Automatic Refactoring for Energy Efficiency in Continuous Integration Pipelines

Ricardo Morais
Instituto Superior Técnico,
Lisbon, Portugal
ricardojhmorais@tecnico.ulisboa.pt

Luís Cruz
INESC TEC
Porto, Portugal
luiscruz@fe.up.pt

Rui Abreu
INESC-ID
Lisbon, Portugal
rui@computer.org

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.

This work introduces 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.

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

ACM Classification Keywords

Software and its engineering---Software notations and tools---Software maintenance tool

INTRODUCTION

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 [1, 2, 3]. In particular, Cruz and Abreu studied the impact of fixing eight Android performance-related anti-patterns on energy efficiency [2] 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 [3], 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 [3, 4], this work introduces the former 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 work focus on automation and adaptability by releasing a new implementation of Leafactor, called LeafactorCI, published as a Gradle plugin powered by Spoon (https://github.com/INRIA/spoon). The fact that most 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 [5, 6, 7, 8]. Vasilescu et al. assessed the effects of continuous integration by gathering data from GitHub [36]. 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.

Objectives

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 [4]. 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 [5]. This work 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 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 three 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?
    - Can LeafactorCI effectively relay refactoring decisions to the project integrators?
    - How easy it is to adopt LeafactorCI?

RELATED WORK

In [12], 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, eProf, and eCalc. In the same study Cruz and Abreu use a hardware power measuring device to evaluate their work, 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.

Cruz and Abreu also compiled a guideline for fixing a set of five patterns which form in Android projects and impact energy efficiency. They are:

- ViewHolder - This pattern appears in List Views. When in a List View, the system must draw each item and the problem arises when the method findViewById is called several times, this method is known for being a very expensive method.
- DrawAllocation - 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 non-smooth User Interface (UI).
- WakeLock - 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 to perform tasks. If an application fails to release a wake lock or uses it without being strictly necessary, it can drain the battery.
- ObsoleteLayoutParam - During development, UI views might be refactored several times. In this process, some parameters might be left unchanged even when they have no effect 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.

Cruz and Abreu used the refactoring technique to fix anti-patterns in the source-code of mobile applications. In earlier work [3], 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 [9] and the other is an Extensible Markup Language (XML) refactoring engine made from scratch to deal with layout related anti-patterns. 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 plugin. This is a disadvantage as it restricts its domain and therefore its usefulness.

Testing is a fundamental part of continuous integration [5] as such it needs to be taken seriously. Cruz et al. investigated the working habits and challenges of mobile app developers with respect to testing [5]. They conduct a large-scale study on 1000 open-source android applications and concluded that android apps are failing to use automated testing. About 40% of the applications had used testing technologies. The testing technologies were JUnit (used in unit testing); and Espresso (used in Graphical User Interface (GUI) testing). Cloud testing services were not very adopted, the most used technology was 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 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 [5].

ARCHITECTURE

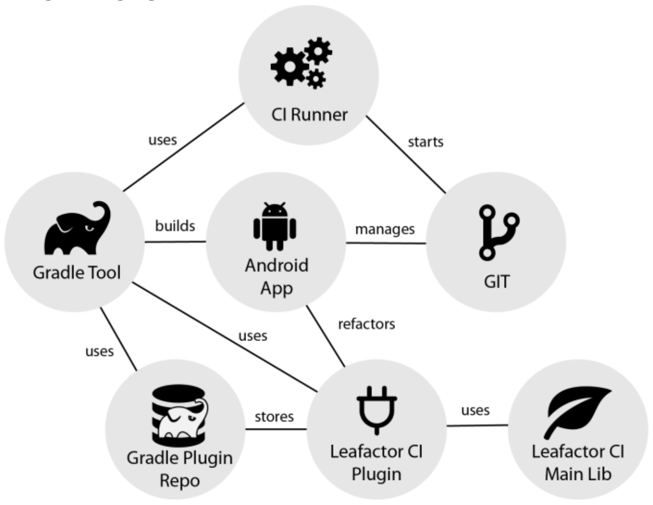


Figure 1. LeafactorCI architecture.

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 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. In LeafactorCI, 4 refactoring rules were implemented to refactor each of the 4 refactoring patterns mentioned previously. 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 (a trimmed out runnable layer of an operating system, used with technologies such as Docker and Vagrant) 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 in a separate branch, leaving the developers with the responsibility to either merge or delete the branch.

IMPLEMENTATION

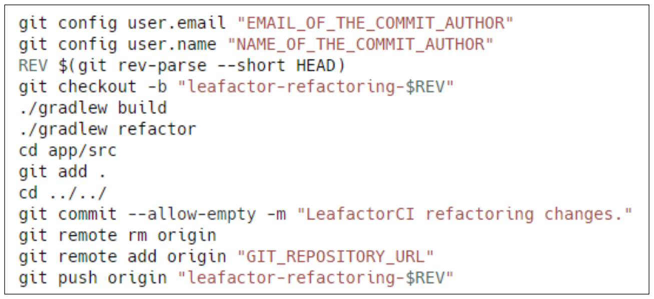
Spoon does not enforce any methodology for purging anti-patterns, in fact it simply lets us find, add, modify, and remove nodes in the AST (Abstract Syntax Tree). We can expect many possible cases to lead to an anti-pattern to form, essentially the disposition of variables and control flow in the code can complicate things. For instance, consider the Recycle anti-pattern, where 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 several cases of interest that need to be evaluated 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 4 phases is what we call a Refactoring Rule.

Testing

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 investigate 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.

Continuous Integration

Gradle does most of the job in supporting CI. A simple script was created such that it can be used as reference for integrating LeafactorCI. The script is shown in the code listing 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 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 branch. The branch name 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.



```
git config user.email "EMAIL_OF_THE_COMMIT_AUTHOR"
git config user.name "NAME_OF_THE_COMMIT_AUTHOR"
REV $(git rev-parse --short HEAD)
git checkout -b "leafactor-refactoring-$REV"
./gradlew build
./gradlew refactor
cd app/src
git add .
cd ../../
git commit --allow-empty -m "LeafactorCI refactoring changes."
git remote rm origin
git remote add origin "GIT_REPOSITORY_URL"
git push origin "leafactor-refactoring-$REV"
```

Code listing 1. CI template script.

EVALUATION

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

To dissimulate the understanding about the usage of energy practices and publicize LeafactorCI we 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?

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. 75.0% of them had worked on Android mobile applications while the rest did not. 56.2% have heard about energy bugs, at least 37.5% say they are a concern to them (there were ambiguous answers that we did not consider 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. After briefly being introduced to LeafactorCI, 78.6% said that they were willing to know more about it, however, only 63.6% said that they were willing to try it.

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.

A small experiment was also conducted with 3 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 3 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. Two of the participants reported that the difficulty was a one out of five and one participant reported that it was a two out of five. Two of the participants had to troubleshoot. Every single one of the participants was able to run the refactoring task. 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 (whose forks can be found in https://gist.github.com/moraispgsi/bc3eca2f92d3151bc85ac86f2248078e) were tried. From what we 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 lead us to an open issue. Other problems were related to Gradle version incompatibilities that were easily overcome. The installation was most of the time easy.

Outside of this user study a few other people have tried and successfully used LeafactorCI to see if there were problems in their project, to which as far as we know, no energy bugs were found.

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 can support the set of conditions present in the input files which account for the most common usage. In all 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.

To be used, LeafactorCI needs to be published and be readily available. During this work, 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 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 known issues and their respective states of resolution.

Now, to show that LeafactorCI can be setup in a CI environment, during this work, 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/f063e548bd2f770b de96b401096236ae6b8cf3af along with some other unrelated changes that were necessary to bring the project back to more modern versions. The changes were easy to make. It was set up such that whenever a commit is done to the codebase, a new branch is created and LeafactorCI is executed. A pull request could then be created to decide the branch's merge-ability.

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, we have yet to compile a list of system requirements and supported versions of Gradle and the Android SDK. we believe that 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 community support. Early adopters must 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 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 with 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 made as difficult as the developers want; it depends on the use case.

CONCLUSION

Motivation

This work 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.

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 \ac{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.

ACKNOWLEDGMENTS

We thank all the user study participants for their time and for their insightful feedback.

REFERENCES

1. 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. http://dl.acm.org/citation.cfm?doid=2897073.2897086.
2. 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-45BILESoft), pages 46–57, 2017. doi: 10.1109/MOBILESoft.2017.19. http://ieeexplore.ieee.org/document/7972717/.
3. Luis Cruz and Rui Abreu. Using Automatic Refactoring to Improve Energy Efficiency of Android Apps. 2018.

   http://arxiv.org/abs/1803.05889.
4. 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 Greenand Sustainable Software - GREENS 2014, pages 46–53, 2014. doi: 10.1145/2593743.2593750. http://dl.acm.org/citation.cfm?doid=2593743.2593750
5. Cruz, L., Abreu, R. & Lo, D. To the attention of mobile software developers: guess what, test your app!. Empir Software Eng 24, 2438–2468 (2019). https://doi.org/10.1007/s10664-019-09701-0
6. 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 Inter-national Conference on Automated Software Engineering - ASE 2016, pages 426–437, 2016. doi:10.1145/2970276.2970358. http://dl.acm.org/citation.cfm?doid=2970276.2970358
7. Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. Proceedings of the 201510th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015, pages 805–816,2015. doi: 10.1145/2786805.2786850. http://dl.acm.org/citation.cfm?doid=2786805.2786850
8. 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. International Conference on Automated Software Engineering, pages 60–71, 2017
9. Jean-No ̈el Rouvignac. Autorefactor. URLhttps://github.com/JnRouvignac/AutoRefactor.