing is a fundamental part of continuous integration [14] as such it needs to be taken seriously. Cruz et al. investigated the working habits and challenges of mobile app developers with respect to test-

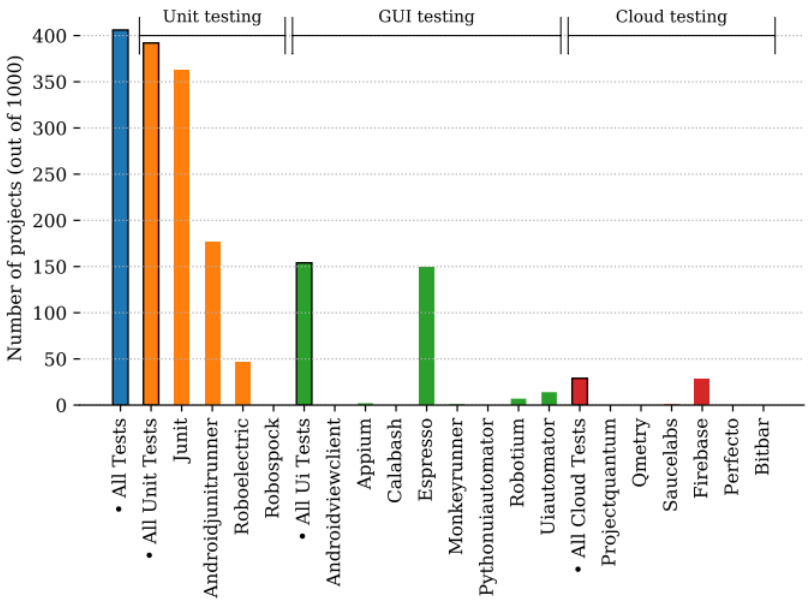

Figure 2.5: Distribution of projects between the testing tools taken from [14].

ing [14]. They conduct a large-scale study on 1000 open-source android applications and concluded that android apps are failing to use automated testing. The study showed only 40% of the applications used testing technologies. Figure 2.5 shows the distribution of testing technologies on the 1000 applications. The testing technologies were: JUnit used in unit testing; and Espresso used in Graphical User Interface (GUI) testing. Cloud testing services are not widely adopted, the most used technology is Google Firebase. Cruz et al. also found that the most popular CI service is TravisCI. Cruz et al. explain that it is important to simplify the learning curve and setup of such tools in order for them to be adopted. Even though the status of testing and CI looks grim, there is a growing number of mobile developers becoming aware of the importance of testing [14].

Zhao et al. led a large-scale empirical study using regression discontinuity design analysis to quantitatively evaluate the effects of adopting CI and how the developers are following Martin Fowlers advice on applying continuous integration [40]. They found that after adopting CI:

- The number of commits starts to align with the ”commit often” principle, however this might be due to the shift to a more distributed work-flow.
- The ’commit often’ was followed only to some extent and depended on the actual project.
- The number of issues closed increased but eventually slowed down
- After initial adjustments, the amount of automated tests seemed to increase.

In 2016, Hilton et al. published a study on the usage, cost and benefits of CI [24]. Hilton et al. identified that there was not enough research done in this area despite CI rising as a successful practice and that, because of this, developers, tool builders, and researchers make decisions based on anecdotes instead of data. The study goes on to analyze 34,544 GitHub open-source projects, adding up to 1,529,291 builds from the most commonly used CI system and survey 442 developers. 40% of the projects evaluated used CI and they predict that the adoption rates will increase. They also discovered that the median time of adoption is one year and the main reason why open-source projects choose to not use CI is that the developers are not familiar enough with it. In this regard, providing means of education, improving the ease of use and raising awareness can eventually reduce the time of adoption.

Although CI practices have their benefits, they can also be challenging to implement. As synthesized by Shahin et al., there are several factors to consider [36] such as: testing(effort and time); team awareness and transparency; good design principles; customer; highly skilled and motivated team; application domain; appropriate infrastructure.

An organization should contemplate their current situation before assimilating CI into their stack. Further, Shahin et al. compiled a set of practices that, when correctly applied, can lead to successful incorporation of CI [36]:

- **Improve team awareness and communication** e.g. using a changelog, labeling versions and features, etc;
- **Planning and documenting** e.g. having a planned path for adopting continuous integration and document builds, tests and other activities related to the integration;
- **Promote team mindset** by organizing events about continuous practices to spread mindset and giving freedom to developers;
- **Improve team qualification and expertise** through formal training and coaching team members;
- **Define new roles and teams** establishing a dedicated team to develop and maintain deployment pipeline;
- **Adopt new rules and policies** e.g. enforcing a rule that all developers should be on call when releasing software;
- **Improve testing activity** by practicing test-driven development, cross team testing etc;
- **Branching strategies** by using short-lived feature branching;
- **Decompose development into smaller units** by breaking down large features and changes into smaller and safer ones;

Organizations and teams can exercise such practices with the support of CI services such as Circle CI and Travis CI.

Taking the previous said into account, it is my desire to make my contribution in a continuous integration format as there seems to be a rising trend in the adoption of these practices. There is still a lack of adoption of CI practices in mobile development, thus hopefully this project contributes to move the needle forward in their adoption.

3

Architecture and implementation of LeafactorCI

Contents

The previous chapter synthesized part of the literature surrounding the problem of improving mobile energy efficiency problems. Based on this synthesis, we can assume that there may be anti-patterns in the source code of android projects that when resolved can lead to a significant improvement on the energy efficiency of a particular application. Furthermore, such anti-patterns can be resolved through the usage of refactoring tools. Such refactoring tools can also be streamlined using continuous integration practices in order to improve the workflow of the development teams and maintain an automated resolution to energy inefficiency problems. In this section, it is detailed the approach taken in this work on the development of LeafactorCI as an effective solution for purging anti-patterns in Android applications, as well as its overall architecture.

3.1 Overview

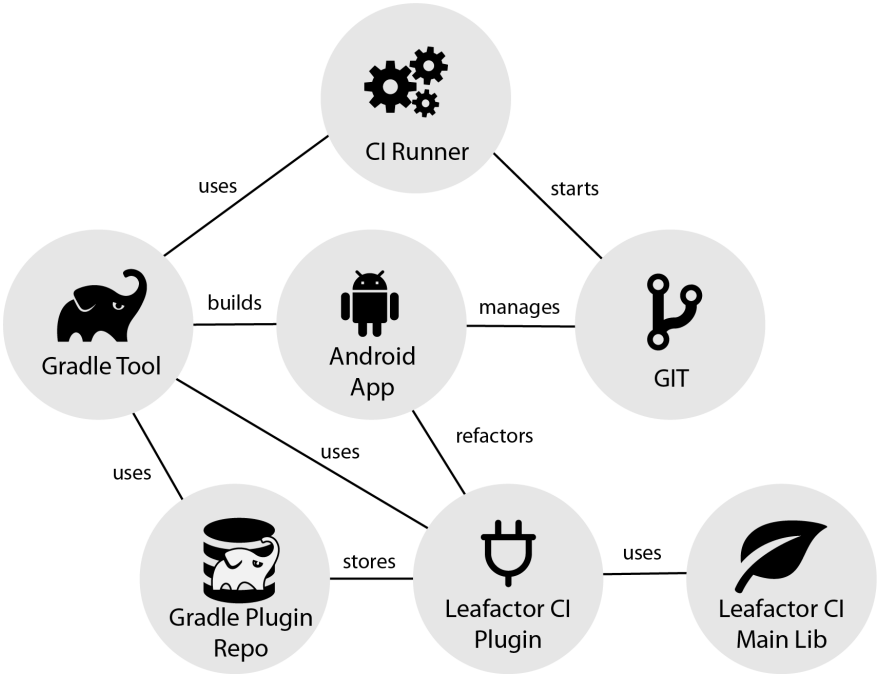

Figure 3.1: Architecture

LeafactorCI is not just a software application, it is a software solution. As such it expands beyond the realm of a single system. Its purpose is to solve a problem, to remove patterns in the source code of Android applications, but in an elegant and easy manner, such that it becomes inconspicuous in the development process.

The architecture aims at taking advantage of some of the existing practices for Android application development. Such as the usage of the GIT version control system and the automatic usage of CI platforms based on containerization technology.

The solution revolves around the Spoon refactoring engine. Spoon is the library that provides support for querying and refactoring the source-code of the Android applications. Using it, four refactoring rules were implemented to refactor each of the Recycle, Draw Allocation, View Holder, and Wake Lock patterns mentioned in the Section 2.2. This implementation is defined as the **LeafactorCI Main Library**. In order to facilitate the integration of the tool with the Android environment, a Gradle plugin was created (**LeafactorCI Plugin**) which allows for any Android project to integrate and use the **LeafactorCI Main Library**. The plugin serves as an interface between the Gradle build tool and the **Leafactor Main Library**. The **LeafactorCI Plugin** is also published in the **Gradle Plugin Repository**, which makes it readily available for download. Now, since Gradle is used, it can be leveraged in a continuous integration environment, since a virtual container[1] can be launched automatically on specific conditions such as when a commit is made in the main branch of the Android application repository. Such a container can be used to run a Gradle task that starts a refactoring on the code through the **LeafactorCI Plugin**. Since GIT is used to clone the Android application to the container it is possible for the changes made by the **LeafactorCI Main Library** to be committed back to a separate branch, leaving the developers with the option to either merge or delete the branch.

3.2 Refactoring

While the concept of refactoring anti-patterns is simple, one must wonder how an actual implementation will interact with the source code. The code of an Android Application is, put simply, a collection of Java source files written to disk memory and therefore can be manipulated in many ways. A refactoring engine that leverages the AST model was chosen to do the manipulation of such files. First, JavaParser[4] was tried. It was not a good fit (due to reasons described in the subsequent subsections), therefore Spoon[5] was used instead. The decision of which refactoring library to use took a few requirements into account.

3.2.1 General requirements for the refactoring library

When deciding on a library, a couple of general key points must be taken into consideration. These are some of the points that were used to select the library:

- Maintenance - Is the library being maintained consistently?
- Community - Is the community active? Are they submitting new issues or questions? Are they creating new features?
- History - Is this library new, has it been tested enough?
- Source Availability - Can the source code be modified and inspected?

[1] A trimmed out runnable layer of an operating system, used with technologies such as Docker and Vagrant.

3.2.2 Specific requirements for the refactoring library

For this distinct use case the library needs to meet additional requirements. The following points were considered:

- AST model - Does it parse the source code as an AST model?
- Version support - Are they supporting the Java versions that are required to support? For Android it needs to support Java 7 and 8 features.
- Library language - Is the library language widely adopted? Does it have a good ecosystem of libraries?
- Low unintended alterations - After modifying the source code, does it look exactly like the original with the exception of the modified parts?
- Efficiency - Does it run fast? Does it take too much time to parse and modify the code?
- Documentation - Is there documentation available? Is it good?

3.2.3 JavaParser

JavaParser is a Java library for analysing, transforming and generating Java source code from Java version 1.0 to 14. It is open-source and hosted on GitHub under a Lesser General Public License (LGPL) or Apache License. At present time it has 3,073 stars, 712 forks and 1,720 dependents. Currently it is being maintained at a steady pace by a single contributor.

The library was first created in 2008 as a simple parser for the 1.5 version of Java. Over time, it became a very popular choice and, at the present date, the latest Java version is 14. It was create by Sreenivasa Viswanadha and Júlio Vilmar Gesser and hosted at Google Code[1]. From there Danny van Bruggen migrated it to GitHub and has been maintaining it ever since. As he started to accept community contributions the comunity and the library grew to what is is today. In 2016 the library JavaSymbolSolver created by Federico Tomassetti in 2015 was added to JavaParser. Later he added a lexical preserving parser that allowed the parser to preserve the style format of the code while parsing.

3.2.4 Analysing the requirements

JavaParser library was analysed with the previously mentioned points to see how it fits the necessary requirements:

- Maintenance - The library is maintained at a steady pace. However, it was discovered that not all of the source code is being maintained at the moment, only part of it.

- Community - There are new issues being submitted by users but it doesn't seem that the community is contributing to the source code.
- History - It was created in 2008 and been improving since then. Still, there are very recent features.
- Source Availability - The code is open-source under a LGPL or Apache License.
- Documentation - Good enough, there is free book about it. [30].
- AST model - It parses the source code to an AST model that can be manipulated and used to generate new code.
- Version support - They support both Java 7 and Java 8 versions that are required.
- Library language - The library is for Java language which is a very widely used language with a big ecosystem.
- Low unintended alterations - It promises to modify the code without compromising the format of the original, however, later it was found out that this feature (Lexical Preservation) was still underdeveloped.
- Efficiency - Well over the needed speed, it did not seem to be a cause for concern.

The library met the use case requirements in a very promising way. It lead far as it enabled the implementation of all of the 4 anti-pattern refactoring rules. However, the Lexical Preservation was was faulty, as simple cases such as retaining the indentation of the original source files were a problem. Ultimately, the issues found in this part of the library brought the implementation to a dead end with unreasonable compromises. Some attempts to fix some of the problems in the Lexical Preservation feature were made, but there were just too many issues to resolve. Since this is considered to be a fundamental feature, the project started from scratch with another library. This was evaluated in the year 2019 so improvements to the Lexical Preservation feature could have been done since then.

3.2.5 Spoon

Spoon is a Java library for analysing, transforming and generating Java source code from Java version 1.0 to 14. It is open-source and hosted on GitHub since 2014 under a double Massachusetts Institute of Technology (MIT) or CEA CNRS INRIA Logiciel Libre (CeCILL-C) license. At present moment it has 946 stars and 202 forks. It has mainly one steady contributor and some sparse contributors.

The library was first introduced by Renaud Pawlak and Nicolas Petitprez in Inria Lille in 2004[2]. After that, Carlos Noguera did a PhD thesis in 2006 with the existing technology. In 2013 it was revived thought subsequent PhD thesis from Benoit Cornu, Matias Martinez and Thomas Durieux. In 2016 Spoon joined the OW2 open-source consortium. And has been steadily been maintained since.

3.2.6 Analysing the requirements

The Spoon library was analysed with the previously mentioned points to see how it fits the necessary requirements:

- Maintenance - The library is maintained at a steady pace.
- Community - There are new issues being submitted by users and the community usually contribute by providing unit tests for the issues they are submitting, which help the maintainers to fix the problems quickly.
- History - It was created in 2004 and been improving since then. Still, there are very recent features.
- Source Availability - The code is open source under a double MIT or CeCILL-C license.
- Documentation - Very good and to the point. Although some of the internals and new features are not very well documented.
- AST model - It parses the source code to an AST model that can be manipulated and used to generate new code.
- Version support - They support both Java 7 and Java 8 versions that are required.
- Library language - The library is for Java language which is a very widely used language with a big ecosystem.
- Low unintended alterations - It promises to modify the code without compromising the format of the original. Later it was found that it had minor bugs that could be resolved.
- Efficiency - Well over the needed speed, it did not seem to be a cause for concern.

Spoon allowed me to achieve the implementation of the 4 refactoring rules whilst still maintaining the format of the original source code when possible. Although there were (and still are) bugs in the Sniper Pretty printer (the equivalent of the JavaParser Lexical Preservation feature), they were more manageable and the support was there to help me solve the issues. The meta-model for the AST in Spoon was, in my opinion, more comprehensible and simpler than that of JavaParser, the Spoon team chose to use inheritance which to me best suits my usual approach and choices. The Java language meta is not a dynamic structure so to me this is where inheritance rules over composition.

A community comparison between JavaParser and Spoon can be found in [6], where both the contributors of Spoon and JavaParser reach consensus on what either side lacks or features.

3.3 Version Control

Android projects are usually managed via an implementation of a Version Control System (VCS)[2] and the GIT VCS is at present moment the most popular option[3]. Since the objective here is to make changes in the source code of the application, it would be wise to leverage GIT mechanisms, two in particular:

- Branching - mechanism which allows code to be submitted without compromising the integrity of the code being manually modified and published by developers
- Merging - feature which allows the modifications on a branch to be joined with the modification of another branch, and therefore can be used to join the changes made in a separate branch into the main branch after such changes are accepted.

The branching and merging mechanics are important for differing any changes done to the Android project to when they are accepted. LeafactorCI is not a perfect tool and as such it can make mistakes, therefore the responsibility for its refactors must be given to the project integrator. For example, the main development branch can be forked into a new one such that LeafactorCI can run a refactoring job on top of it. Changes can then be committed and left for an integrator to review and later merge into the main development branch. Now, in order to make the job of the integrator easier, the changes made to the source code must be kept at a minimum. JavaParser was unable to keep the code changes at a minimum and usually would make unnecessary changes to the code style. If JavaParser was to be used as the refactoring library, the job of the integrator would be harder, because he would have had to go through all the changes that were made to accept them, even those that were unrelated to the anti-pattern refactors. Spoon on the other hand is able to keep it at a minimum. The branching and merging mechanics are also very important for continuous integration as a branch can be automatically created, the changes committed and eventually accepted.

3.4 Distribution and usage

LeafactorCI is intended to be easy to use, and that means that the configuration steps should be kept at a minimum. The way that this is done in LeafactorCI is by leveraging Gradle, the default build tool that is used in the development of Android applications. LeafactorCI Main Library is bundled in the form of a Gradle plugin that can conveniently be installed without much configuration. Since Gradle has all the information of the Android project, the plugin can access that information without requiring user intervention. The Gradle plugin can also be published in the Gradle Plugin Repository, so that a user can

[2] Version control systems are a category of software tools that helps record changes to files by keeping a track of modifications done to the code.

choose to install a particular version from there. Gradle also helps in preparing a continuous integration task as it already has all the setup necessary to build the project.

The alternate solution of developing a Command Line Interface (CLI) was considered given it would provide a more versatile approach. However, the decision not to make it stood on the increasing number of user configurations with inherited use complexity and need of documentation. Thus it was decided that Gradle would be more practical.

3.5 Continuous Integration

To use LeafactorCI it is not required to have a continuous integration system in place, it is optional, but it is helpful in automating the refactoring process. In order to design LeafactorCI to support continuous integration, some requirements need to be considered:

- Fast run-time.
- Low download size.
- Easy setup.

As described in the previous section 3.4 LeafactorCI is bundled as a Gradle Plugin. Gradle is not particularly fast at building, it depends on the size of the project. A project with more dependencies and artifacts will take a longer time than a smaller project. All things considered, the Gradle build is the most expensive operation of the refactoring process. This could have been avoided by using a CLI instead since the LeafactorCI Main Library does not require the project to be built as it can work on individual files (this feature is called ”noClasspath” mode in Spoon). However, the benefits of Gradle out-weight the speed of the simpler CLI alternative, this is also the case for the download size as Gradle would still need to be downloaded. In the future, it may be necessary to provide the full classpath to the LeafactorCI Main Library, meaning that dependencies can be analyzed in the AST which may help to make better refactoring decisions, Gradle facilitates this endeavor by providing an Application Programming Interface (API) for discovering the entire classpath of the project.

The envisioned way of integrating LeafactorCI with CI is:

1. Prepare a virtual container using a virtualization technology such as Docker or Vagrant. Alternatively use a virtual machine.
2. Schedule the container/virtual machine to run a script e.g. when a commit is pushed to the main development branch, such that it runs the following sequence:
   (a) Clone the GIT repository.

(b) Checkout the branch that you want to analyse (e.g. the main development branch).

(c) Create a new local branch with a recognizable name.

(d) Run the refactoring Gradle task.

(e) Commit the changes with an adequate message.

(f) Push the changes to the remote repository.

3. Review/correct the changes done to the newly created branch or discard it.

4. Merge the changes into the original branch.

With this setup developers can focus on making changes of the features. After those features are pushed to the main branch the CI runner triggers a script execution, such that the refactoring process starts. The changes are then put in a branch that can be integrated in the main development branch after they are accepted. If GitHub is used, a pull-request can alternatively be made to review and merge the changes.

Now that we know the overall structure of the solution, the remaining of this chapter will explain how the technologies were used to achieve it and dwell in greater detail on the implementation details. All of the mentioned code references relate to the alpha version of LeafactorCI (`https://github.com/TQRG/leafactor-ci`) and commits up to September of 2020.

Spoon does not enforce any methodology for purging anti-patterns, in fact it simply lets us find, add, modify and remove nodes in the AST. We can expect a large number of different scenarios leading to the same kind of anti-pattern to form, the disposition of variables and control flow in the code can complicate things. For instance, let us consider the Recycle anti-pattern, which the objective is to release an acquired resource after using it. Now picture a method that acquires such a resource and sends it to another method if some condition is met. We know that the resource should be recycled, but where it should be recycled is the question. There are a number of cases of interest that need to be evaluated in order to decide what is the right modification to be applied. LeafactorCI uses a pipeline algorithm to detect and process such cases. The algorithm is divided into 4 phases to deal with the process of concisely detecting cases of interest, transforming them and refactoring them inside imperative blocks of code. Structuring the refactoring process this way allows for measurements to be taken and should lead to more consistency and predictability. The artifact responsible for refactoring each of the anti-patterns using the four phases will be referred as a Refactoring Rule.

3.6 Refactoring Rule

In the main library the RefactoringRule is an interface (Figure 3.2) that extends the Spoon Processor interface along with some others. In Spoon, processors are the visitors of the nodes in the AST. The

Processor interface also uses Java Generics in order to specify the type of node that will be visited in the tree (See more in Spoon and Visitor software pattern). This means that every refactoring rule implements this interface, and during execution it will search for specific types of nodes in the AST. Four refactoring rules were made so that they search for CtClass objects (Java classes), they explore from there by traversing the tree. The RefactoringRule interface also extends other important interfaces. It extends the Iterable interface which provides the declaration for a number of life-cycle methods used during the iteration of the code blocks, they serve as hooks for the iteration algorithm. It also extends from the interfaces CaseDetector, CaseTransformer and CaseProcessor which means that a RefactoringRule has methods for detecting, transforming and processing cases of interest (CaseOfInterest interface).

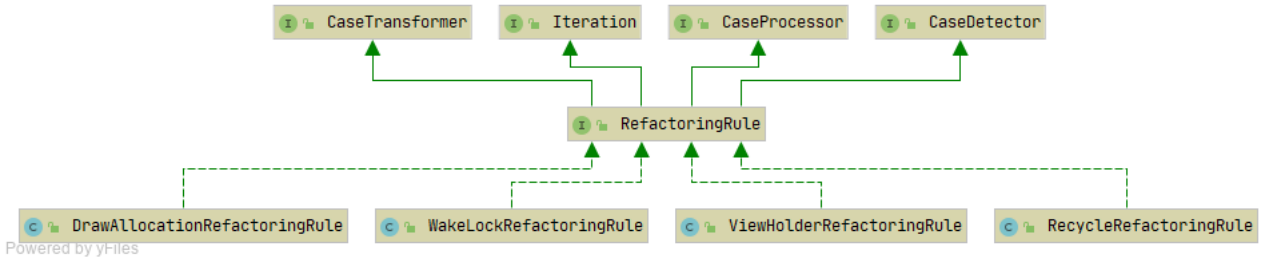

Figure 3.2: Refactoring rules Unified Modeling Language (UML) diagram

The refactoring rules make use of a iteration method called iterateBlock that governs the way that the rule life-cycle methods are called. The source code for the iterateBlock can be seen in the code listing (A.1). The iterateBlock method accepts a RefactoringRule object, along with a logger and a CtBlock (which is a block of code from the source code that is being evaluated e.g. a method). The purpose of the function is to organize the refactoring process into phases.

First, it starts with a setup phase that begins in line 11, of the code listing (A.1). The code prepares the logger that will record the time intervals of each phase execution. After that the algorithm will call the onSetup life-cycle method of the refactoring rule in order for it to internally prepare for the iteration.

Then comes the detection phase that starts in line 22, in this phase each of the block statements will be evaluated one at a time in sequence. For each statement, the onWillIterate, the detectCase and the onDidIterate methods of the refactoring rule object will be called. Ideally, if the block has inner blocks (e.g a method with an if statement inside), we might want to process the inner block using the same algorithm, however, in practice and for the purpose of refactoring the 4 refactoring rules, this was not necessary and the code was commented. In this phase the cases of interest that are detected by the refactoring rule are collected and sent to the next phase.

Next comes the transformation phase that starts in line 39. The idea of the transformation phase is to allow the refactoring rule the option to evaluate all the collected cases of interest and filter them, add more or mutate them e.g. there can be a case of interest that only happens when two other cases

of interest are present, so in this stage we can, for example, remove both of them and create a new one. Three life-cycle methods are called on the refactoring rule during the phase for each cases of interest, namely, the onWillTransformCase, the transformCase and the onDidTransformCase. After the transformation, the remaining cases of interest are sent to the final stage.

The final stage is the refactoring phase that starts in line 55. This is where the AST model will be altered by the refactoring rule. For each case of interest, three life-cycle methods are called on the refactoring rule, the onWillRefactorCase, the refactorCase and the onDidRefactorCase. Finally the logs of the phases are collected and are added to the logger.

Four refactoring rules were implemented that used this iteration algorithm, one for each of the anti-patterns, respectively:

- RecycleRefactoringRule.java
- DrawAllocationRefactoringRule.java
- WakeLockRefactoringRule.java
- ViewHolderRefactoringRule.java

The following subsections will briefly explain how each of the patterns manifests and what heuristics were used to detect and refactor the anti-patterns along with the refactoring rule class implementations. All of the following refactoring rules are by no means perfect and there can be many situations where they would not work or lead to false positive situations due to their infancy, this is safeguarded by the inspections done by the project integrator. This section will not be presenting any transformation examples as they can be found in the project test resource folder under root/src/test/resources. Also, in the following code listings, parts of the code may not be present if it was deemed irrelevant, such as empty life-cycle methods and import statements.

3.6.1 RecycleRefactoringRule.java

The recycling refactoring rule is meant to prepare the code to release resources improperly managed. This rule has a set of known resource classes that it knows to be of interest, for now it uses only the name of the resource class to identify the class, which may lead to incorrect results but this is good enough enough a means of flagging potential problems, eventually the entire package of the class will be evaluated for more accurate results.

In line 3 of the code listing (A.4) we can see the opportunities Map attribute that is used to hold all the classes that are flagged as resource classes. In the line range from 9 to 16 we can see the insertions of the flagged classes and their respective resource release methods. In this case there are 8 hard-coded classes. Eventually by using full classpath mode in Spoon we could detect some Recycle interface and

see if the evaluated class inherits from it, making it more generalized. As it is, these 8 are the ones that are verified. They were taken from Leafactor AutoRefactor, which is the predecessor of LeafactorCI.

As previously mentioned, every refactoring rule class extends the Spoon Processor interface and expects to visit CtClass's. The visitor method is implemented in the line 294. It accepts the CtClass element and extracts its methods, and then calls its private refactor method (implemented in line 287) for each of the methods in the CtClass element. The refactor method then extracts the CtBlock's of the methods and for each of them runs the iterateBlock method described previously. The iterateBlock method will start the algorithm and run each of the stages for that particular CtBlock.

In the detection phase the detectCase method (line 20) of the recycle refactoring rule is called and it looks for five types of cases of interest:

- VariableDeclared - When a variable is declared. Later used to check if the variable is one of the resource flagged types.
- VariableReassigned - When a variable is reassigned to another value. This normally means that if the variable had an allocated resource it will no longer reference it and/or that It can be assigned the value of another allocated resource.
- VariableUsed - The variable is being used somewhere. This is important in order to recycle the variable after the last variable usage.
- VariableLost - The code block lost the variable. This means that the code block lost control over the variable and cannot ascertain its last usage, e.g. the variable was sent to another method, therefore it is not the responsibility of this block to recycle the variable.
- VariableRecycled - The variable is under control and was recycled properly, we do not wish to make any changes. This case of interest prevents a refactoring from happening on an already recycled variable.

Each of the cases detected instances are put inside the detection phase context. This detection does not account for the relation between the cases of interest, they are independent of one another. Only in the transformation phase and refactoring phase are they analysed together. Reason being to improve extensibility, as some of these detectors can be used by other refactoring rules.

The transformation phase of the recycle refactoring rule (method transformCase in line 56) is basically a filter to check if the cases of interest apply to the conditions of a refactoring, it tests if the used variables were defined as classes that were flagged (at that point it will only detected variable usages but not the type of the variables) and that they were declared in this block (if there was a VariableDeclared case of interest with the same variable name). The same is done for the VariableReassigned cases of

interest. VariableUsed and VariableReassigned cases of interest without a matching variable declaration are ignored.

Finally in the refactoring phase (method refactorCase in line 279) the refactoring rule invokes 3 refactoring methods for processing the remaining cases of interest. The cases of interest prone to lead to refactoring are the VariableDeclared, VariableReassigned and VariableUsed, and as such each type of case has its own evaluation method. In the recycleVariableDeclared method (line 154), if and only if there is no variable usages in the code a recycling statement is pushed inside an if statement into the code after the variable declaration statement. The if statement checks if the variable is null and if it is not the recycle method is called. We only use the VariableDeclared case for refactoring in this particular case since we do not have any variable reassignments and variable usages and the variable needs to be recycled. The logic behind this is that VariableUsed cases are more important for refactoring, so the edge cases where VariableUsed are not present are dealt by analysing the other cases of interest.

For every VariableUsed case of interest we will have a refactoring happen if and only if:

- This is the last VariableUsed case of the respective variable
- The variable was not recycled right after the last statement.
- The variable is still in control (e.g. it was not sent to another method).

When this conditions are met the recycling statement is pushed below the last variable usage.

For the VariableReassigned, we only consider a situation where the variable was not used, was not recycled and is still in control before the reassignment since the last declaration or reassignment. If so we push a recycling statement right before the reassignment, assuring that the variable reference that will be lost is properly recycled.

There are of course many more possibilities to consider, which will be left to the community to bring them forward.

3.6.2 DrawAllocationRefactoringRule.java

The draw allocation anti-pattern occurs when we start to allocate variables in the onDraw method. Allocating such variables can cause garbage collection operations that will slow down the execution and lead to energy inefficiency. The refactoring rule must find allocations in the onDraw method (the onDraw method has a specific signature) and remove them without compromising the integrity of the code. The rule is simple, once a variable initialization is found, it is removed and pushed to an attribute of the class, if and only if the initialization does not depend on other variables, after that the code statement is changed such that the original variable references the previous initialized class attribute. A cleaning method is also called before usage of the variable if it is a collection which is necessary because the allocation is done when the class object is created and not on every onDraw call.

Just like the RecycleRefactoringRule.java file, the execution starts with the detection of CtClass's that are narrowed down to CtBlocks for processing. The iterateBlock is then called for each of the blocks. However, in this case we filter the methods by the onDraw method signature, such that only onDraw methods are iterated and analysed. Then, the blocks are processed with the phasing algorithm using the life-cycle methods. The detection phase only searches for a single case of interest, which is the ObjectAllocation case of interest. After finding it the transformation phase is skipped and the refactoring phase starts immediately. Then the refactoring phase executes (line 46 of the code listing A.3) and it checks if there is already a class attribute with the same name as the variable that is being assigned the allocation. If it is not present, the attribute is created and the allocation is placed in the attribute's initialization. The reference to the attribute is then assigned to the original statement. Finally, the clear method invocation is also placed before the variable assignment if the variable type is determined to be a collection.

Ideally, methods invoked inside the onDraw method should also be considered, however, this feature is not currently present and will only be created once there is need of it.

3.6.3 WakeLockRefactoringRule.java

Wake locks are mechanisms that prevent the device from hibernating. They are useful when the application needs to be running on the background. Keeping a wake lock for too long will consume a lot of battery energy, therefore, it is important to manage its usage well by releasing them right after they are no longer necessary.

The code listing (A.6) shows the implementation of the wake lock refactoring rule. Just like the other refactoring rules, the process method(line 227) finds CtClass's. It then tries to find the onCreate method (this is the method where most wake locks are created) and iterates over it using the phasing algorithm.

In the detection phase it looks for two cases of interest:

- WakeLockAcquired - A statement where a wake lock is acquired. To know if a wake lock was acquired in the onCreate method.
- VariableDeclared - A statement where a variable is declared.

After collecting the cases of interest the transformation phase is skipped and the refactoring phase starts.

In the refactoring phase (line 50) only WakeLockAcquired cases of interest are analysed. The case is analysed with the following process, first we check if the variable that is used for the acquire call (the acquire method is invoked in a wake lock variable object) is being declared in its scope. If so, then it is a hint that the wake lock is being badly managed since it is in the onCreate method and it does not seem to be used elsewhere. This is not always the case as the variable may be sent somewhere, this,

however, is not taken into consideration for now. If the variable is declared in the scope then its value must be saved in a class attribute for later release of the wake lock. This is done in line 78, first the attribute existence is checked in order not to repeat its declaration. If it does not exist then it is created and an assignment statement is added before the acquire invocation. The management of the wake lock is done through four Android life-cycle methods:

- onCreate - When the activity is first created.
- onPause - When the activity is put on pause.
- onResume - When the activity resumes.
- onDestroy - When the activity is destroyed.

To fix the wake lock problem, the idea is to release the wake lock in the onPause and onDestroy methods and acquire it in the onCreate and onResume methods. For that the code starting in line 111 ensures that all the methods exist, otherwise the methods are created and the release/acquire calls as added where they are necessary.

3.6.4 ViewHolderRefactoringRule.java

Implementations of the getView method from the Android Adapter interface often have inflating method invocations that instantiate a View object from an XML layout file. Doing this without care leads to the recreation of the same View over and over which was shown to increase battery consumption. To avoid this, the convertView parameter of the getView method must be leveraged in order to reuse a previously created view. Not only that, calling findViewById in the inflated view can also lead to battery drainage. To avoid this, all the views that are found should be placed inside of an object called the ViewHolder for later reuse. Again, like the other refactoring rules, the ViewHolder refactoring rule processes CtClass elements. It finds the method of the class with the getView signature and runs the phasing algorithm over it.

In the detection phase there are a couple of cases that need to be detected, they are:

- VariableDeclared - A variable was declared.
- ConvertViewReassignInflator - ConvertView variable was reassigned with a new inflated View instance.
- ConvertViewReuseWithTernary - ConvertView variable was reassigned with a new inflated View instance using a ternary conditional that verifies if the ConvertView already exists (is not null).

- VariableAssignedGetTag - A variable is assigned a value using the getTag method of the ConvertView. The ViewHolder object is saved using the setTag between getView invocations, therefore this case is most likely a retrieval of the ViewHolder instance.
- VariableAssignedFindViewById - A variable is assigned a view using the findViewById method. This can be a potential problem as the code might not be managing the views using the ViewHolder.
- VariableAssignedInflator - A variable received the value of an inflated View. This can mean that the inflation process is being repeated.
- VariableCheckNull - A variable is being checked to see if it is null or not. Null checks may represent reuse of the inflated view or the ViewHolder.

In this refactoring rule the transformation phase is not necessary, so it is skipped. After that the refactoring phase starts in line 230 of the code listing (A.5) in the method onWillRefactor which prepares an extra object with data that will be helpful during the refactorCase execution. After that the refactorCase method (line 83) is invoked for each of the cases of interest one at a time. The first portion of the refactorCase method (line 84 to 176) is dedicated to making sure (by refactoring if necessary) that the ConvertView is being reused and that, if there are call to the findViewById methods, a ViewHolder class exists. The remaining portion of the code deals with the usage of the ViewHolder, it makes sure that the variables are being assigned the values of the views from the ViewHolder and that the ViewHolder attributes are being correctly instantiated using the findViewById method.

3.6.5 Gradle Plugin

The implementation of the Gradle plugin is rather simple because there was no need to use the full classpath mode, meaning that Spoon does not need to know about the target project dependencies. In the code there are some artifacts that deal with this feature, but since the implementation of the full classpath mode was problematic, they were left there to be picked up later. The gradle plugin code is implemented under src/tqrg/leafactor/ci/gradle/plugin. For the no-classpath mode there are 3 files that of particular importance:

- LeafactorPlugin.java - Prepares the plugin. Registers the LauncherExtension and adds the Refactor task to the Gradle tasks list.
- LauncherExtension.java - Used to keep the configurations that will be used in the refactoring task.
- Refactor.java - Has the task implementation, refactors the source code of the application.

One thing worth noting is that if we disregard the refactoring rules (that are specific for Android projects), the skeleton of the project can be used for any Java application that uses Gradle.

The code listing (A.2) shows the portion of the source code of the Refactor.java file that is responsible for the no-classpath mode execution. The processWithoutClassPath method is called after the Gradle plugin is installed and the refactor task is executed with the command gradlew app:refactor. First it constructs a compilation unit group that will hold the references of every Java file found in the project source directory. Then an iteration logger is created for collecting metrics about the execution. After that a list of the refactoring rules is created, all of the four refactoring rules are instantiated and added to the list. The compilationUnitGroup.runInIsolation method is then invoked with the list of refactoring rules as a parameter. This function is responsible for running the Spoon launcher and modifying the Java source files. For each of the Java files, a launcher is prepared with no-classpath mode and a sniper pretty printer (for lexical preservation). Then, if everything is okay the code proceeds to check the configuration. If it is configured to replace the original files, then the output of the Spoon launcher will be the original folder that contains the file, otherwise it will use the configured directory path. The launcher is then run and the control flow will transfer to the Spoon library which will parse the Java file into an AST model and invoke the process methods of the refactoring rules using the visitor pattern.

To publish the plugin, an account was created in the https://plugins.gradle.org/ site, then a configuration was made in the project using the pluginBundle extension in the build.gradle file. Also, it was necessary to add a META-INF.gradle-plugins file in the src/resources folder of the project. After that the project was ready for publishing by login using gradlew login and then publish with gradlew publishPlugins.

3.7 Testing

In order to assert that the cases are being correctly refactored, a testing suit was created. The testing suit is powered by JUnit a very popular unit testing tool. What it does is dynamically look into the src/test/resource folder to find folders with the same name as the refactoring rules classes. If the names match then it looks for its sub-folders to find the tests. Each test folder contains an input and an output file where the input is the file provided and the output is the file with the expected result that should be generated by LeafactorCI when using the input file.

With this setup, adding new tests is easy. Simply add a new folder under the refactoring rule folder and an input and output file inside it.

As of now there are 25 tests distributed between the four refactoring rules. The recycle refactoring is the most tested rule of the four, because it was the first one that was introduced. Since LeafactorCI is based on Leafactor (an early version for the eclipse plugin implemented by Luís Cruz), it was important to support the same tests, they were adapted and added to the testing suit.

3.8 Continuous Integration

Gradle does most of the job in supporting CI. A simple script was created that can be used as reference for integrating LeafactorCI. The script is shown in the code listing (3.1). The script leverages GIT to control and push changes to the repository. It is expected that a cloned GIT repository is in place as it is common for a CI platform to clone the repository at a specific branch or commit. It starts by attributing the user identification since operations will be done by an automatic script. Then the revision number is collected in order to identify the changes that will be made. A new branch is then created and checked out, meaning that any changes will be committed in this particular branch. The branch name also has the revision suffixed. Then the Gradle build process is started, followed by the execution of the refactor operation. Every change is then added to the stage and committed with a simple message. If the remote is wrong, it can be setup using the GIT remote add operation. Finally, the changes are pushed to the remote repository.

Listing 3.1: The CI sample script

```
1  git config user.email "EMAIL_OF_THE_COMMIT_AUTHOR"
2  git config user.name "NAME_OF_THE_COMMIT_AUTHOR"
3  REV=$(git rev-parse --short HEAD)
4  git checkout -b "leafactor-refactoring-$REV"
5  ./gradlew build
6  ./gradlew refactor
7  cd app/src
8  git add .
9  cd ../../
10 git commit --allow-empty -m "LeafactorCI refactoring changes."
11 git remote rm origin
12 git remote add origin "GIT_REPOSITORY_URL"
13 git push origin "leafactor-refactoring-$REV"
```

4

Evaluation

Contents

The previous chapters presented the architecture of LeafactorCI and how it is designed to be easy to use in CI environments. In this chapter I will be describing how LeafactorCI was evaluated.

4.1 User study

To evaluate the difficulty of adoption a small user study was conducted, composed of 2 surveys and a hands-on installation trial.

To dissimulate the understanding about the usage of energy practices and publicize LeafactorCI, it was devised a short and informational survey and published it in the GitHub software community. The survey was composed of the following questions along with related information:

- What is the role that best describes you?
- Have you worked in any Android mobile applications?
- Have you ever heard about energy bugs (bugs in the code that lead to more energy consumption), are they a concern to you?
- Would you like to hear more about what they are?
- There is a new free and open-source tool called LeafactorCI that just came out in alpha stage that can detect and refactor the previously mentioned anti-patterns and can even be integrated into a CI environment, would you be willing to learn more about it?
- LeafactorCI is very easy to setup. Would you be willing to try LeafactorCI?

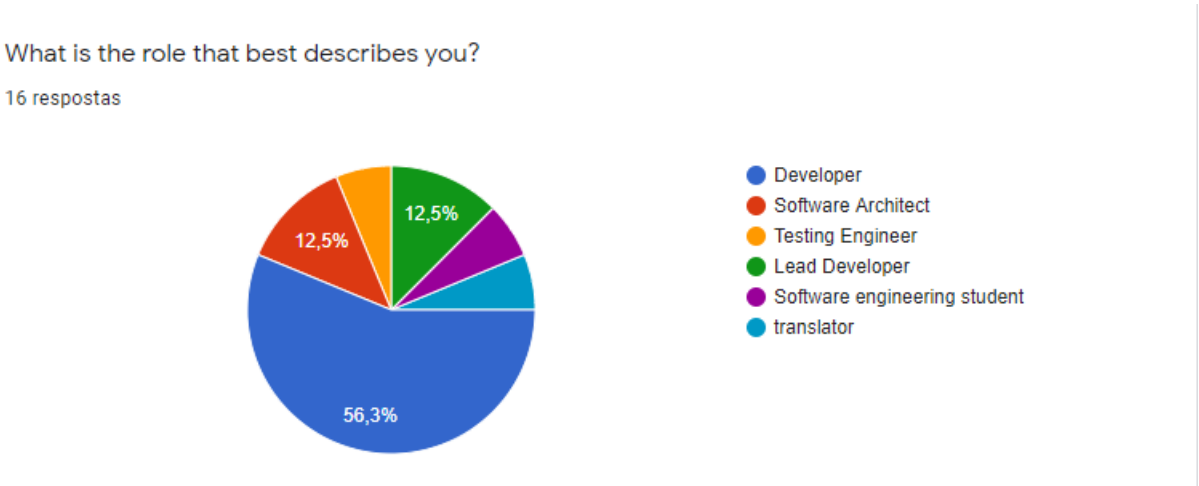

Figure 4.1: Distribution of the participant roles.

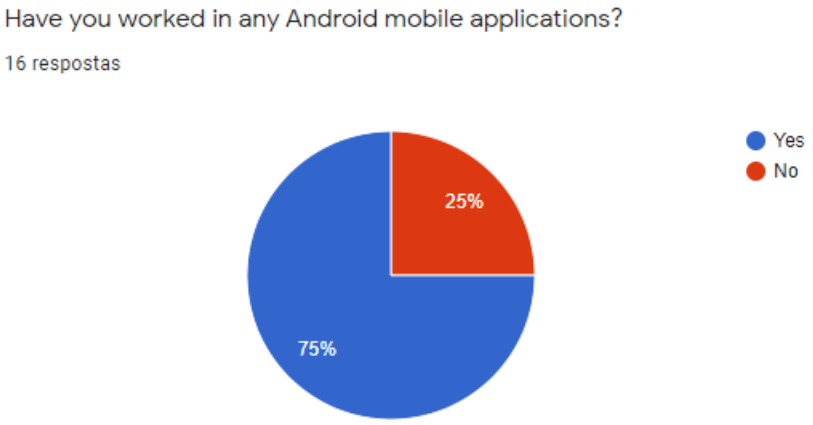

Figure 4.2: Distribution of participants that worked on Android app previously.

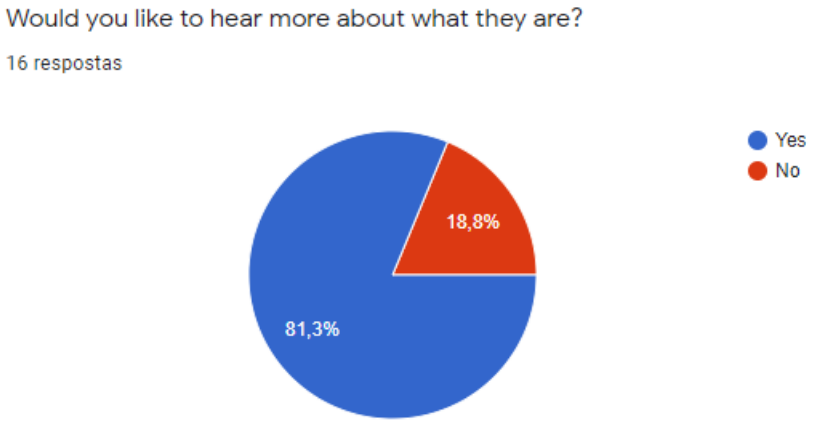

Figure 4.3: Distribution of participant's willingness for hearing more about the anti-patterns.

The questionnaire was answered by 16 different people. 56.3% described themselves as developers, 12.5% of them as Software Architects, 12.5% as Lead Developers the rest of them described themselves as other roles related to application development (Figure 4.1). 75.0% of them had worked on Android mobile applications while the rest did not (Figure 4.2). 56.2% have heard about energy bugs, at least 37.5% say they are a concern to them (there were ambiguous answers that were not considered as a definite yes). Only 31.3% knew about at least one of the 4 patterns (Recycle, View Holder, Draw Allocation and Wake Lock). 82.3% wanted to know more about the patterns (Figure 4.3). After briefly being introduced to LeafactorCI, 78.6% said that they were willing to know more about it (Figure 4.4), however, only 63.6% said that they were willing to try it (Figure 4.5).

While the sample is small, the results suggest that energy practices are not disseminated enough through the community and that the community is willing to try LeafactorCI.

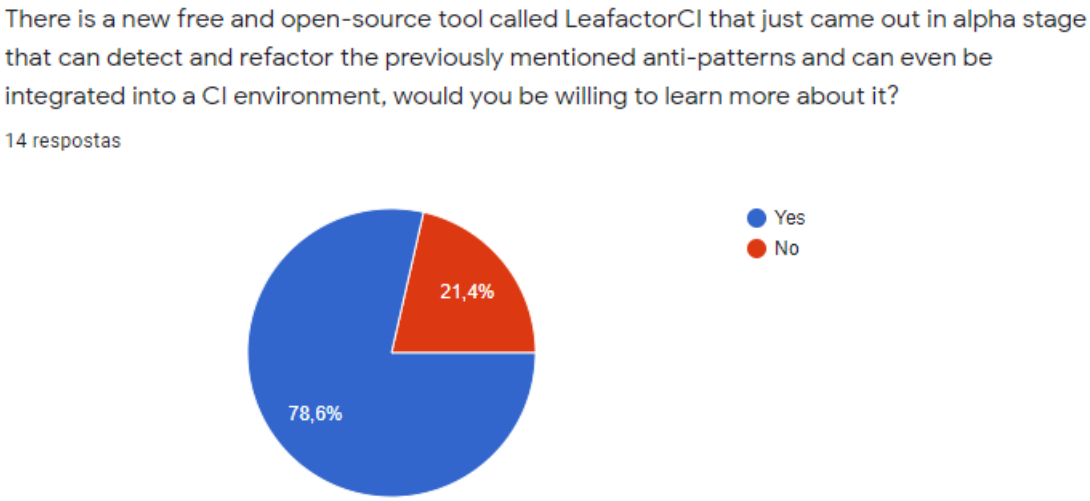

Figure 4.4: Distribution of participant's willingness for hearing more about the LeafactorCI.

A small experiment was also conducted with three developers. The process was as follows, they had to choose a couple of Android open-source repositories at random with more than 300 commits, then they had to try to install LeafactorCI on each of them to see how easy it would be. They were asked to first select the repositories (at least three repositories per person) and then to fork them. After that they were to make the installation and commit the changes back to the forked repository. Then, they were asked to run the LeafactorCI tool and if any changes were to occur, they were to be committed to the forked repositories as well. At the end they were asked to fill in a survey with the following questions:

- Were you able to install LeafactorCI?
- How difficult was the installation process? Leave empty if you were not able to set it up. (1 - 5).
- Did you need to troubleshoot while setting up LeafactorCI? Leave empty if you were not able to set it up.
- Were you able to run the refactoring task?
- Did LeafactorCI correct any problems in your application? Leave empty if you were unable to run it.
- Did you find LeafactorCI useful? Do you see potential in it?

The three participants were able to make the installation (Figure 4.6). Two of the participants reported that the difficulty scoring was a one out of five and one participant reported that it was a two out of five (Figure 4.7). Two of the participants had to troubleshoot (Figure 4.8). Every single one of the participants

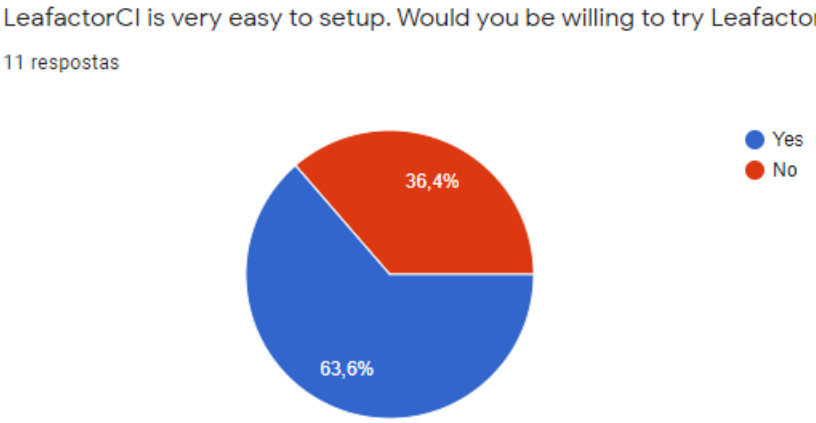

Figure 4.5: Distribution of participant's willingness for trying LeafactorCI.

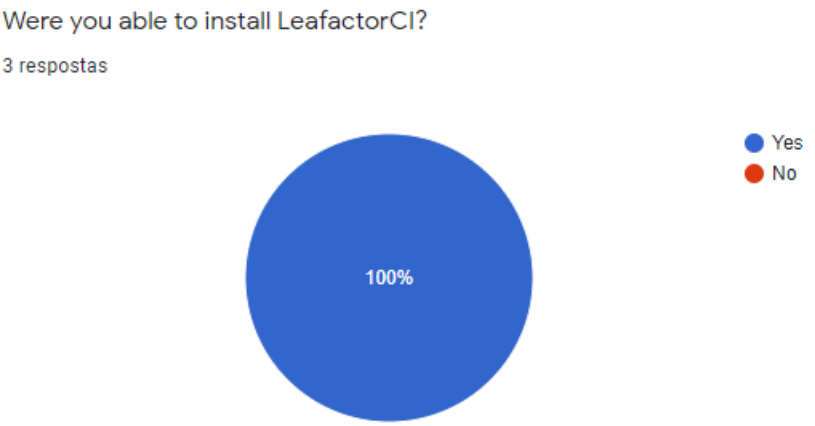

Figure 4.6: Distribution of participant's ability in installing LeafactorCI.

was able to run the refactoring task (Figure 4.9). Two of the participants found at least one of the anti-patterns in at least one of the repositories. Finally, all the participants found the tool useful and with potential.

In total 11 arbitrary repositories[1] were tried. From what I could gather, LeafactorCI did not detect energy bugs in most of the repositories, only two repositories out of the 11 were found to have energy bugs. Also, there were one or two repositories where problems were found that prevented the execution of the refactoring task due to bugs in the Spoon library. One of which led to an open issue. Other problems were related to Gradle version incompatibilities that were easily overcome. The installation procedure was at most times easy and without problems. It is also important to delineate that the participants had no prior experience with the repositories that they had chosen, and yet they were able

[1] Whose forks can be found in https://gist.github.com/moraispgsi/bc3eca2f92d3151bc85ac86f2248078e

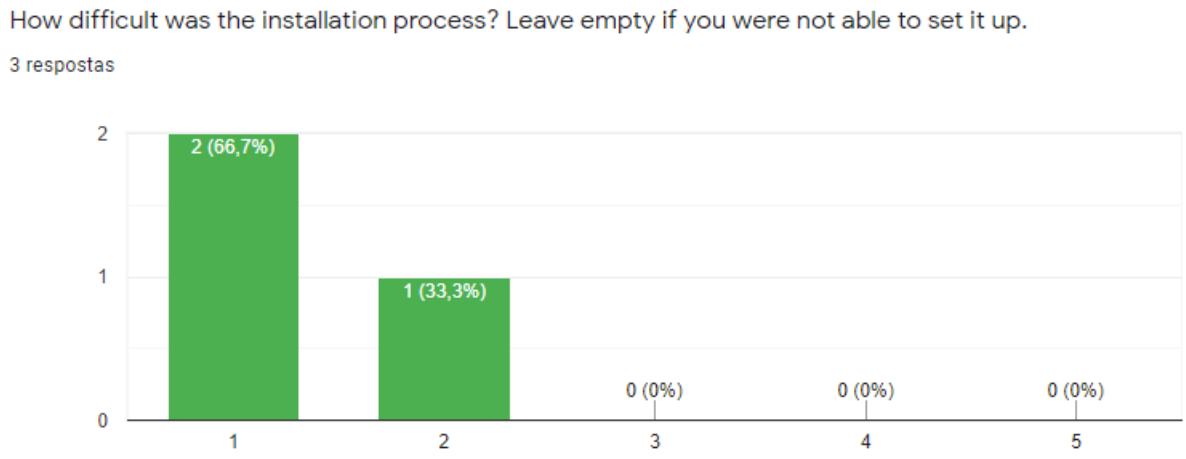

Figure 4.7: Distribution of participant's difficulty in installing LeafactorCI.

to install and run LeafactorCI, this fact provides a substantial indication of its ease of use.

Outside of this user-study other people tried and successfully used LeafactorCI to see if there were problems in their project, to which, no energy bugs were found.

4.2 Can LeafactorCI be used inside a CI environment?

Before addressing the main question, let us consider a sub-question. Does LeafactorCI work in a normal environment? Yes, and the way this was guaranteed was by mean of introducing a test battery that given an input file executes LeafactorCI over it and check if the output corresponds to the optimal output file. There are 25 tests each, in most cases, with more than one variation of the anti-patterns present. This guarantees that LeafactorCI is able to support the set of conditions present in the input files which account for the most common usage. In all of the testing cases the code semantics were guaranteed. For other use cases that this test battery does not cover, the guarantee for maintaining the code semantics falls back to the reviewer (which in a collaborative environment would be called the project integrator), along with the possibility of false positives.

In order to be used, LeafactorCI needs to be published and be readily available. During this dissertation, LeafactorCI was released in alpha stage (as a Gradle plugin in the Gradle plugins repository at `https://plugins.gradle.org/plugin/tqrg.leafactor.ci`) and can be found at `https://github.com/TQRG/leafactor-ci`. In the README.md of the LeafactorCI repository is the instructions for the installation along with a Frequently Asked Questions (FAQ) section. In the same README.md file is present the instructions to execute and publish LeafactorCI as a contributor. Finally, there is a section of

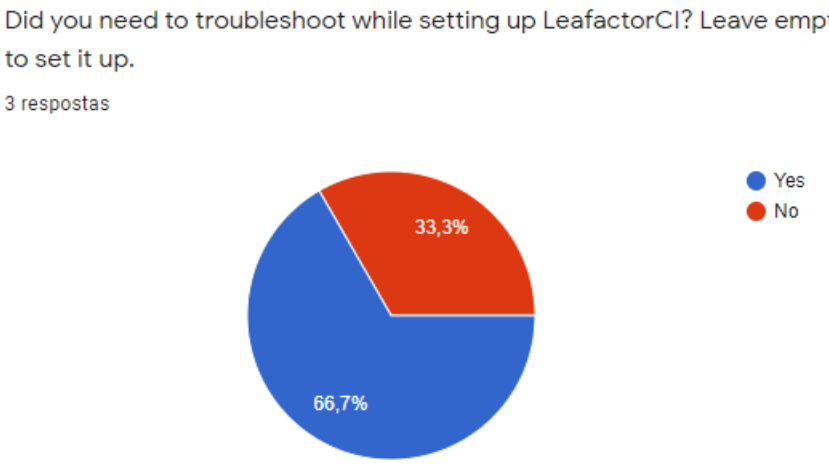

Figure 4.8: Distribution of participant's necessity for troubleshooting during the LeafactorCI installation.

known issues and their respective states of resolution.

Now, to show that LeafactorCI can be setup in a CI environment, during this dissertation, a fork was made from an open-source Android project called Slide (the forked repository is, at present time, at https://github.com/TQRG/Slide). Then, the LeafactorCI plugin installation was made and the travis.yml file was modified(which is the file that is used to configure the CI pipeline in TravisCI), the changes can be found in `https://github.com/TQRG/Slide/commit/f063e548bd2f770bde96b401096236ae6b8cf3af` along with some other unrelated changes that were necessary to bring the project back to more modern versions. The changes were fairly easy to make. It was set up such that whenever a commit is done to the code-base, a new branch is created and LeafactorCI is executed. A pull request could then be created in order to decide the branch's merge-ability.

4.3 How easy it is to adopt LeafactorCI?

LeafactorCI is meant to be easy to adopt since it leverages the same platform that the Android project is built upon, Gradle. The installation process is very easy, excluding some hiccups that may happen it can be as simple as adding a line of code inside a file(build.gradle file). Of-course due to the differences between every Android project, such as its own setup and its version dependencies there might be some inconveniences to be overcome. As of now, it have yet to be compiled a list of system requirements and supported versions of Gradle and the Android Software Development Kit (SDK). By publishing an alpha version those problems will become more evident and troubleshooting instructions will be added incrementally to the LeafactorCI project documentation. Adopting LeafactorCI right now comes with the problems of any new software project, it has bugs, it has the bare minimum options and there is no

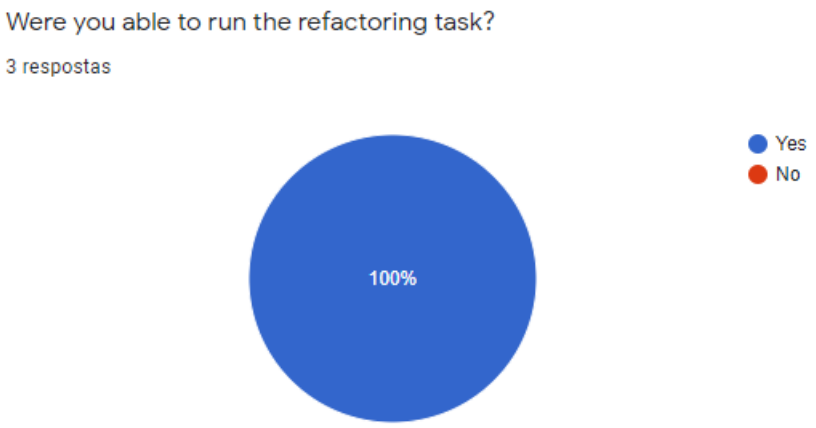

Figure 4.9: Distribution of participant's ability for executing LeafactorCI.

community support. Early adopters have to take this into consideration and look past to see its potential. A major advantage of LeafactorCI is its open-source nature, anyone finding difficulties can open an issue or even contribute to the source-code.

In terms of adopting LeafactorCI in a CI environment, some questions should be place on the developers:

- Do I need LeafactorCI? A project with very few changes over time might not be a good candidate for using LeafactorCI in CI.

- What should trigger the refactoring process? Should it be a commit in the development branch, or a commit in another specific branch, should it be when opening a pull-request, those options should be considered.

- How often should the refactoring process happen? This should account for the number of changes that are made over time in the project. More changes lead to more possibilities for anti-patterns to form.

- What to do with the changes? Should a branch be created or should another way be used to evaluate the changes that were made. Like e.g. sending an informative e-mail which then can be used as reference for a manual commit.

The adoption can be as difficult as the developers want, it depends on the use case.

4.4 Performance

When Gradle was adopted, it was a choice between performance and ease of use. Creating a CLI instead of a Gradle plugin would run much faster as there would be no need to build the entire project before refactoring. However, minding the long run, eventually refactoring rules might need to use full classpath mode, meaning that it would need to find all the projects dependencies. Therefore, a CLI would turn out to be inferior because it would be difficult to extract the dependencies, which in contrast is fairly manageable in Gradle. So, there is a great cost in performance when executing LeafactorCI due to the Gradle build task. The execution of LeafactorCI mainly depends on the speed of the build of the Android project itself. The speed of the actual refactoring turns out to be insignificant when compared to the speed of the build, such that it is not worth measuring.

5

Conclusion

Contents

In the course of this work, LeafactorCI was exposed as a solution to a problem in android applications energy consumption. The first chapter (chapter 1) was dedicated to defining the problem at hand and the objectives delineated for this work to answer it in a general sense. In chapter 2 an overview of the related work defines the current contributions to help solving this problem, the related studies results which better define the problem and evaluates solutions. Afterwards, in chapter 3 the overall solution is defined establishing the design decisions for the development of LeafactorCI. Subsequently, the final solution's implementation details were shown. Finally, in chapter 4 an evaluation of resulting solution, along with subjects and feedback was presented.

5.1 Motivation

This dissertation started with a problem and an opportunity. The problem was the lack of availability of tools and means for fixing energy bugs in the Android application development community, which may lead to bad application reviews and consequently less sales. As for the opportunity, it was the rise of CI, the increasing adoption of CI practices and usage of CI services that is improving the way that we integrate software.

5.2 Contributions

This work adopted a method of refactoring 4 distinct anti-patterns (Wake Lock, View Holder, Recycle, and Draw Allocation) through static analysis of the source-code of Android Java projects in order to improve the energy efficiency of Android applications, furthermore, a CI solution was designed. The solution was composed of a refactoring tool called LeafactorCI and a strategy for integrating it inside a CI environment. The design decisions were driven by the current and rising practices of Android application development, which include the usage of GIT, Gradle and CI platforms that use containerization technology. The tool was evaluated by means of a user study to discern its usability, and the data suggest that it falls in the easy to use category.

LeafactorCI's current stage solution is very promising having acquired versatility, adaptability and functional potential, which already surpasses its predecessor (Leafactor on AutoRefactor). The user no longer needs to be constrained to a specific IDE to get rid of the energy bugs of his project. The source-code is available for any contributor who would like to advance this technology at `https://github.com/TQRG/leafactor-ci`. For developers looking to improve LeafactorCI, they are provided with a simple way to add new refactoring rules and test them without the added risk of tempering with the existing functionality. The current version of LeafactorCI is alpha and there is a long road ahead until it becomes a stable and widely used solution.

The architecture was designed to adapt to the project's team specific workflow. This work provided a strategy to integrate LeafactorCI in a CI environment. There are of course many other alternative strategies that can be adopted and the responsibility falls on the project developers to make sure that the one adopted is a success.

5.3 System Limitations and Future Work

LeafactorCI is new, containing some limitations. The first limitation and most important one right now is the lexical preservation. Right now the refactoring library Spoon, the library that LeafactorCI is based upon, has a volatile support for this feature, which means that LeafactorCI is at the mercy of the Spoon community when it comes to providing clean refactorings with a small change footprint. During the development of LeafactorCI many were the times where issues arose in this functionality, and there are still some issues to be solved on their end. Hopefully, the Spoon team will continue to improve this feature.

Another limitation is that there are no clear definition of the versions that are supported. A developer intending to use it does not know if his project is supported by LeafactorCI. This limitation can be resolved by testing a big variety of project versions and Gradle versions to see where it breaks and, for starters, solving the problems when possible for, at least, the most common versions.

There are still some polishing to be done to the refactoring rules. They only account for the simplest of cases. More work has to be done to support more complex cases such as when there are branches in the control flow and more complex statements such as using lambda functions with closures.

The fact that there are only 4 refactoring rules so far makes it harder to adopt. If more refactoring rules were created, the more value this would have and therefore it would make it more appealing to adopting.

These limitations should be considered when using LeafactorCI. Thus, its early adoption should account with the will to be critical and contributive to this project with a lot of potential to help us do better with our applications.

Bibliography

[1] Google code, 2020. URL https://code.google.com/. Visited in May of 2020.

[2] Spoon history, 2020. URL https://github.com/INRIA/spoon/issues/2741. Visited in May of 2020.

[3] Git popularity, 2020. URL https://rhodecode.com/insights/version-control-systems-2016. Visited in May of 2020.

[4] Javaparser, 2020. URL https://github.com/javaparser/javaparser. Visited in May of 2020.

[5] Spoon, 2020. URL https://github.com/INRIA/spoon. Visited in May of 2020.

[6] Spoon vs javaparser, 2020. URL https://github.com/INRIA/spoon/issues/1303. Visited in May of 2020.

[7] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Feng Xia, and Muhammad Shiraz. A Review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues. *Journal of Network and Computer Applications*, 58:42–59, 2015. ISSN 10958592. doi: 10.1016/j.jnca.2015.09.002.

[8] Abhijeet Banerjee and Abhik Roychoudhury. Automated re-factoring of Android apps to enhance energy-efficiency. *Proceedings of the International Workshop on Mobile Software Engineering and Systems - MOBILESoft '16*, pages 139–150, 2016. doi: 10.1145/2897073.2897086. URL http://dl.acm.org/citation.cfm?doid=2897073.2897086.

[9] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 588–598, 2014. doi: 10.1145/2635868.2635871. URL http://dl.acm.org/citation.cfm?doid=2635868.2635871.

[10] Luis Cruz and Rui Abreu. Performance-Based Guidelines for Energy Efficient Mobile Applications. *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MO-*

BILESoft), pages 46–57, 2017. doi: 10.1109/MOBILESoft.2017.19. URL http://ieeexplore.ieee.org/document/7972717/.

[11] Luis Cruz and Rui Abreu. Using automatic refactoring to improve energy efficiency of android apps. *CoRR*, abs/1803.05889, 2018. URL http://dblp.uni-trier.de/db/journals/corr/corr1803.html#abs-1803-05889.

[12] Luis Cruz and Rui Abreu. EMaaS: Energy measurements as a service for mobile applications. *IEEE/ACM 41st International Conference on Software Engineering*, 2019. doi: 10.1007/s10664-019-09701-0.

[13] Luis Cruz, Rui Abreu, and Jean Noel Rouvignac. Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. *Proceedings - 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2017*, 2017. doi: 10.1109/MOBILESoft.2017.21. URL http://www.mendeley.com/research/leafactor-improving-energy-efficiency-android-apps-via-automatic-refactoring.

[14] Luis Cruz, Rui Abreu, and David Lo. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering*, 24(4):2438–2468, 2019. URL http://dblp.uni-trier.de/db/journals/ese/ese24.html#CruzAL19.

[15] Luís Cruz. Tools and Techniques for Energy-Efficient Mobile Application Development, PhD thesis. 2019.

[16] Luís Cruz and Rui Abreu. Catalog of energy patterns for mobile applications. *Empirical Software Engineering*, 03 2019. doi: 10.1007/s10664-019-09682-0.

[17] Andre Luiz Tinassi Damato, Linnyer Beatrys Ruiz, Anderson Faustino Da Silva, and Jose Camargo Da Costa. EProf: An accurate energy consumption estimation tool. *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, pages 210–218, 2012. ISSN 15224902. doi: 10.1109/SCCC.2011.28.

[18] Ralph Johnson Richard Helm Erich Gamma, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

[19] Martin Fowler. 2006. URL https://martinfowler.com/articles/continuousIntegration.htm.

[20] Martin Fowler and Kent Beck. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1 edition, July 2013. ISBN 0201485672. URL http://martinfowler.com/books/refactoring.html.

[21] Mark Gordon, Lide Zhang, and Birjodh Tiwana. Powertutor, 2020. URL https://github.com/msg555/PowerTutor. Visited in May of 2020.

[22] Marion Gottschalk. Energy Refactorings. *Master Thesis, http://www.se.uni-oldenburg.de/documents/gottschalk-MA2013.pdf*, 2013.

[23] Shuai Hao, Ding Li, William G.J. Halfond, and Ramesh Govindan. Estimating Android applications' CPU energy usage via bytecode profiling. *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings*, pages 1–7, 2012. ISSN 1050-2947. doi: 10.1109/GREENS.2012.6224263.

[24] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 426–437, 2016. doi: 10.1145/2970276.2970358. URL http://dl.acm.org/citation.cfm?doid=2970276.2970358.

[25] Ding Li and William G. J. Halfond. An investigation into energy-saving programming practices for Android smartphone app development. *Proceedings of the 3rd International Workshop on Green and Sustainable Software - GREENS 2014*, pages 46–53, 2014. doi: 10.1145/2593743.2593750. URL http://dl.acm.org/citation.cfm?doid=2593743.2593750.

[26] Ding Li, Shuai Hao, Jiaping Gui, and William G.J. Halfond. An empirical study of the energy consumption of android applications. *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, pages 121–130, 2014. ISSN 1063-6773. doi: 10.1109/ICSME.2014.34.

[27] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: an empirical study. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 2–11, 2014. doi: 10.1145/2597073.2597085. URL http://dl.acm.org/citation.cfm?doid=2597073.2597085.

[28] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M Almurshed, and Carlos Jensen. Understanding Code Smells in Android Applications. *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, page 2997, 2016. doi: 10.1109/MobileSoft.2016.048.

[29] Rodrigo Morales, Ruben Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. *IEEE Transactions on Software Engineering*, 2017. ISSN 00985589. doi: 10.1109/TSE.2017.2757486.

[30] Danny van Bruggen Nicholas Smith and Federico Tomassetti. *JavaParser: Visisted*. 2019. URL https://leanpub.com/javaparservisited.

[31] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, (June):1–13, 2018. ISSN 09505849. doi: 10.1016/j.infsof.2018.08.004. URL https://linkinghub.elsevier.com/retrieve/pii/S0950584918301678.

[32] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In Pascal Felber, Frank Bellosa, and Herbert Bos, editors, *EuroSys*, pages 29–42. ACM, 2012. ISBN 978-1-4503-1223-3. URL http://dblp.uni-trier.de/db/conf/eurosys/eurosys2012.html#PathakHZ12.

[33] Jan Reimann, Martin Brylski, and Uwe Aßmann. A tool-supported quality smell catalogue for android developers. *Softwaretechnik-Trends*, 34(2), 2014. URL http://dblp.uni-trier.de/db/journals/stt/stt34.html#ReimannBA14.

[34] Jean-Noël Rouvignac. Autorefactor, 2020. URL https://github.com/JnRouvignac/AutoRefactor. Visited in May of 2020.

[35] Cagri Sahin, Furkan Cayci, Irene Lizeth Manotas Gutiérrez, James Clause, Fouad Kiamilev, Lori Pollock, and Kristina Winbladh. Initial explorations on design pattern energy usage. *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings*, pages 55–61, 2012. doi: 10.1109/GREENS.2012.6224257.

[36] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5:3909–3943, 2017. ISSN 21693536. doi: 10.1109/ACCESS.2017.2685629.

[37] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87(1):48–59, 2014. ISSN 01641212. doi: 10.1016/j.jss.2013.08.032. URL http://dx.doi.org/10.1016/j.jss.2013.08.032.

[38] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 805–816, 2015. doi: 10.1145/2786805.2786850. URL http://dl.acm.org/citation.cfm?doid=2786805.2786850.

[39] Roberto Verdecchia, René Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. Empirical evaluation of the energy impact of refactoring code smells. In Birgit Penzenstadler, Steve Easter-

brook, Colin C. Venters, and Syed Ishtiaque Ahmed, editors, *ICT4S*, volume 52 of *EPiC Series in Computing*, pages 365–383. EasyChair, 2018. URL http://dblp.uni-trier.de/db/conf/ict4s/ict4s2018.html#VerdecchiaSPL18.

[40] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *ASE*, pages 60–71. IEEE Computer Society, 2017. ISBN 978-1-5386-2684-9. URL http://dblp.uni-trier.de/db/conf/kbse/ase2017.html#ZhaoSZFV17.

A

Project Code

This section shows only the most important source files of the project. The files were trimmed out in order to abide to the sizing constraints.

Listing A.1: The Iterable interface

```
1  ...
2  public interface Iterable {
3      ...
4      static void iterateBlock(
5              RefactoringRule rule,
6              IterationLogger logger,
7              CtBlock block,
8              boolean isDeep,
9              int depth
10     ) {
11         SimpleIterationPhaseLogEntry setupLogEntry = new SimpleIterationPhaseLogEntry(rule, "Setting up iteration", "Logs the setup phase of an
               iteration");
12
13         // SETUP PHASE
14         setupLogEntry.start();
15         SimpleIterationPhaseLogEntry detectionPhaseLogEntry = new SimpleIterationPhaseLogEntry(rule, "Detecting Patterns", "Logs the detection
               phase of an iteration");
16         SimpleIterationPhaseLogEntry transformationPhaseLogEntry = new SimpleIterationPhaseLogEntry(rule, "Transforming Cases of Interest", "Logs
               the transformation phase of an iteration");
17         SimpleIterationPhaseLogEntry refactoringPhaseLogEntry = new SimpleIterationPhaseLogEntry(rule, "Refactoring Cases of Interest", "Logs the
               processing phase of an iteration");
```

```
18          DetectionPhaseContext detectionPhaseContext = new DetectionPhaseContext();
19          detectionPhaseContext.block = block;
20          rule.onSetup(detectionPhaseContext);
21          setupLogEntry.stop();
22          // END SETUP PHASE
23
24          // DETECTION PHASE
25          detectionPhaseLogEntry.start();
26          for (int i = 0; i < block.getStatements().size(); i++) {
27              detectionPhaseContext.statement = block.getStatements().get(i);
28              detectionPhaseContext.statementIndex = i;
29              rule.onWillIterate(detectionPhaseContext);
30              if (isDeep && detectionPhaseContext.statement instanceof CtBlock) {
31                  CtBlock statementBlock = (CtBlock) detectionPhaseContext.statement;
32  //                Iteration.iterateBlock(rule, logger, statementBlock, true, depth + 1);
33  //                Todo: do something with the innerContext
34              }
35              rule.detectCase(detectionPhaseContext);
36              rule.onDidIterate(detectionPhaseContext);
37          }
38          detectionPhaseLogEntry.stop();
39          // END DETECTION PHASE
40
41          // TRANSFORMATION PHASE
42          transformationPhaseLogEntry.start();
43          List<CaseOfInterest> copyCasesDetected = new ArrayList<>(detectionPhaseContext.caseOfInterestList);
44          TransformationPhaseContext transformationPhaseContext = new TransformationPhaseContext();
45          transformationPhaseContext.block = block;
46          transformationPhaseContext.caseOfInterestList = copyCasesDetected;
47          rule.onWillTransform(transformationPhaseContext);
48          for (CaseOfInterest caseOfInterest : copyCasesDetected) {
49              transformationPhaseContext.caseOfInterest = caseOfInterest;
50              rule.onWillTransformCase(transformationPhaseContext);
51              rule.transformCase(transformationPhaseContext);
52              rule.onDidTransformCase(transformationPhaseContext);
53          }
54          transformationPhaseLogEntry.stop();
55          // END TRANSFORMATION PHASE
56
57          // REFACTORING PHASE
58          refactoringPhaseLogEntry.start();
59          List<CaseOfInterest> copyCasesFiltered = transformationPhaseContext.getResult();
60          RefactoringPhaseContext refactoringPhaseContext = new RefactoringPhaseContext();
61          refactoringPhaseContext.offset = 0;
62          refactoringPhaseContext.block = block;
63          refactoringPhaseContext.casesOfInterest = copyCasesFiltered;
64          rule.onWillRefactor(refactoringPhaseContext);
65          for (CaseOfInterest caseOfInterest : copyCasesFiltered) {
66              refactoringPhaseContext.caseOfInterest = caseOfInterest;
67              rule.onWillRefactorCase(refactoringPhaseContext);
68              rule.refactorCase(refactoringPhaseContext);
69              rule.onDidRefactorCase(refactoringPhaseContext);
70          }
71          refactoringPhaseLogEntry.stop();
72          // END REFACTORING PHASE
73
74          logger.getLogs().add(setupLogEntry);
75          logger.getLogs().add(detectionPhaseLogEntry);
76          logger.getLogs().add(transformationPhaseLogEntry);
77          logger.getLogs().add(refactoringPhaseLogEntry);
78      }
79  }
```

Listing A.2: The Refactor class

```
1   ...
```

```
2  public class Refactor extends DefaultTask {
3      private Project project;
4      private LauncherExtension launcherExtension;
5
6      void init(Project project, LauncherExtension launcherExtension) {
7          this.project = project;
8          this.launcherExtension = launcherExtension;
9      }
10
11     private AppExtension getAppExtension() {
12         // Check if the Android AppPlugin is present
13         if (!project.getPlugins().hasPlugin(AppPlugin.class)) {
14             throw new RuntimeException("should be declared after 'com.android.application'");
15         }
16         // Get the AppExtension from the gradle runtime
17         AppExtension appExtension = project.getExtensions().findByType(AppExtension.class);
18         assert appExtension != null;
19         return appExtension;
20     }
21     ...
22     private void processWithoutClassPath() throws IOException {
23         String projectPath = project.getProjectDir().toPath().toString();
24         String sourcePath = Paths.get(projectPath, "src", "main", "java").toString();
25
26         CompilationUnitGroup compilationUnitGroup = new CompilationUnitGroup(null);
27         setupOutputDirectory(compilationUnitGroup, "main");
28         compilationUnitGroup.add(new File(sourcePath));
29
30         IterationLogger logger = new IterationLogger();
31         List<RefactoringRule> refactoringRules = new ArrayList<>();
32
33         // Adding all the refactoring rules
34         refactoringRules.add(new RecycleRefactoringRule(logger));
35         refactoringRules.add(new ViewHolderRefactoringRule(logger));
36         refactoringRules.add(new DrawAllocationRefactoringRule(logger));
37         refactoringRules.add(new WakeLockRefactoringRule(logger));
38
39         // Run the group of compilation units with the set of refactoring rules
40         compilationUnitGroup.runInIsolation(refactoringRules);
41     }
42
43     @TaskAction
44     public void task() throws IOException {
45         AppExtension appExtension = getAppExtension();
46         if (launcherExtension.isUsingClasspath()) {
47             DependenciesManager dependenciesManager = new DependenciesManager(appExtension, project);
48             iterateOverApplicationVariants(appExtension, dependenciesManager);
49         } else {
50             processWithoutClassPath();
51         }
52     }
53 }
```

Listing A.3: The DrawAllocationRefactoringRule class

```
1  ...
2  public class DrawAllocationRefactoringRule extends AbstractProcessor<CtClass> implements RefactoringRule<CtClass> {
3      private IterationLogger logger;
4
5      public DrawAllocationRefactoringRule(IterationLogger logger) {
6          this.logger = logger;
7      }
8
9      private boolean methodSignatureMatches(CtMethod method) {
10         // SIGNATURE:
11         // public void onDraw(Canvas canvas)
```

```
12          boolean nameMatch = method.getSimpleName().equals("onDraw");
13          CtTypeReference type = method.getType();
14          boolean returnTypeMatch = type != null && type.getSimpleName().equals("void");
15
16          boolean hasSameNumberOfArguments = method.getParameters().size() == 1;
17          if (hasSameNumberOfArguments) {
18              List parameterList = method.getParameters();
19
20              CtTypeReference firstArgumentType = ((CtParameter) parameterList.get(0)).getType();
21              boolean firstArgumentTypeMatches = firstArgumentType.getSimpleName().endsWith("Canvas");
22
23              return nameMatch &&
24                      returnTypeMatch &&
25                      firstArgumentTypeMatches;
26          }
27
28          return false;
29      }
30
31      @Override
32      public void detectCase(DetectionPhaseContext context) {
33          // Detect object allocations
34          ObjectAllocation objectAllocation = ObjectAllocation.detect(context);
35          if (objectAllocation != null) {
36              context.caseOfInterestList.add(objectAllocation);
37          }
38      }
39
40      @Override
41      public void transformCase(TransformationPhaseContext context) {
42          CaseTransformer.createPassThroughTransformation().transformCase(context);
43      }
44
45      @Override
46      public void refactorCase(RefactoringPhaseContext context) {
47          if (context.caseOfInterest instanceof ObjectAllocation) {
48              ObjectAllocation objectAllocation = (ObjectAllocation) context.caseOfInterest;
49              if (objectAllocation.getStatement() instanceof CtVariable) {
50                  // We are declaring a variable, pull the declaration out of the scope.
51                  CtClass ctClass = RefactoringRule.getClosestClassParent(context.block);
52                  if (ctClass == null) {
53                      return;
54                  }
55                  // Check if the field is inside, create it if necessary
56                  List<CtField<?>> fields = ctClass.getFields();
57                  Optional<CtField<?>> optionalField = fields.stream().filter(field -> field.getSimpleName()
58                          .equals(objectAllocation.variable.getSimpleName())).findFirst();
59                  if (optionalField.isPresent() && !optionalField.get().getType().getSimpleName()
60                          .equals(objectAllocation.variable.getType().getSimpleName())) {
61                      // If types do not match we ignore for now.
62                      return;
63                  }
64                  if (!optionalField.isPresent()) {
65                      CtTypeReference typeReference = objectAllocation.variable.getType();
66                      CtField field = ctClass.getFactory().createCtField(objectAllocation.variable.getSimpleName(), typeReference,
67                              objectAllocation.constructorCall.toString(), ModifierKind.PRIVATE);
68                      ctClass.addField(field);
69
70                      if (ObjectAllocation.isClearable(typeReference)) {
71                          objectAllocation.getStatement().insertBefore(
72                                  ctClass.getFactory().createCodeSnippetStatement(
73                                          objectAllocation.variable.getSimpleName() + ".clear()"));
74                      }
75                      context.block.removeStatement(objectAllocation.getStatement());
76                  }
77              }
78          }
79      }
80
81      private void refactor(CtMethod method) {
```

```
82          if (!methodSignatureMatches(method)) {
83              return;
84          }
85          List<CtBlock> blocks = RefactoringRule.getCtElementsOfInterest(method, CtBlock.class::isInstance, CtBlock.class);
86          for (CtBlock block : blocks) {
87              Iterable.iterateBlock(this, logger, block, false, 0);
88          }
89      }
90
91      public void process(CtClass element) {
92          Set methods = element.getMethods();
93          for (Object method : methods) {
94              if (method instanceof CtMethod) {
95                  refactor((CtMethod) method);
96              }
97          }
98      }
99  ...
100 }
```

Listing A.4: The RecycleRefactoringRule class

```
1   ...
2   public class RecycleRefactoringRule extends AbstractProcessor<CtClass> implements RefactoringRule<CtClass> {
3       // List of classes that need to be recycled
4       private Map<String, String> opportunities = new LinkedHashMap<>();
5       private IterationLogger logger;
6
7       public RecycleRefactoringRule(IterationLogger logger) {
8           this.logger = logger;
9           // todo - Should detect with the full namespace.
10          opportunities.put("TypedArray", "recycle");
11          opportunities.put("Bitmap", "recycle");
12          opportunities.put("Cursor", "close");
13          opportunities.put("VelocityTracker", "recycle");
14          opportunities.put("Message", "recycle");
15          opportunities.put("MotionEvent", "recycle");
16          opportunities.put("Parcel", "recycle");
17          opportunities.put("ContentProviderClient", "release");
18      }
19
20      @Override
21      public void detectCase(DetectionPhaseContext context) {
22          // Detect variables declared
23          VariableDeclared variableDeclared = VariableDeclared.detect(context);
24          if (variableDeclared != null) {
25              String typeName = variableDeclared.variable.getType().getSimpleName();
26              if (opportunities.containsKey(typeName)) {
27                  context.caseOfInterestList.add(variableDeclared);
28              }
29          }
30          // Detect variables reassigned
31          VariableReassigned variableReassigned = VariableReassigned.detect(context);
32          if (variableReassigned != null) {
33              CtExpression lhs = variableReassigned.assignment.getAssigned();
34              if (lhs instanceof CtVariableWrite) {
35                  context.caseOfInterestList.add(variableReassigned);
36              }
37          }
38          // Detect variables usage
39          VariableUsed variableUsed = VariableUsed.detect(context);
40          if (variableUsed != null) {
41              context.caseOfInterestList.add(variableUsed);
42          }
43          // Detect lost variables
44          VariableLost variableLost = VariableLost.detect(context);
```

```
45          if (variableLost != null) {
46              context.caseOfInterestList.add(variableLost);
47          }
48
49          // Detect recycled variables
50          VariableRecycled variableRecycled = VariableRecycled.detect(context, opportunities);
51          if (variableRecycled != null) {
52              context.caseOfInterestList.add(variableRecycled);
53          }
54      }
55
56      @Override
57      public void transformCase(TransformationPhaseContext context) {
58          List<VariableDeclared> variables = context.caseOfInterestList.stream()
59                  .filter(VariableDeclared.class::isInstance)
60                  .map(VariableDeclared.class::cast).collect(Collectors.toList());
61          if (context.caseOfInterest instanceof VariableUsed) {
62              // Filtering considering the variables declared
63              VariableUsed variableUsed = (VariableUsed) context.caseOfInterest;
64              boolean interesting = variables.stream().anyMatch(variableDeclared -> variableUsed.variableAccesses.stream()
65                      .anyMatch(ctVariableAccess -> ctVariableAccess.getVariable().getSimpleName()
66                              .equals(variableDeclared.variable.getSimpleName())));
67              if (interesting) {
68                  context.accept(context.caseOfInterest);
69              }
70          } else if (context.caseOfInterest instanceof VariableReassigned) {
71              // Filtering considering the variables declared
72              VariableReassigned variableReassigned = (VariableReassigned) context.caseOfInterest;
73              boolean interesting = variables.stream().anyMatch(variableDeclared -> {
74                  CtExpression assigned = variableReassigned.assignment.getAssigned();
75                  return (assigned instanceof CtVariableWrite &&
76                          ((CtVariableWrite) assigned).getVariable().getSimpleName()
77                                  .equals(variableDeclared.variable.getSimpleName()));
78              });
79              if (interesting) {
80                  context.accept(context.caseOfInterest);
81              }
82          } else {
83              CaseTransformer.createPassThroughTransformation().transformCase(context);
84          }
85      }
86
87      private List<CaseOfInterest> getCasesByVariableName(String variableName, List<CaseOfInterest> caseOfInterests) {
88          return caseOfInterests.stream().filter(caseOfInterest -> {
89              if (caseOfInterest instanceof VariableDeclared) {
90                  return ((VariableDeclared) caseOfInterest).variable.getSimpleName().equals(variableName);
91              } else if (caseOfInterest instanceof VariableReassigned) {
92                  CtExpression assigned = ((VariableReassigned) caseOfInterest).assignment.getAssigned();
93                  if (assigned instanceof CtVariableWrite) {
94                      return ((CtVariableWrite) assigned).getVariable().getSimpleName().equals(variableName);
95                  }
96              } else if (caseOfInterest instanceof VariableUsed) {
97                  return ((VariableUsed) caseOfInterest).variableAccesses.stream()
98                          .anyMatch(ctVariableAccess -> ctVariableAccess.getVariable().getSimpleName()
99                                  .equals(variableName));
100             } else if (caseOfInterest instanceof VariableLost) {
101                 return ((VariableLost) caseOfInterest).variableAccesses.stream()
102                         .anyMatch(ctVariableAccess -> ctVariableAccess.getVariable().getSimpleName()
103                                 .equals(variableName));
104             } else if (caseOfInterest instanceof VariableRecycled) {
105                 return ((VariableRecycled) caseOfInterest).variableAccesses.stream()
106                         .anyMatch(ctVariableAccess -> ctVariableAccess.getVariable().getSimpleName()
107                                 .equals(variableName));
108             }
109             return false;
110         }).collect(Collectors.toList());
111     }
112
113     private String getTypeByVariableName(String variableName, List<CaseOfInterest> caseOfInterests) {
114         Optional<VariableDeclared> match = caseOfInterests.stream().filter(VariableDeclared.class::isInstance)
```

```
115                 .map(VariableDeclared.class::cast)
116                 .filter(variableDeclared -> variableDeclared.variable.getSimpleName().equals(variableName))
117                 .findFirst();
118         if (!match.isPresent()) {
119             return null;
120         }
121         return match.get().variable.getType().getSimpleName();
122     }
123 
124     private boolean isVariableUnderControl(String variableName, RefactoringPhaseContext context) {
125         List<CaseOfInterest> filtered = getCasesByVariableName(variableName, context.casesOfInterest);
126         // NOTE: Only check up to this point in the phase
127         filtered = filtered.stream()
128                 .filter(VariableLost.class::isInstance)
129                 .map(VariableLost.class::cast)
130                 .filter(variableLost -> variableLost.getStatementIndex() < context.caseOfInterest.getStatementIndex())
131                 .collect(Collectors.toList());
132         return filtered.size() == 0;
133     }
134 
135     private boolean wasVariableRecycled(String variableName, RefactoringPhaseContext context) {
136         List<CaseOfInterest> filtered = getCasesByVariableName(variableName, context.casesOfInterest);
137         int index = filtered.indexOf(context.caseOfInterest);
138         if (context.caseOfInterest instanceof VariableReassigned) {
139             // We do not want to consider this case of interest
140             index--;
141         }
142         filtered = filtered.subList(0, index);
143         for (int i = filtered.size() - 1; i >= 0; i--) {
144             CaseOfInterest current = filtered.get(i);
145             // NOTE: Only check up to this point in the phase and after the last declaration or redeclaration of this variable
146             if (current instanceof VariableReassigned || current instanceof VariableDeclared) {
147                 break;
148             } else if (current instanceof VariableRecycled) {
149                 return true;
150             }
151         }
152         return false;
153     }
154 
155     private void recycleVariableDeclared(RefactoringPhaseContext context) {
156         if (!(context.caseOfInterest instanceof VariableDeclared)) {
157             return;
158         }
159         VariableDeclared variableDeclared = (VariableDeclared) context.caseOfInterest;
160         String variableName = variableDeclared.variable.getSimpleName();
161         String typeName = opportunities.get(getTypeByVariableName(variableName, context.casesOfInterest));
162         if (typeName == null) {
163             return;
164         }
165         List<CaseOfInterest> casesOfInterest = getCasesByVariableName(variableName, context.casesOfInterest); // TODO - EXCLUDE RETURN STATEMENTS
166         boolean isLast = casesOfInterest.get(casesOfInterest.size() - 1).equals(context.caseOfInterest);
167         if (!isLast) {
168             return;
169         }
170         Factory factory = context.caseOfInterest.getStatement().getFactory();
171         CtIf ctIf = factory.createIf();
172         CtBlock ctBlock = factory.createBlock();
173         ctBlock.addStatement(factory
174                 .createCodeSnippetStatement(String.format("%s.%s()", variableName, typeName)));
175         ctIf.setThenStatement(ctBlock);
176         ctIf.setCondition(factory
177                 .createCodeSnippetExpression(String.format("%s != null", variableName)));
178         context.caseOfInterest.getStatement().insertAfter(ctIf);
179     }
180 
181     private void recycleVariableReassigned(RefactoringPhaseContext context) {
182         if (!(context.caseOfInterest instanceof VariableReassigned)) {
183             return;
184         }
```

```
185         // We consider reassigns because there could be no usage in between Declarations and Reassignments
186         CtExpression assigned = ((VariableReassigned) context.caseOfInterest).assignment.getAssigned();
187         if (assigned instanceof CtVariableWrite) {
188             String variableName = ((CtVariableWrite) assigned).getVariable().getSimpleName();
189             String typeName = opportunities.get(getTypeByVariableName(variableName, context.casesOfInterest));
190             if (typeName == null) {
191                 return;
192             }
193
194             boolean wasVariableRecycled = wasVariableRecycled(variableName, context);
195             if (wasVariableRecycled) {
196                 return;
197             }
198
199             boolean isInControl = isVariableUnderControl(variableName, context);
200             if (!isInControl) {
201                 return;
202             }
203
204             Factory factory = assigned.getFactory();
205             CtIf ctIf2 = factory.createIf();
206             CtBlock ctBlock2 = factory.createBlock();
207             ctBlock2.addStatement(factory
208                     .createCodeSnippetStatement(String.format("%s.%s()", variableName, typeName)));
209             ctIf2.setThenStatement(ctBlock2);
210             ctIf2.setCondition(factory
211                     .createCodeSnippetExpression(String.format("%s != null", variableName)));
212             context.caseOfInterest.getStatement().insertBefore(ctIf2);
213
214
215             List<CaseOfInterest> casesOfInterest = getCasesByVariableName(variableName, context.casesOfInterest);
216             boolean isLast = casesOfInterest.get(casesOfInterest.size() - 1).equals(context.caseOfInterest);
217             if (!isLast) {
218                 return;
219             }
220
221             CtIf ctIf = factory.createIf();
222             CtBlock ctBlock = factory.createBlock();
223             ctBlock.addStatement(factory
224                     .createCodeSnippetStatement(String.format("%s.%s()", variableName, typeName)));
225             ctIf.setThenStatement(ctBlock);
226             ctIf.setCondition(factory
227                     .createCodeSnippetExpression(String.format("%s != null", variableName)));
228             context.caseOfInterest.getStatement().insertAfter(ctIf);
229
230
231         }
232     }
233
234     private void recycleVariableUsed(RefactoringPhaseContext context) {
235         if (!(context.caseOfInterest instanceof VariableUsed)) {
236             return;
237         }
238         List<CtVariableAccess> variableAccesses = ((VariableUsed) context.caseOfInterest).variableAccesses;
239         Set<String> alreadyRecycles = new HashSet<>();
240         variableAccesses.forEach(ctVariableAccess -> {
241             String variableName = ctVariableAccess.getVariable().getSimpleName();
242
243             if (alreadyRecycles.contains(variableName)) {
244                 return;
245             }
246
247             String typeName = opportunities.get(getTypeByVariableName(variableName, context.casesOfInterest));
248             if (typeName == null) {
249                 return;
250             }
251             List<CaseOfInterest> casesOfInterest = getCasesByVariableName(variableName, context.casesOfInterest);
252             boolean isLast = casesOfInterest.get(casesOfInterest.size() - 1).equals(context.caseOfInterest);
253             if (!isLast) {
254                 return;
```

```
255                 }
256                 boolean wasVariableRecycled = wasVariableRecycled(variableName, context);
257                 if (wasVariableRecycled) {
258                     return;
259                 }
260 
261                 boolean isInControl = isVariableUnderControl(variableName, context);
262                 if (!isInControl) {
263                     return;
264                 }
265 
266                 Factory factory = ctVariableAccess.getFactory();
267                 CtIf ctIf = factory.createIf();
268                 CtBlock ctBlock = factory.createBlock();
269                 ctBlock.addStatement(factory
270                         .createCodeSnippetStatement(String.format("%s.%s()", variableName, typeName)));
271                 ctIf.setThenStatement(ctBlock);
272                 ctIf.setCondition(factory
273                         .createCodeSnippetExpression(String.format("%s != null", variableName)));
274                 context.caseOfInterest.getStatement().insertAfter(ctIf);
275                 alreadyRecycles.add(variableName);
276             });
277     }
278 
279     @Override
280     public void refactorCase(RefactoringPhaseContext context) {
281 //        System.out.println("BEFORE: " + context.block.toStringDebug());
282         recycleVariableDeclared(context);
283         recycleVariableReassigned(context);
284         recycleVariableUsed(context);
285 //        System.out.println("AFTER: " + context.block.toStringDebug());
286     }
287 
288     private void refactor(CtMethod method) {
289         List<CtBlock> blocks = RefactoringRule.getCtElementsOfInterest(method, CtBlock.class::isInstance, CtBlock.class);
290         for (CtBlock block : blocks) {
291             Iterable.iterateBlock(this, logger, block, false, 0);
292         }
293     }
294 
295     public void process(CtClass element) {
296         Set methods = element.getMethods();
297         for (Object method : methods) {
298             if (method instanceof CtMethod) {
299                 refactor((CtMethod) method);
300             }
301         }
302     }
303 ...
304 }
```

Listing A.5: The ViewHolderRefactoringRule class

```
1  ...
2  public class ViewHolderRefactoringRule extends AbstractProcessor<CtClass> implements RefactoringRule<CtClass> {
3  
4      private IterationLogger logger;
5  
6      public ViewHolderRefactoringRule(IterationLogger logger) {
7          this.logger = logger;
8      }
9  
10     private class RefactoringPhaseExtra {
11         String viewVariableName = null;
12         String argumentName = null;
13         Factory factory = null;
```

```
14          boolean convertViewInflated = false; // If true the convertView is properly null checked and assigned the layout
15          boolean hasViewHolderInstance = false; // If true at this point there is a viewHolderInstance
16          boolean isPopulatingViewHolder = false; // If true there is a condition for populating the viewHolder
17          boolean hasIfPreamble = false;
18          boolean hasIfStmt = false;
19          CtIf ifStmt = null;
20          CtBlock thenBlock = null;
21          String viewHolderInstanceName = "viewHolderItem";
22
23          CtIf createIfStatement(RefactoringPhaseContext context) {
24              CtIf ifStmt = factory.createIf();
25
26              ifStmt.setCondition(factory.createCodeSnippetExpression(String.format("%s == null", this.viewHolderInstanceName)));
27              CtStatement st1 = factory.createCodeSnippetStatement(String.format("viewHolderItem = new ViewHolderItem()", this.
                    viewHolderInstanceName));
28              CtStatement st2 = factory.createCodeSnippetStatement(String.format("%s.setTag(viewHolderItem)", argumentName));
29
30              CtBlock thenBlock = factory.createBlock();
31              thenBlock.addStatement(st1);
32              thenBlock.addStatement(st2);
33              ifStmt.setThenStatement(thenBlock);
34
35              this.isPopulatingViewHolder = true;
36              this.ifStmt = ifStmt;
37              this.thenBlock = thenBlock;
38              this.hasIfStmt = true;
39
40              return ifStmt;
41          }
42      }
43
44      @Override
45      public void detectCase(DetectionPhaseContext context) {
46          VariableDeclared variableDeclared = VariableDeclared.detect(context);
47          if (variableDeclared != null) {
48              context.caseOfInterestList.add(variableDeclared);
49          }
50          ConvertViewReassignInflator convertViewReassignInflator = ConvertViewReassignInflator.detect(context);
51          if (convertViewReassignInflator != null) {
52              context.caseOfInterestList.add(convertViewReassignInflator);
53          }
54          ConvertViewReuseWithTernary convertViewReuseWithTernary = ConvertViewReuseWithTernary.detect(context);
55          if (convertViewReuseWithTernary != null) {
56              context.caseOfInterestList.add(convertViewReuseWithTernary);
57          }
58          VariableAssignedGetTag variableAssignedGetTag = VariableAssignedGetTag.detect(context);
59          if (variableAssignedGetTag != null) {
60              context.caseOfInterestList.add(variableAssignedGetTag);
61          }
62          VariableAssignedFindViewById variableAssignedFindViewById = VariableAssignedFindViewById.detect(context);
63          if (variableAssignedFindViewById != null) {
64              context.caseOfInterestList.add(variableAssignedFindViewById);
65          }
66          VariableAssignedInflator variableAssignedInflator = VariableAssignedInflator.detect(context);
67          if (variableAssignedInflator != null) {
68              context.caseOfInterestList.add(variableAssignedInflator);
69          }
70          VariableCheckNull variableCheckNull = VariableCheckNull.detect(context);
71          if (variableCheckNull != null) {
72              context.caseOfInterestList.add(variableCheckNull);
73          }
74      }
75
76      @Override
77      public void transformCase(TransformationPhaseContext context) {
78          // No need for transformations in this case
79          CaseTransformer.createPassThroughTransformation().transformCase(context);
80      }
81
82      @Override
```

```
83      public void refactorCase(RefactoringPhaseContext context) {
84          // Variables for easier access
85          RefactoringPhaseExtra extra = (RefactoringPhaseExtra) context.extra;
86
87          if (context.caseOfInterest instanceof VariableAssignedInflator) {
88              // In this case we want to check where the inflated view is stored
89              VariableAssignedInflator variableAssignedInflator = (VariableAssignedInflator) context.caseOfInterest;
90              if (variableAssignedInflator.variable.getSimpleName().equals(extra.argumentName)) {
91                  return;
92              }
93              CtConditional conditional = extra.factory.createConditional();
94              conditional.setCondition(extra.factory.createCodeSnippetExpression(extra.argumentName + " == null"));
95              conditional.setElseExpression(extra.factory.createCodeSnippetExpression(extra.argumentName));
96              if (variableAssignedInflator.getStatement() instanceof CtVariable) {
97                  CtVariable variable = ((CtVariable) variableAssignedInflator.getStatement());
98                  conditional.setThenExpression(variable.getDefaultExpression());
99                  variable.setDefaultExpression(conditional);
100             } else if (variableAssignedInflator.getStatement() instanceof CtAssignment) {
101                 CtAssignment assignment = (CtAssignment) variableAssignedInflator.getStatement();
102                 conditional.setThenExpression(assignment.getAssignment());
103                 assignment.setAssignment(conditional);
104             }
105             extra.viewVariableName = variableAssignedInflator.variable.getSimpleName();
106             extra.convertViewInflated = true;
107         } else if (context.caseOfInterest instanceof ConvertViewReassignInflator) {
108             ConvertViewReassignInflator convertViewReassignInflator = (ConvertViewReassignInflator) context.caseOfInterest;
109             CtExpression assignment = convertViewReassignInflator.assignment.getAssignment();
110             CtConditional conditional = extra.factory.createConditional();
111             conditional.setCondition(extra.factory.createCodeSnippetExpression(extra.argumentName + " == null"));
112             conditional.setThenExpression(assignment);
113             conditional.setElseExpression(extra.factory.createCodeSnippetExpression(extra.argumentName));
114             convertViewReassignInflator.assignment.setAssignment(conditional);
115             extra.viewVariableName = extra.argumentName;
116             extra.convertViewInflated = true;
117         } else if (context.caseOfInterest instanceof ConvertViewReuseWithTernary) {
118             // In this case everything is well no need to worry about the convertView anymore.
119             extra.viewVariableName = extra.argumentName;
120             extra.convertViewInflated = true;
121         } else if (context.caseOfInterest instanceof VariableAssignedGetTag) {
122             // In this case we want to check the name of the viewHolderInstance
123             VariableAssignedGetTag variableAssignedGetTag = (VariableAssignedGetTag) context.caseOfInterest;
124             extra.viewHolderInstanceName = variableAssignedGetTag.variable.getSimpleName();
125             extra.hasViewHolderInstance = true;
126         } else if (context.caseOfInterest instanceof VariableCheckNull) {
127             VariableCheckNull variableCheckNull = (VariableCheckNull) context.caseOfInterest;
128             if (variableCheckNull.variable.getSimpleName().equals(extra.viewHolderInstanceName)) {
129                 extra.hasIfStmt = true;
130                 extra.ifStmt = variableCheckNull.ifStmt;
131                 extra.thenBlock = variableCheckNull.ifStmt.getThenStatement();
132             }
133         } else if (context.caseOfInterest instanceof VariableAssignedFindViewById && extra.convertViewInflated) { // The convert view must be
                inflated otherwise we do not want any changes
134             VariableAssignedFindViewById variableAssignedFindViewById = (VariableAssignedFindViewById) context.caseOfInterest;
135             // Find the Class that contains this case
136             CtClass rootClass = RefactoringRule.getClosestClassParent(variableAssignedFindViewById.getStatement());
137             // Find every inner class inside the root class that matches the viewHolder description
138             List<CtClass> viewHolderItemClasses = RefactoringRule.getCtElementsOfInterest(rootClass, node -> {
139                 // TODO - We have a situation where class could be declared inside another inner class, we should search only narrowly
140                 if (node instanceof CtClass) {
141                     CtClass ctClass = (CtClass) node;
142 //                    boolean isStatic = classOrInterfaceDeclaration.isStatic();
143                     return ctClass.getSimpleName().equals("ViewHolderItem");
144                 }
145                 return false;
146             }, CtClass.class);
147             CtClass ctClass;
148             if (viewHolderItemClasses.size() == 0) {
149                 // There isn't a viewHolder - Create it with the field inside it
150                 ctClass = extra.factory.createClass("ViewHolderItem");
151                 ctClass.addModifier(ModifierKind.STATIC);
```

```
152                 CtTypeReference typeReference = variableAssignedFindViewById.variable.getType();
153                 CtField field = extra.factory.createCtField(variableAssignedFindViewById.variable.getSimpleName(), typeReference,
154                         "null", ModifierKind.PUBLIC);
155                 ctClass.addTypeMember(field);
156                 CtMethod method = RefactoringRule.getClosestMethodParent(variableAssignedFindViewById.getStatement());
157                 int methodIndex = rootClass.getTypeMembers().indexOf(method);
158                 rootClass.addTypeMemberAt(methodIndex, ctClass);
159             } else {
160                 // There is a viewHolder - Check if the field is inside, create it if necessary
161                 ctClass = viewHolderItemClasses.get(0);
162                 List<CtField<?>> fields = ctClass.getFields();
163                 Optional<CtField<?>> optionalField = fields.stream().filter(field -> field.getSimpleName()
164                         .equals(variableAssignedFindViewById.variable.getSimpleName())).findFirst();
165                 if (optionalField.isPresent() && !optionalField.get().getType().getSimpleName()
166                         .equals(variableAssignedFindViewById.variable.getType().getSimpleName())) {
167                     // If types do not match we ignore for now.
168                     return;
169                 }
170                 if (!optionalField.isPresent()) {
171                     CtTypeReference typeReference = variableAssignedFindViewById.variable.getType();
172                     CtField field = extra.factory.createCtField(variableAssignedFindViewById.variable.getSimpleName(), typeReference,
173                             "null", ModifierKind.PUBLIC);
174                     ctClass.addField(field); // TODO - Printer leaves a blank line after the field.
175                 }
176             }
177
178             // MILESTONE: From this point we know that we have a ViewHolder class and a field variable matching the assigned variable
179
180             // New variables for easier access
181             Set<CtTypeReference<?>> ctTypeReferences = variableAssignedFindViewById.resource.getReferencedTypes();
182
183             if (!extra.hasViewHolderInstance) {
184                 // Then we need to create it.
185                 // Create a statement to get the viewHolder instance if it exists.
186                 CtStatement newStatement1 = extra.factory.createCodeSnippetStatement(String.format(
187                         "ViewHolderItem %s = (ViewHolderItem) %s.getTag()",
188                         extra.viewHolderInstanceName, extra.viewVariableName));
189                 if (extra.hasIfStmt) {
190                     extra.ifStmt.insertBefore(newStatement1);
191                 } else {
192                     context.caseOfInterest.getStatement().insertBefore(newStatement1);
193                 }
194                 extra.hasViewHolderInstance = true;
195             }
196
197             if (!extra.isPopulatingViewHolder) {
198                 // We have reached this point which means that the there is no if statement for populating the ViewHolder
199                 CtIf ctIf = extra.createIfStatement(context);
200                 context.caseOfInterest.getStatement().insertBefore(ctIf);
201             }
202
203             // TODO - we need to check if refactoringPhaseExtra.thenBlock already populates the viewHolder, we don't want duplicates
204             if (true) {
205                 System.out.println("variableAssignedFindViewById.resource - " + variableAssignedFindViewById.resource);
206                 extra.thenBlock.addStatement(extra.factory.createCodeSnippetStatement(
207                         String.format("%s.%s = (TextView) %s.findViewById(%s)",
208                                 extra.viewHolderInstanceName,
209                                 variableAssignedFindViewById.variable.getSimpleName(),
210                                 extra.viewVariableName, variableAssignedFindViewById.resource.toString())));
211             }
212
213             // Replace the assignment of the variable with the ViewHolder field
214             CtStatement statement = variableAssignedFindViewById.getStatement();
215             CtExpression assignmentExpression = extra.factory
216                     .createCodeSnippetExpression(String.format("%s.%s",
217                             extra.viewHolderInstanceName,
218                             variableAssignedFindViewById.variable.getSimpleName()));
219             if (statement instanceof CtVariable) {
220                 CtVariable ctVariable = ((CtVariable) statement);
221                 ctVariable.setDefaultExpression(assignmentExpression);
```

```
222             } else if (statement instanceof CtAssignment) {
223                 CtAssignment assignment = ((CtAssignment) statement);
224                 assignment.setAssignment(assignmentExpression);
225             }
226         }
227     }
228     @Override
229     public void onWillRefactor(RefactoringPhaseContext context) {
230         // Create a Refactoring phase extra for data support
231         RefactoringPhaseExtra extra = new RefactoringPhaseExtra();
232         context.extra = extra;
233         if (context.casesOfInterest.size() > 0) {
234             String argumentName = ((CtParameter) Objects.requireNonNull(
235                     RefactoringRule.getClosestMethodParent(context.casesOfInterest.get(0).getStatement()))
236                     .getParameters().get(1)).getSimpleName();
237             Factory factory = Objects.requireNonNull(context.block).getFactory();
238             extra.argumentName = argumentName;
239             extra.viewVariableName = argumentName;
240             extra.factory = factory;
241         }
242     }
243
244     @Override
245     public void process(CtClass element) {
246         Set methods = element.getMethods();
247         for (Object method : methods) {
248             if (method instanceof CtMethod) {
249                 refactor((CtMethod) method);
250             }
251         }
252     }
253
254     private void refactor(CtMethod method) {
255         if (!methodSignatureMatches(method)) {
256             return;
257         }
258         Iterable.iterateMethod(this, logger, method, false);
259     }
260
261     private boolean methodSignatureMatches(CtMethod method) {
262         // SIGNATURE:
263         // public View getView(final int position, final View convertView, final ViewGroup parent)
264         boolean nameMatch = method.getSimpleName().equals("getView");
265         CtTypeReference type = method.getType();
266         boolean returnTypeMatch = type.getSimpleName().equals("View");
267
268         boolean isPublic = method.getModifiers().contains(ModifierKind.PUBLIC);
269         boolean hasSameNumberOfArguments = method.getParameters().size() == 3;
270         if (hasSameNumberOfArguments) {
271             List parameterList = method.getParameters();
272
273             CtTypeReference firstArgumentType = ((CtParameter) parameterList.get(0)).getType();
274             boolean firstArgumentTypeMatches = firstArgumentType.isPrimitive() &&
275                     firstArgumentType.getSimpleName().equals("int");
276
277             CtTypeReference secondArgumentType = ((CtParameter) parameterList.get(1)).getType();
278             boolean secondArgumentTypeMatches = secondArgumentType.getSimpleName().equals("View");
279
280             CtTypeReference thirdArgumentType = ((CtParameter) parameterList.get(2)).getType();
281             boolean thirdArgumentTypeMatches = thirdArgumentType.getSimpleName().equals("ViewGroup");
282
283             return isPublic &&
284                     nameMatch &&
285                     returnTypeMatch &&
286                     firstArgumentTypeMatches &&
287                     secondArgumentTypeMatches &&
288                     thirdArgumentTypeMatches;
289         }
290
291         return false;
```

```
292         }
293         ...
294     }
```

Listing A.6: The WakeLockRefactoringRule class

```
1   ...
2   public class WakeLockRefactoringRule extends AbstractProcessor<CtClass> implements RefactoringRule<CtClass> {
3       ...
4       private boolean methodSignatureMatches(CtMethod method) {
5           // SIGNATURE:
6           // protected void onCreate(Bundle savedInstanceState)
7           boolean nameMatch = method.getSimpleName().equals("onCreate");
8           CtTypeReference type = method.getType();
9           boolean returnTypeMatch = type != null && type.getSimpleName().equals("void");
10
11          boolean hasSameNumberOfArguments = method.getParameters().size() == 1;
12          if (hasSameNumberOfArguments) {
13              List parameterList = method.getParameters();
14
15              CtTypeReference firstArgumentType = ((CtParameter) parameterList.get(0)).getType();
16              boolean firstArgumentTypeMatches = firstArgumentType.getSimpleName().endsWith("Bundle");
17
18              return nameMatch &&
19                      returnTypeMatch &&
20                      firstArgumentTypeMatches;
21          }
22
23          return false;
24      }
25
26      @Override
27      public void detectCase(DetectionPhaseContext context) {
28          WakeLockAcquired wakeLockAcquired = WakeLockAcquired.detect(context);
29          if (wakeLockAcquired != null) {
30              context.caseOfInterestList.add(wakeLockAcquired);
31          }
32
33          VariableDeclared variableDeclared = VariableDeclared.detect(context);
34          if (variableDeclared != null) {
35              context.caseOfInterestList.add(variableDeclared);
36          }
37      }
38
39      @Override
40      public void transformCase(TransformationPhaseContext context) {
41          CaseTransformer.createPassThroughTransformation().transformCase(context);
42      }
43
44      @Override
45      public void refactorCase(RefactoringPhaseContext context) {
46          if (context.caseOfInterest instanceof WakeLockAcquired) {
47              WakeLockAcquired wakeLockAcquired = (WakeLockAcquired) context.caseOfInterest;
48              Optional<VariableDeclared> optionalVariableDeclared = context.casesOfInterest.stream()
49                      .filter(VariableDeclared.class::isInstance)
50                      .map(VariableDeclared.class::cast)
51                      .filter(variableDeclared -> variableDeclared.variable.getSimpleName()
52                              .equals(wakeLockAcquired.variable.getVariable().getSimpleName())).findFirst();
53              if (optionalVariableDeclared.isPresent()) {
54
55                  CtClass ctClass = RefactoringRule.getClosestClassParent(context.block);
56                  if (ctClass == null) {
57                      return;
58                  }
59                  List<CtField<?>> fields = ctClass.getFields();
60                  Optional<CtField<?>> optionalField = fields.stream().filter(field -> field.getSimpleName()
```

```
61                      .equals(optionalVariableDeclared.get().variable.getSimpleName())).findFirst();
62              if (optionalField.isPresent() && !optionalField.get().getType().getSimpleName()
63                      .equals(optionalVariableDeclared.get().variable.getType().getSimpleName())) {
64                  // If types do not match we ignore for now.
65                  return;
66              }
67              if (!optionalField.isPresent()) {
68                  CtTypeReference typeReference = optionalVariableDeclared.get().variable.getType();
69                  CtField field = ctClass.getFactory().createCtField(optionalVariableDeclared.get().variable.getSimpleName(), typeReference,
70                          "null", ModifierKind.PRIVATE);
71                  ctClass.addField(field);
72
73                  Factory factory = optionalVariableDeclared.get().getStatement().getFactory();
74                  CtAssignment assignment = factory.createAssignment();
75                  assignment.setAssigned(factory.createCodeSnippetExpression(optionalVariableDeclared.get().variable.getSimpleName()));
76                  assignment.setAssignment(optionalVariableDeclared.get().variable.getDefaultExpression());
77                  optionalVariableDeclared.get().getStatement().insertBefore(assignment);
78                  context.block.removeStatement(optionalVariableDeclared.get().getStatement());
79                  context.block.removeStatement(wakeLockAcquired.getStatement());
80              }
81
82          }
83
84          // From here on we assume the variable is in the class as a field
85
86          CtClass ctClass = RefactoringRule.getClosestClassParent(context.block);
87          if (ctClass == null) {
88              return;
89          }
90
91          Set<CtMethod<?>> methods = ctClass.getMethods();
92          boolean hasOnPause = false;
93          boolean hasOnResume = false;
94          boolean hasOnDestroy = false;
95          String variableName = wakeLockAcquired.variable.getVariable().getSimpleName();
96          for (CtMethod<?> ctMethod : methods) {
97
98              if (ctMethod.getParameters().size() != 0 || !ctMethod.getType().getSimpleName().equals("void")) {
99                  continue;
100             }
101
102             switch (ctMethod.getSimpleName()) {
103                 case "onPause":
104                     hasOnPause = true;
105                     boolean hasRelease = false;
106                     for (CtStatement statement : ctMethod.getBody().getStatements()) {
107                         if (statement instanceof CtInvocation) {
108                             CtInvocation invocation = (CtInvocation) statement;
109                             CtExpression target = invocation.getTarget();
110                             if (target.toString().equals(variableName) &&
111                                     invocation.getExecutable().getSimpleName().equals("release")) {
112                                 hasRelease = true;
113                                 break;
114                             }
115                         }
116                     }
117                     if (!hasRelease) {
118                         ctMethod.getBody().addStatement(ctMethod.getFactory()
119                                 .createCodeSnippetStatement(variableName + ".release()"));
120                     }
121
122                     break;
123                 case "onResume":
124                     hasOnResume = true;
125
126                     boolean hasAcquire = false;
127                     for (CtStatement statement : ctMethod.getBody().getStatements()) {
128                         if (statement instanceof CtInvocation) {
129                             CtInvocation invocation = (CtInvocation) statement;
130                             CtExpression target = invocation.getTarget();
```

```
131                             if (target.toString().equals(variableName) &&
132                                     invocation.getExecutable().getSimpleName().equals("acquire")) {
133                                 hasAcquire = true;
134                                 break;
135                             }
136                         }
137                     }
138                     if (!hasAcquire) {
139                         ctMethod.getBody().addStatement(ctMethod.getFactory()
140                                 .createCodeSnippetStatement(variableName + ".acquire()"));
141                     }
142 
143                     break;
144 
145                 case "onDestroy":
146                     hasOnDestroy = true;
147 
148                     boolean hasDestroy = false;
149                     for (CtStatement statement : ctMethod.getBody().getStatements()) {
150                         if (statement instanceof CtInvocation) {
151                             CtInvocation invocation = (CtInvocation) statement;
152                             CtExpression target = invocation.getTarget();
153                             if (target.toString().equals(variableName) &&
154                                     invocation.getExecutable().getSimpleName().equals("release")) {
155                                 hasDestroy = true;
156                                 break;
157                             }
158                         }
159                     }
160                     if (!hasDestroy) {
161                         ctMethod.getBody().addStatement(ctMethod.getFactory()
162                                 .createCodeSnippetStatement(variableName + ".release()"));
163                     }
164 
165                     break;
166             }
167         }
168 
169         Factory factory = ctClass.getFactory();
170 
171         if (!hasOnPause) {
172             CtMethod method = factory.createMethod();
173             method.setType(factory.Type().voidPrimitiveType());
174             method.setSimpleName("onPause");
175             CtAnnotation<Annotation> annotation = factory.Code()
176                     .createAnnotation(getFactory().Code().createCtTypeReference(Override.class));
177             method.addAnnotation(annotation);
178             method.setBody(factory.createBlock());
179             method.getBody().addStatement(factory.createCodeSnippetStatement("super.onPause()"));
180             method.getBody().addStatement(factory.createCodeSnippetStatement(variableName + ".release()"));
181             ctClass.addMethod(method);
182         }
183 
184         if (!hasOnResume) {
185             CtMethod method = factory.createMethod();
186             method.setType(factory.Type().voidPrimitiveType());
187             method.setSimpleName("onResume");
188             CtAnnotation<Annotation> annotation = factory.Code()
189                     .createAnnotation(getFactory().Code().createCtTypeReference(Override.class));
190             method.addAnnotation(annotation);
191             method.setBody(factory.createBlock());
192             method.getBody().addStatement(factory.createCodeSnippetStatement("super.onResume()"));
193             method.getBody().addStatement(factory.createCodeSnippetStatement(variableName + ".acquire()"));
194             ctClass.addMethod(method);
195         }
196 
197         if (!hasOnDestroy) {
198             CtMethod method = factory.createMethod();
199             method.setType(factory.Type().voidPrimitiveType());
200             CtAnnotation<Annotation> annotation = factory.Code()
```

```
201                             .createAnnotation(getFactory().Code().createCtTypeReference(Override.class));
202                     method.addAnnotation(annotation);
203                     method.setSimpleName("onDestroy");
204                     method.setBody(factory.createBlock());
205                     method.getBody().addStatement(factory.createCodeSnippetStatement("super.onDestroy()"));
206                     method.getBody().addStatement(factory.createCodeSnippetStatement(variableName + ".release()"));
207                     ctClass.addMethod(method);
208                 }
209             }
210         }
211
212         private void refactor(CtMethod method) {
213             if (!methodSignatureMatches(method)) {
214                 return;
215             }
216             List<CtBlock> blocks = RefactoringRule.getCtElementsOfInterest(method, CtBlock.class::isInstance, CtBlock.class);
217             for (CtBlock block : blocks) {
218                 Iterable.iterateBlock(this, logger, block, false, 0);
219             }
220         }
221
222         public void process(CtClass element) {
223             Set methods = element.getMethods();
224             for (Object method : methods) {
225                 if (method instanceof CtMethod) {
226                     refactor((CtMethod) method);
227                 }
228             }
229         }
230     ...
231     }
```