ecoCode: a SonarQube Plugin to Remove Energy Smells from Android Projects

Olivier Le Goaer, Julien Hertout

ecoCode: a SonarQube Plugin to Remove Energy Smells from Android Projects

Olivier Le Goaër
Universite de Pau et des Pays de l'Adour
Pau, France
olivier.legoaer@univ-pau.fr

Julien Hertout
Snapp' Group
Bordeaux, France
jhertout@snapp.fr

ABSTRACT

To face the climate change, Android developers urge to become green software developers. But how to ensure carbon-efficient mobile apps at large? In this paper, we introduce *ecoCode*, a SonarQube plugin able to highlight code structures that are smelly from an energy perspective. It is based on a curated list of energy code smells likely to impact negatively the battery lifespan of Android-powered devices. The *ecoCode* plugin enables analysis of any native Android project written in Java in order to enforce green code.
— Demo video on https://youtu.be/4XIYGyPEhXQ

CCS CONCEPTS

• **Software and its engineering** → *Application specific development environments.*

KEYWORDS

android, energy, battery, smells, quality, debt

ACM Reference Format:
Olivier Le Goaër and Julien Hertout. 2022. ecoCode: a SonarQube Plugin to Remove Energy Smells from Android Projects. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3551349.3559518

1 INTRODUCTION

Climate change may not seem like an issue that should concern Android mobile developers, but the truth is that their work does have a carbon footprint. It is not only about instant over-consumption of energy at runtime but also about the limited number of charge-/discharge cycles of the battery that incidentally shorten the lifespan of handheld devices. Indeed, it is now well-known that most of the carbon footprint is emitted during the manufacturing of new terminals, and that this fast pace is no more sustainable.

Mobile developers, perhaps even more than other developers, *lack of knowledge on how to write, maintain, and evolve energy-efficient software* [2]. Whilst energy efficiency is becoming a major quality attribute, as is security or maintenability, we pinpoint the absence of lint-like tools to avoid poorly designed apps from an energy viewpoint. Linters have been used to address specific aspects of Android development like performance [3], permissions [5] or UI layout [6], and very recently for energy hotspots. Noticeable initiatives in that field are the EcoAndroid [4] plugin, a specific extension of Android Lint [7], the revised version of ADoctor [1] and the E-Debitum plugin [8]. Unfortunately, all these code analysis tools suffer from a limited number of detection rules and have not gone beyond the prototype stage.

Our work confirms the existence of a more significant number of android-specific energy smells that could be detected by code inspection. Energy smells can be defined as *structures with technical (and ecological) debt which affect energy consumption negatively.* Energy code smells imply the possibility of refactoring. In addition, among the static code analysers that work for Android projects, SonarQube is a world-class solution which can foster a rapid adoption by green developers worldwide. The result of this applied research is called *ecoCode* (stands for "ecological code"), and was released as an Open Source Software on GitHub[1] in January 2022. *ecoCode* has reached the Minimum Viable Product stage to attract early-adopter customers.

This paper is organized as follows: Section 2 outlines the empirical catalog of energy smells for Android. Section 3 describes how SonarQube was extended to cope with this new code smells, before we illustrate how it operates in Section 4. The final word is in Section 5.

2 ANDROID-SPECIFIC ENERGY SMELLS

Our work does not deals with the energy impact of classic object-oriented code smells [10, 11] or idioms [12, 13] on Android system, but rather with code patterns and API calls that have a vivid influence on the battery drain. It ranges from true battery-killers that must absolutely be avoided, to merely good practices.

2.1 Elements of Methodology

We yielded a catalog of smells by mixing several sources of knowledge. The primary and most indisputable of all is the Android API reference itself. We mined the online documentation (D) with the query `.*(energy|battery|power).*` to find energy-related hints, and refined the results. We also inspired from research (R) literature on that topic (e.g. [9]), and conducted interviews (I) of senior Android developers to get new understandings, during brainstorming workshops. The result is a premiere gran catalog of 40 energy smells tied to the Android platform.

However, despite their empirical evidence, some smells may suffers controversies, like for example the real benefit of dark UI on

[1] https://github.com/cnumr/ecoCode/tree/main/src/android-plugin

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559518

OLED screens. But what makes them very relevant is that they are android-specific, statically detectable and have multiple detection scopes (not only Java source code). We also found that energy smells can be arranged into 8 categories:

(1) `Optimized API`: refers to APIs have been specifically designed to perform some tasks in a more battery-efficient way than regular ones.
(2) `Leakage`: refers to resources acquisition without their release, which cause an unnecessary workload.
(3) `Bottleneck`: refers to accumulation of data or operations that requires an energy peak to be processed.
(4) `Sobriety`: refers to variants known as less energy-greedy, calling for a trade-off between the gain and the user experience.
(5) `Idleness`: refers to resources usage strategies when the app enters in an idle state.
(6) `Power`: refers to actions undertaken in regard of the battery-level and its internal management by the OS.
(7) `Batch`: refers to grouped execution of operations in order to optimize energy consumption
(8) `Release`: refers to compile-time adjustments before a large-scale deployment.

2.2 Empirical Catalog

Table 1 shows the source of knowledge the smell originated from, its category, its name and its detection scope within a native Android project. Due to place limitation, the detailed description of the catalog is available online[2].

The catalog simply lists the Android coding practices in a quite neutral way. Depending on the static detection strategy that will be chosen, some smells can be regarded as either good smells (saving energy) or bad smells (draining energy). This peculiarity is rare enough to be noted, in a code quality context where only bad smells are traditionally considered.

3 ECOCODE: A SONARQUBE PLUGIN

SonarQube, edited by SonarSource, allows to improve quality in software development projects. It is mainly focused on maintainability and security aspects but can be extended to address new concerns. Thus, the SonarQube framework allows writing custom coding rules to detect the aforesaid energy smells. We went one step further by customizing the GUI of the web-based dashboard.

3.1 Multi-scope Analysis

Unlike Android Lint for exemple, SonarQube is a general-purpose solution, unaware of the Android nature of project being analyzed. A challenge is that our catalog of smells implies a threefold scanning at once: Java files (source code), Xml files (manifest, certain types of resources) and Gradle files (build, settings, properties). The SonarQube framework provides built-in solutions to traverse Java files throughout an Abstract Syntax Tree, and Xml files throughout XPath expressions. As a side note, whilst Kotlin rules are available to end-user in the base SonarQube product, the Kotlin AST is not available for custom rules. Similarly, there is nothing to scan Gradle

[2] https://olegoaer.perso.univ-pau.fr/android-energy-smells/

Table 1: 40 Energy Smells for Android

Origin	Name	Scope
	`Optimized API (2)`	
D	Fused Location	Java, Gradle
D	Bluetooth Low-Energy	Java
	`Leakage (3)`	
D, R	Media Leak	Java
D, R	Sensor Leak	Java
I	Everlasting Service	Java
	`Bottleneck (4)`	
R	Internet In The Loop	Java
D	Wifi Multicast Lock	Java
R	Uncompressed Data Transmission	Java
D, R	Uncached Data Reception	Java
	`Sobriety (10)`	
R	Dark UI	Xml, Bitmap
D	Day Night Mode	Xml, File System
I	Brightness Override	Java
D	Thrifty Geolocation	Java
D	Thrifty BLE	Java
D	Thrifty Motion Sensor	Java
I	Thrifty Notification	Java
I	Vibration-free	Java
I	Torch-free	Java
D	High Frame Rate	Java
	`Idleness (6)`	
D	Keep Screen On	Java, Xml
D	Keep CPU On	Java
D, R	Durable Wake Lock	Java
D, R	Rigid Alarm	Java
D	Continuous Rendering	Java
D	Keep Voice Awake	Java
	`Power (4)`	
D	Ignore Battery Optimizations	Xml
D	Companion in background	Xml
D	Charge Awareness	Java, Xml
D	Save Mode Awareness	Java, Xml
	`Batch (3)`	
I	Service@Boot-time	Java, Xml
D	Sensor Coalesce	Java
D	Job Coalesce	Java
	`Release (8)`	
I	Supported Version Range	Xml
I	Same dependencies	Gradle
I	Duplicate dependencies	Gradle
I	Fat app	Gradle
D	Convert to WebP	File System
D	Clear cache	Java
D	Shrink Resources	Gradle
R	Disable Obfuscation	Gradle

files written in Groovy. That is why we embedded the Codenarc project[3] into the ecoCode plugin to leverage from its Groovy parser. Finally, scanning the project's structure in terms of directories and files (scope File System) is done with plain Java IO API.

3.2 Rules Implementation

The catalog is described regardless of the static code analysis tool to be used. In the present case of SonarQube, a given energy smell has sometimes had to be broken down into several rules, especially because target languages are managed separately. Basically, a custom rule is written in Java following the visitor-programming style. We written some helpers to avoid doing things the hard way. Helpers include checking for invocation of methods with a given signature, checking values behind constants, checking method calls are paired (e.g. `open()`/`close()` or `acquire()`/`release()`).

A rule is then associated with a tag which refers to one of the 8 category it belongs to. It is associated with a severity (among *Info*, *Minor*, *Major*, *Critical*) and a remediation cost that indicate the estimated time to fix the code. For lack of anything better, the severity *Info* is used to emulate a good smell while the 3 others are devoted to bad smells.

Last but not least, a rule is accompanied with a unit test at the plugin level that tests as many cases as possible. The plugin also comes with an integration test in the form of a mock Android project having no other purpose than violating all the rules at the same time.

3.3 Graphical User Interface

Energy efficiency being a novel quality attribute, we deemed it relevant to provide a tailored graphic interface, applying surface modifications to the extant web-based user interface. Behind the scene, this tedious work requires to locate key places into the ReactJS codebase to be altered. Thus, the menubar now adheres to the *ecoCode* look and feel, and more generally, icons and color scheme. Visible modifications include: home page, project cards, display of issues and view of a rule, overview page of rules, and detailed analysis of a project.

Advanced visual revamping is planned for a better user experience, but it requires to dive deep into the core concepts of SonarQube product. For example, by default, code smells are reported without any distinction whereas we have defined 8 categories and that rules have been tagged accordingly.

4 WALK THROUGH ECOCODE

SonarQube works on web side. Hence, from the developer side, he just has to upload his working project to the server spinning up the instance of *ecoCode*. It can also be done by branching *ecoCode* to Source Control Management tools like GitLab or GitHub (pipeline CI/CD).

4.1 Android Studio Setup

The considered Android project must have a dependency to the SonarQube plugin in the `build.gradle` file at the app level. Configuration of the project to be uploaded is set in the `gradle.properties`. This is where you fill in, for example, the host of the *ecoCode* instance and the access token.

The preferred way for pushing effortlessly the current project to the *ecoCode* server is to create a specific running configuration based on a Gradle task, thereby directly accessible from Android Studio panels and menus (Figure 1). This task have to be launched every time the developer wants a new analysis of his project.

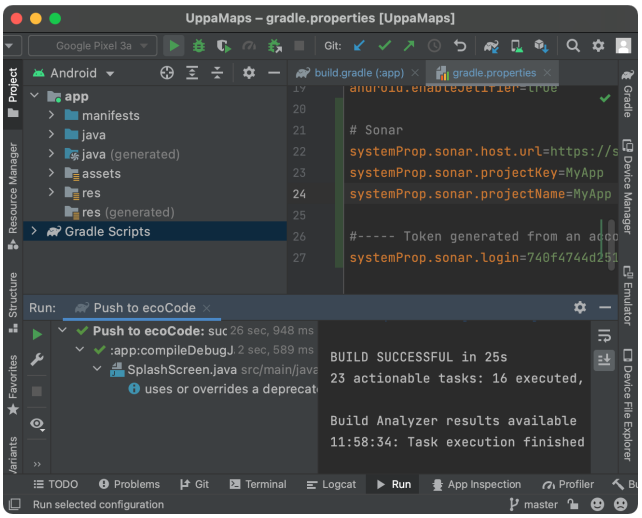

Figure 1: Pushing the *MyApp* project to *ecoCode* server through Android Studio (2021.1.1 version here) is a breeze.

4.2 Android Project Analysis

ecoCode comes with the specific quality profile *ecoCode way*, in addition to the default profile *Sonar way*. It means that the enforcement of green code adds up to the enforcement of clean code on any Android development project. Of course, each development team can adjust this profile based on what makes sense to him, as well as quality gate.

The developer then finds the functionalities it is used to with this kind of code quality tool. Especially, he can browse to the list of energy smells detected. A click on the issue shows the code snippet concerned. He can then see the rule detail which provides an explanation and illustrate both non-compliant and compliant code in order to guide the developer to fix the issue.

4.3 The *MyApp* Example

In this paper, *MyApp* refers to an open source Android app[4] with a codebase of 1.5k lines, which helps new students to find buildings on our university campus. A closer look at Figure 2 shows that 6 issues were detected on the main branch, either in Java or Xml files: 1 × Ignore Battery Optimizations, 4 × Fused Location, 1 × Vibration-free. It is worthwhile mentioning that a type of energy smell can be reported several times simply if it occurs in several places.

In the present case, the app failed the green quality gate (see Figure 3) that states no more than 4 issues (by default). It means

[3] https://codenarc.org/

[4] https://git.univ-pau.fr/summer-school/uppamaps

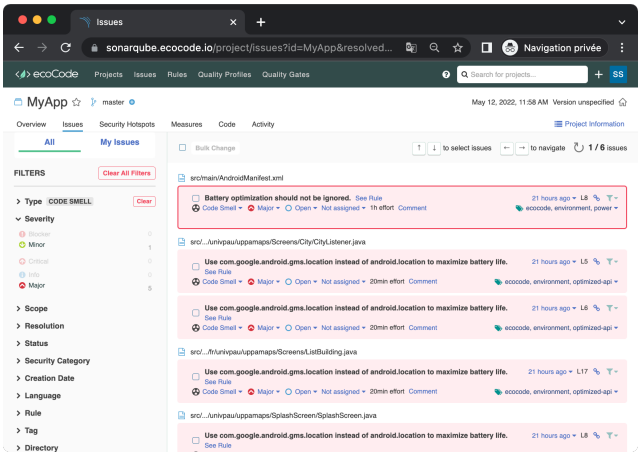

Figure 2: Custom web page reporting the energy smells detected in the project *MyApp*.

that the project should be reworked before it is published on Google Play Store and thus prior to be run on a potential huge number of devices. Technical debt is estimated to 2h25 min. Actually, since a smell is repeated 4 times, it is likely that the remediation step will be more rapid.

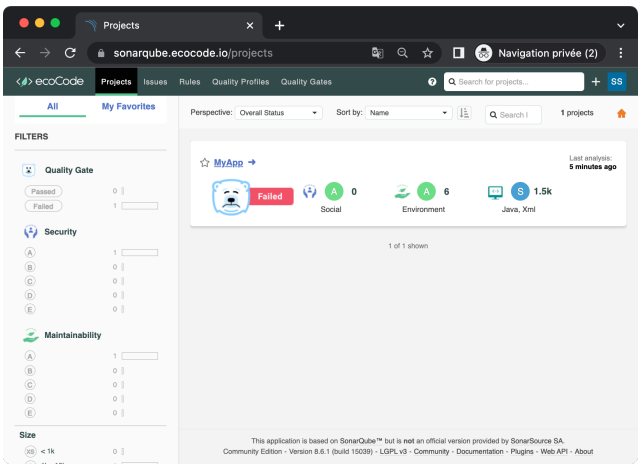

Figure 3: Custom web page showing the environmental status of *MyApp* (Social status is a future work).

5 CONCLUSION

ecoCode empowers all native android developers to write greener code. At the time of writing, three-quarters of the catalog of energy smells have been implemented into the plugin. The extension of the catalog of smells pursue on one side, and their concrete implementations on the other side. We are expecting a lot from events like hackathons to engage as many people as possible, and not only developers.

ecoCode is still in its infancy and debates have only just begun about the severity of the rules (i.e. their measurable impact), their occurrence in real-world Android projects, and their remediation cost. Yet, it provides a free, cutting-edge tool to engineers and researchers to conduct a broad panel of empirical software engineering studies.

ACKNOWLEDGMENTS

ecoCode is a joint work between Université de Pau et des Pays de l'Adour and Snapp', a company specialized in mobile apps realisations. The project was funded by a grant from Region Nouvelle Aquitaine (France).

REFERENCES

[1] Iannone, E., Pecorelli, F., Nucci, D.D., Palomba, F., Lucia, A.D. (2020). "Refactoring Android-specific Energy Smells: A Plugin for Android Studio". Proceedings of the 28th International Conference on Program Comprehension.
[2] Gustavo Pinto and Fernando Castor (2017). "Energy efficiency: a new concern for application software developers". Commun. ACM 60, 12 (December 2017), 68–75.
[3] Sarra Habchi, Xavier Blanc, and Romain Rouvoy. (2018). "On adopting linters to deal with performance concerns in Android apps". Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. Association for Computing Machinery, New York, NY, USA, 6–16.
[4] A. Ribeiro, J. F. Ferreira and A. Mendes, (2021). "EcoAndroid: An Android Studio Plugin for Developing Energy-Efficient Java Mobile Applications," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 62-69
[5] Sowndarajan, Karthick & Binu, Sumitra. (2017). "Static Analysis Tool for Identification of Permission Misuse by Android Applications". International Journal of Applied Engineering Research.
[6] Suelen Goularte Carvalho et al. (2019). "An Empirical Catalog of Code Smells for the Presentation Layer of Android Apps". Empirical Software Engineering.
[7] Olivier Le Goaër. (2020). "Enforcing Green Code With Android Lint,". 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pp. 85-90
[8] Daniel Maia, Marco Couto, João Saraiva, and Rui Pereira. (2020). "E-Debitum: managing software energy debt". Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops. Association for Computing Machinery, New York, NY, USA, 170–177.
[9] Luis Cruz and Rui Abreu. (2019). "Catalog of energy patterns for mobile applications". Empirical Software Engineering. 24, 4, 2209–2235.
[10] Oliveira, J., Viggiato, M., Santos, M.F., Figueiredo, E., and Marques-Neto, H.T. (2018). "An Empirical Study on the Impact of Android Code Smells on Resource Usage". In Proceedings of The Thirtieth International Conference on Software Engineering and Knowledge Engineering (SEKE 2018).
[11] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M. Almurshed, Danny Dig, and Carlos Jensen. (2016). "Understanding code smells in Android applications". In Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft '16). ACM, New York, NY, USA, 225–234.
[12] Palomba, Fabio & Di Nucci, Dario & Panichella, Annibale & Zaidman, Andy & Lucia, Andrea. (2018). "On the Impact of Code Smells on the Energy Consumption of Mobile Applications". Information and Software Technology.
[13] Carette, Antonin & Younes, Mehdi & Hecht, Geoffrey & Moha, Naouel & Rouvoy, Romain. (2017). "Investigating the energy impact of Android smells". 115-126. 10.1109/SANER.2017.7884614.