Refactoring Android-specific Energy Smells:
A Plugin for Android Studio

Emanuele Iannone
SeSa Lab - University of Salerno
Fisciano (SA), Italy
e.iannone16@studenti.unisa.it

Fabiano Pecorelli
SeSa Lab - University of Salerno
Fisciano (SA), Italy
fpecorelli@unisa.it

Dario Di Nucci
JADE Lab - University of
Tilburg/JADS
's-Hertogenbosch, The Netherlands
d.dinucci@uvt.nl

Fabio Palomba
SeSa Lab - University of Salerno
Fisciano (SA), Italy
fpalomba@unisa.it

Andrea De Lucia
SeSa Lab - University of Salerno
Fisciano (SA), Italy
adelucia@unisa.it

ABSTRACT

Mobile applications are major means to perform daily actions, including social and emergency connectivity. However, their usability is threatened by energy consumption that may be impacted by code smells *i.e.,* symptoms of bad implementation and design practices. In particular, researchers derived a set of mobile-specific code smells resulting in increased energy consumption of mobile apps and removing such smells through refactoring can mitigate the problem. In this paper, we extend and revise ADOCTOR, a tool that we previously implemented to identify energy-related smells. On the one hand, we present and implement automated refactoring solutions to those smells. On the other hand, we make the tool completely open-source and available in ANDROID STUDIO as a plugin published in the official store. The video showing the tool in action is available at: https://www.youtube.com/watch?v=1c2EhVXiKis

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

Code smells, Refactoring, Energy Consumption.

1 INTRODUCTION

Mobile applications (*a.k.a.* apps) have gained great popularity in recent years. Developing a successful app requires to take into account different contrasting constraints such as user experience, performance, privacy, and energy consumption. One of the leading causes of a quick battery drain is the hardware setup, *i.e.,* the battery capacity, and the quality of the components. However, recent studies have shown that even wrong source code implementation decisions may decrease energy efficiency. For instance, Hasan *et al.* [5] stated that choosing a wrong data structure can lead to up to 300% energy waste. Palomba *et al.* [2, 10] analyzed the impact of some Android-specific code smells and discovered that their presence has a notable impact on battery usage, while Hecht *et al.* [6] later confirmed these findings and report that those smells also affect apps performance. These works based their analyses on a particular subset of the Android-specific code smells defined by Reimann's *et al.* [11] called "energy-smells" (*i.e.,* smells that have an impact on energy consumption) and were supported by their energy smell detection tools: PAPRIKA and ADOCTOR, respectively.

On the one hand, ADOCTOR (AnDrOid Code smell detecTOR) [9] identifies 15 Android-specific code smells. It syntactically analyzes the source code by extracting the Abstract Syntax Trees from the classes (*i.e., Java* files) and then runs the detection algorithms which are based on the Visitor pattern. On the other hand, PAPRIKA [6] works at the byte-code level and currently detects 16 code smells (both Object-Oriented and Android-specific). The empirical experimentation conducted to assess the accuracy of these tools showed that both can reach a high precision and recall; nevertheless, none of them is still able to provide developers with mechanisms to (1) **identify** refactoring opportunities and (2) **automatically refactor** the code to remove code smell instances.

Indeed, while most of these energy smells can be removed by applying simple program transformations, *e.g.,* adding a single statement or a keyword into a method signature, practitioners still tend not to refactor their code because (i) they are not aware of the existence of certain code smells and/or cannot estimate their impact [8]; (ii) they perceive the refactoring as a fault-prone activity [1, 4, 7]. For these reasons, an automated energy smell refactoring tool has the potential to be used in practice to help developers dealing with energy consumption.

To reach this goal, in this paper we enhance our previously proposed tool ADOCTOR by (i) devising and applying refactoring solutions for the detected Android-specific energy smells; (ii) integrating it as a plugin for ANDROID STUDIO, *i.e.,* the reference IDE

for Android native app development, and (iii) making it publicly available in the official *JetBrains Plugin Repository*.[1]

2 SMELL DETECTION AND REFACTORING

The original version of ADoctor was able to detect 15 different Android-specific code smells, five of which were related to energy consumption, i.e., Durable WakeLock, Inefficient Data Structure, Internal Setter, Leaking Thread and Member Ignoring Method. This new version can identify and remove these energy-related code smells. Since the tool can also apply refactoring operations, we updated the acronym of the tool, which now stands for AnDrOid Code smell detecTiOn and Refactoring. In the following sections, we present a usage scenario (Section 2.1) followed by a brief explanation of the detection rule and refactoring action implemented for each of the considered code smells (Sections 2.2 to 2.6) according to Reimann's catalogue [11].

2.1 How to Use aDoctor

ADoctor is available in the *JetBrains Plugin Repository*, the repository that contains the plugins for the JetBrains IDEs (*e.g.,* IntelliJ IDEA and Android Studio).

Potentially, any Java class can be analyzed by ADoctor if syntactically correct. To launch the plugin, the developers have to navigate the *Refactor* menu and select *ADoctor*. From the main dialog window, the developers can select (i) the energy smells to detect and (ii) the package to analyze. By default, ADoctor detects all the five smells mentioned above, starting from the root package.

After clicking the *Start* button, the detection algorithms are executed on the Abstract Syntax Trees extracted from the source classes. Although the analysis is quite rapid (taking no more than a minute on projects with multiple modules), it can be aborted by the user before ending. Once this phase is completed, the results are shown as depicted in Figure 1. The first column presents a combo box containing all the detected smell instances, while the second and the third ones show the "diff" panel: the former exhibits the current version of the source code, the latter the proposed refactored version. In this way, the user can analyze how the source code would be changed and confirm the operation by clicking on *Apply*. In this case, the final step rewrites the *Java* classes according to the refactoring algorithm. Note that only a single smell instance at the time can be selected for refactoring.

2.2 Durable WakeLock

Definition. To avoid unnecessary battery consumption, an idle Android device goes on standby, *i.e.,* starts to dim the screen and then disables the CPU. When an app needs to keep the CPU active to complete some background work, the Android API provides "wake-locks" that can be acquired to keep the device awake. A wake lock should be acquired only when necessary, so it is a good practice to either specify a release timeout or release it explicitly.

Identification. A class has a Durable WakeLock (DW) smell when it has a method with an instance of `PowerManager.WakeLock` (either declared locally or at instance level) that acquires a wake

lock without setting a release timeout and without subsequently calling `release()` within the same scope.

Refactoring. A DW smell can be fixed by adding a `release()` call statement at the end of the source code block where the smelly `PowerManager.WakeLock` instance calls the `acquire()` method.

2.3 Inefficient Data Structure

Definition. A `HashMap`'s key type parameter can be any `Object` subclass, typically primitive types wrapper classes, like `Integer`. Almost all method calls on a `HashMap` let the Android RunTime (ART) to apply the autoboxing continuously and unboxing (the automatic two-way conversion between primitive type with their corresponding wrappers), that determines a non-trivial computational overhead.

The Android API offers the `SparseArray` class, that acts similarly to a `HashMap<Integer, Object>` but the key is always a primitive `int` type, instead of an `Integer`—an `int` variable requires much less memory than an `Integer` object. There are some drawbacks about `SparseArray`: it is only Android-specific, (so the portability to other Java platforms is reduced); it has a quite different API with respect to `HashMap`; the data structure does not scale well, implying that a `HashMap` with over 1000 items has generally better performance over a `SparseArray` of the same size [3] [5].

Identification. A class has an Inefficient Data Structure (IDS) smell when it declares a `HashMap` local variable whose first type argument (*i.e.,* key) is an `Integer`.

Refactoring. An IDS smell can be fixed by changing the variable declaration type from `HashMap<Integer, X>` (where X is any subclass of `Object`) to `SparseArray<X>`. Afterwards, every method call done by each of the variable contained into the smelly variable declaration (*i.e.,* `HashMap<Integer, X>`) should be changed to the `SparseArray` equivalent ones. Finally, in case the `SparseArray` class has not been imported in the class, the related import statement should be added.

2.4 Internal Setter

Definition. Setter methods are a fundamental component of Object-Oriented programming. They usually accept a single argument that is assigned to an instance variable; optionally, they might add a precondition check before this assignment. A non-static method of the same class that calls a setter of this kind (*i.e.,* with only a single assignment) makes a useless computational effort because it has the access rights to make a direct assignment on that property, possibly causing an energy loss.

Identification. A class has an Internal Setter (IS) smell if one of its methods calls an internal setter.

Refactoring. An IS smell can be fixed by using a direct instance variable assignment instead of a setter method call; nevertheless, the setter method is not removed completely, so that public accesses to it are preserved.
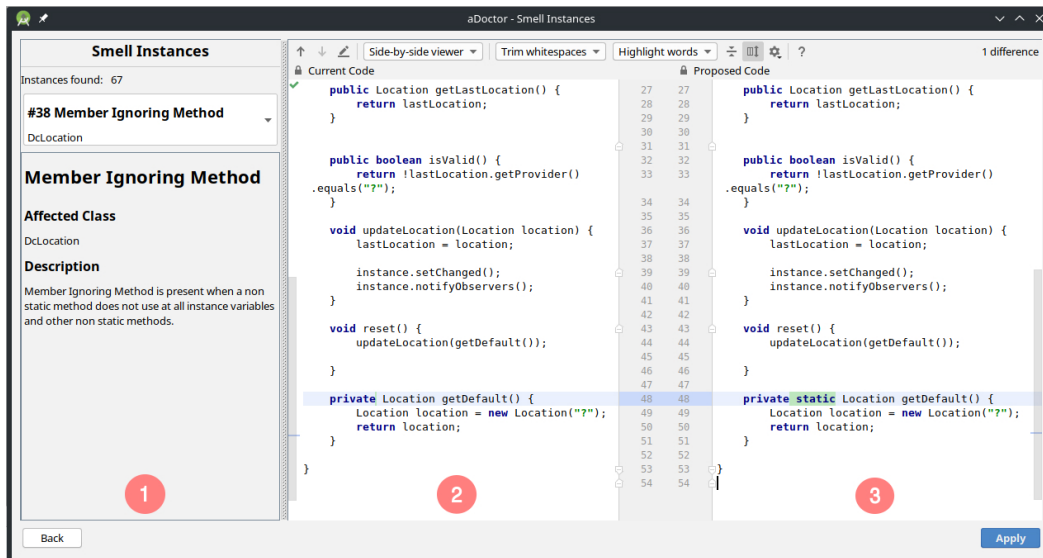
---

[1]https://plugins.jetbrains.com/plugin/13443-adoctor/

Figure 1: Smell dialog where the analysis results and refactoring proposals are shown.

2.5 Leaking Thread

Definition. The ANDROID RUNTIME (ART) treats an active `Thread` instance as a Garbage Collector (GC) root, meaning that its memory cannot be reclaimed. Whenever a `Thread` is stopped (by calling `stop()` or `interrupt()`), it ceases to be treated as a GC root, becoming eligible for garbage collection. Note that a fully executed `Thread` is not considered stopped until `stop()` or `interrupt()` is called: this causes memory waste.

Identification. A subclass of an `Activity` has a LEAKING THREAD (LT) smell when it exhibits a `Thread` instance variable that calls `start()` in a class method but not `interrupt()` in any of the class methods.

Refactoring. A LT smell can be fixed by calling the `interrupt()` method for the leaking `Thread` in the `onDestroy()` callback. If `onDestroy()` does not exist in the source code of the class, it should be added.

2.6 Member Ignoring Method

Definition. According to the Java Memory Model, a static method is faster than its equivalent non-static one: mainly because the caller object `this` reference is not passed to static methods, so the reference resolution does not take place. A static method does not access any internal properties (*i.e.,* instance variables and non-static methods). Therefore, if a non-static method does not access any internal properties of its belonging class, it should be set as a static one.

Identification. A class has a MEMBER IGNORING METHOD (MIM) smell when it has a non-static and non-empty method that (i) does not access any instance variable; (ii) does not use `this` and `super` keywords; (iii) does not override an inherited method.

Refactoring. A MIM smell can be fixed by adding the `static` keyword to the smelly method signature.

3 TOOL ARCHITECTURE

The plugin infrastructure has been implemented with the INTELLIJ PLATFORM SDK, which makes available a set of libraries that allows the extension of the INTELLIJ PLATFORM by creating plugins, custom languages support or a custom IDE. In this work we used it to integrate ADOCTOR as a plugin for ANDROID STUDIO and to get access to INTELLIJ PLATFORM's menus and built-in tools (like diff) to (i) launch the plugin, (ii) fetch all project files, and (iii) use the diff to show the refactoring proposals.

Figure 2 shows the tool architecture, which is a two-layer model. The *Presentation* layer contains both the presentation and control logic, mainly in the *Dialog* subsystem. INTELLIJ PLATFORM provides good support for *Swing* components, letting the plugins have the same look-and-feel of the host IDE; for this reason, the presentation logic is entirely composed of *Swing* dialog windows. The global control logic is centralized in a single class, called `CoreDriver`, that selects the right dialog to display and moves data among them through callback interfaces.

The *Application* layer contains the whole business logic. The *Analysis* subsystem performs the smell detection; the *Proposal* subsystem receives the smells data and prepares the refactoring proposals; the *Refactoring* subsystem refactors the code if developers accept a proposal. ADOCTOR does not need to store any persistent data, so, through INTELLIJ PLATFORM SDK, it only accesses the local file system for reading and writing on source files.

Since its first version, ADOCTOR has been based on ECLIPSE JDT CORE library for parsing Java files and extracting their Abstract Syntax Trees; for this reason, we decided not to change this dependency to reuse the logic of the detection algorithms—thus, being also sure to preserve their accuracy.

Finally, the *Analytics* subsystem gathers some usage statistics, such as the type of smell selected by the user during the analysis and refactoring phases. These data will be used in future works to investigate the practical usefulness of the tool in the wild.
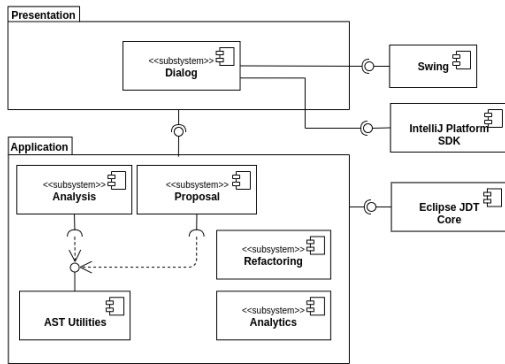
Figure 2: Overview of the ADOCTOR architecture.

4 NOTES ON EVALUATION

Since ADOCTOR provides both identification and refactoring of energy smells, we report two different strategies for its evaluation.

4.1 Evaluating Energy Smell Identification

The identification algorithms of all the smells mentioned above have already been evaluated in the study by Palomba *et al.* [9], in which the tool achieved an average F-measure close to 100%. The previous empirical study required a human oracle due to the absence of a dataset for the aforementioned energy smells in literature. The authors collected 18 different apps of different scope and size and asked a master student of the University of Salerno, having experience with Android development, to seek the presence of energy smells in the code; a second master student of the same university validated the produced oracle so that some of the false positives were removed. Finally, the smell set built from the oracle was compared to the candidate set built by ADOCTOR to compute the precision and recall scores.

4.2 Evaluating Energy Smell Refactoring

Most of the refactoring techniques are concerned with small program transformations that do not change the external behaviour of the target class: for instance, the MIM code smell is removed by adding the keyword `static` to the signature of the affected method. This, of course, changes the way the method is called but not its external behaviour.

The only exception is represented by the refactoring of INEFFICIENT DATA STRUCTURE, which is generally more complex: it requires the update of all method calls previously done on a `HashMap`. Being a set of one-by-one replacements of each `HashMap` method calls, we expect no changes in the external behaviour of refactored classes. To confirm this assumption, we plan an early evaluation that (1) exercises the apps with the available test suites before and after having applied the automated refactoring of IDS instances and (2) compares the test outputs. Afterwards, we plan another evaluation involving both industrial and academic developers to assess refactoring performance better.

Our next purpose is to evaluate the tool's statistical significance: understanding to what extent the tool is considered useful for Android developers. We published the plugin in the *JetBrains Plugins Repository*, and we are collecting usage data on which kind of smell developers choose to detect and, in case, refactor.

Finally, having a good evaluation of refactoring performance can reduce developers' concerns about automatic techniques, as explained in [7].

5 CONCLUSION

In this demo, we presented an extended version of ADOCTOR. The main contributions can be summarized as follows:

- ADOCTOR, a tool integrated into Android Studio [11] to identify and refactor five Android-specific energy smells;
- A detailed explanation of its detection and refactoring features;
- Details about the evaluation of the tool performance and its utility in real scenarios;

As future work, we plan to implement additional energy smell detection and refactoring and make the tool able to return more than one smell instance per class.

REFERENCES

[1] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When Does a Refactoring Induce Bugs? An Empirical Study. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM '12)*. IEEE Computer Society, Washington, DC, USA, 104–113. https://doi.org/10.1109/SCAM.2012.20
[2] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea Lucia. 2017. PETrA: A Software-Based Tool for Estimating the Energy Profile of Android Applications. https://doi.org/10.1109/ICSE-C.2017.18
[3] Google. 2019. *SparseArray*. https://developer.android.com/reference/android/util/SparseArray
[4] Sarra Habchi, Romain Rouvoy, and Naouel Moha. 2019. On the Survival of Android Code Smells in the Wild. In *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft '19)*. IEEE Press, Piscataway, NJ, USA, 87–98. http://dl.acm.org/citation.cfm?id=3340730.3340749
[5] Samir Hasan, Zachary King, Hafiz Suliman Munawar, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy profiles of Java collections classes. 225–236. https://doi.org/10.1145/2884781.2884869
[6] G. Hecht, N. Moha, and R. Rouvoy. 2016. An Empirical Study of the Performance Impacts of Android Code Smells. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 59–69. https://doi.org/10.1109/MobileSoft.2016.030
[7] M. Kim, T. Zimmermann, and N. Nagappan. 2014. An Empirical Study of RefactoringChallenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (July 2014), 633–649. https://doi.org/10.1109/TSE.2014.2318734
[8] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014* (12 2014), 101–110. https://doi.org/10.1109/ICSME.2014.32
[9] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. 2017. Lightweight detection of Android-specific code smells: The aDoctor project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 487–491. https://doi.org/10.1109/SANER.2017.7884659
[10] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea Lucia. 2018. On the Impact of Code Smells on the Energy Consumption of Mobile Applications. *Information and Software Technology* (09 2018). https://doi.org/10.1016/j.infsof.2018.08.004
[11] Jan Reimann, Martin Brylski, and Uwe Assmann. 2014. A Tool-Supported Quality Smell Catalogue For Android Developers.