Cross-Detection of Mobile-specific Energy Hotspots: MBSE to the Rescue

Léa Brunschwig, Olivier Le Goaër

▶ **To cite this version:**

Léa Brunschwig, Olivier Le Goaër. Cross-Detection of Mobile-specific Energy Hotspots: MBSE to the Rescue. 1st International Workshop on Sustainability and Modeling co-located with MoDELS'24, Sep 2024, Linz, Austria. pp.518-522, 10.1145/3652620.3687797 . hal-04743948

HAL Id: hal-04743948

https://hal.science/hal-04743948v1

Submitted on 18 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Cross-Detection of Mobile-specific Energy Hotspots: MBSE to the Rescue

Léa Brunschwig
Université de Pau et des Pays de l'Adour
Pau, France
lea.brunschwig@univ-pau.fr

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

ABSTRACT

Regarding mobile applications (or apps), energy efficiency is becoming as important a quality attribute as security. One interesting approach is to automatically pinpoint energy hotspots, i.e., areas in the code base of an Android or iOS project that may negatively impact battery life. The basic principle is to statically analyze the input source code based on a growing catalogue of mobile-specific energy code smells or anti-patterns. Although some anti-patterns overlap across Android and iOS, detection strategies must be implemented from scratch for each mobile platform and for each code analyzer. This situation is not sustainable in the medium term in the race to develop environmentally friendly mobile apps. This paper demonstrates how the MBSE can address this industrial use case.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

Code Smell, Energy, Mobile app, Static analysis

ACM Reference Format:
Léa Brunschwig and Olivier Le Goaër. 2024. Cross-Detection of Mobile-specific Energy Hotspots: MBSE to the Rescue. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3652620.3687797

1 INTRODUCTION

Model-Based Software Engineering (MBSE) quickly became of interest in the field of mobile applications due to the heterogeneity of the underlying platforms. That was quite true ten years ago during the OS war, and it's still true today with the two remaining platforms – Android and iOS – which together account for 99% of the market. Over the years, the research literature has focused on producing native code for both platforms from a single code base, following the principles of MDA/MBSE like in [1, 7, 13]. There is no doubt that this research craze has declined with the arrival of mature and shiny cross-platform solutions such as Flutter, React Native and Kotlin Multiplatform Mobile.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MODELS Companion '24, September 22–27, 2024, Linz, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687797

Yet, the story doesn't end there: the heterogeneity challenge strikes again when it comes to detecting and fixing flaws in the source code of Android or iOS native projects. Since suboptimal coding choice for energy consumption at runtime is considered a defect, this duplicate effort is becoming very important for developers that target both platforms (i.e. a large majority). It is important to note that submitting the codebase to a lint-type tool is a widespread practice for improving the overall quality of the code delivered by teams. Doing it for "green quality" is a new trend driven by a climate-conscious tech landscape.

Intuition tells us that there are energy-related flaws (or anti-patterns) that are the same from one platform to another and that it should be possible to detect them, even if Android and iOS are different, in terms of languages used and APIs provided. Unfortunately, hand-developed detection rules are a complex piece of engineering and, hence, a tedious task.

In this research paper, we introduce domain-specific languages (DSLs) designed to describe code smells and map them with the mobile language of choice, ultimately generating detection rules for the static analysis tools of our choice.

The remainder of this paper follows this organization: Section 2 lays out the theoretical and practical foundations for this study. Section 3 presents a motivational example, and Section 4 outlines the development of meta-models for describing and translating energy code smells across development environments. Finally, we contrast our approach with similar works in Section 5 and outline conclusions and future work in Section 6.

2 BACKGROUND

Energy code smells are surface symptoms, indicating that something is potentially wrong with energy efficiency. They imply that the app's source code could be improved or that additional effort could be put into it. If they are well-defined, it is possible to review the entire codebase automatically and highlight them, so-called energy hot spots, thus requiring the developer's attention.

2.1 Mobile-Specific Energy Code Smells

An extensible catalogue of mobiles-specific energy code smells was yielded by O. Le Goaer as a digital common [8]. This work drew significant inspiration from the 22 energy patterns for mobile applications of Cruz et al. [3] but with the special objective of turning good/bad practices into statically detectable code smells. The empirical catalogue now provides more than 40 energy code smells, divided into eight categories, and targets both mobile platforms. Some code smells go beyond the energy concern (i.e. environmental concerns), but for the sake of simplicity, this paper will only focus

Code Smell Name	Common Practice
	LEAKAGE
Sensor Leak	Always remember to unsubscribe from a sensor once you have subscribed to it to avoid wasting data acquisition
	IDLENESS
Keep Screen On	Never prevent the device from going to sleep after a certain time to avoid draining the battery in just a few hours
Rigid Alarm	Applications are strongly discouraged from using exact alarms unnecessarily as they reduce the OS's ability to minimize battery use
	POWER
Charge Awareness	Adapt workload accordingly when the device is connected/disconnected to a power station or switch to a different battery level
Save Mode Awareness	Adapt workload accordingly when energy save mode is activated intentionally by the end-user or by the system
	SOBRIETY
Thrifty Geolocation	Configure the geolocation sensor (aka GPS) in a less accurate mode and with a lower position update rate
Dark Mode	Dark themes should be preferred to light themes as this can affect the AMOLED display under certain conditions
Brightness Override	Don't override the screen brightness value, which automatically adjusts to ambient light to save energy
Animation-free	Avoid extraneous animations, which consume a lot of power as they require the CPU, GPU and screen to be active
Torch-free	Don't programmatically activate the LED flashlight, a notoriously power-hungry component

Table 1: Energy code smells shared by both mobile platforms.

on energy code smells. The interesting point to notice is that a subset of code smells is common to both platforms. Table 1 keeps the 10 code smells that intersect both platforms, describing commonly shared practices for energy savings and falling into the *leakage*, *idleness*, *power* and *sobriety* categories. At this level of abstraction, each code smell has an evocative name and a platform-independent and hence scanner-independent description.

2.2 Energy Code Smells Detection

The detection technique can range from a simple regular expression in the textual source code to a more robust solution based on an abstract syntax tree (AST). This approach is chosen by static code analysis tools on the market. They embed code scanners specific to the target languages and everything needed to report defects to the individual developer or team, ultimately leading to their resolution. We can mention IDE-based solutions such as Android Lint, Ktlint, SwiftLint, or code quality tools such as PMD, Checkstyle, Semgrep, SonarQube or even GitHub CodeQL.

Such tools have historically focused on maintainability and/or vulnerability, but their scope is just beginning to shift to sustainability, which includes energy concerns, as in [4, 12]. This is also the special purpose of pioneering plugins for the world-class solution SonarQube for pinpointing the energy code smells mentioned above [6]. The Android Java and iOS Swift plugins are already available while the Android Kotlin plugin is under construction, while the iOS Objective-C plugin has been dismissed. This fragmentation, which we are facing in the single case of SonarQube, is exacerbated when targeting further tools. Ideally, to reach the maximum number of mobile developers and align with their practices, all of them should be targeted.

3 MOTIVATING EXAMPLE

This section will argue for leveraging model-driven engineering, recognized for offering abstraction, automation, adaptable models, enhanced productivity, and enforcement of best practices. To justify the necessity of MBSE in defining energy code smells for mobile devices, we will illustrate our point by drawing on the example of using the flashlight. Indeed, this feature is specific to smartphones or tablets and is not commonly addressed by the general-purpose energy code smells dealing with computers or servers. The summary of our motivations is depicted in Figure 1, and the statement of our mobile energy code smell example is as follows (cf. Table 1):

> *"Don't programmatically activate the LED flashlight, a notoriously power-hungry component"*

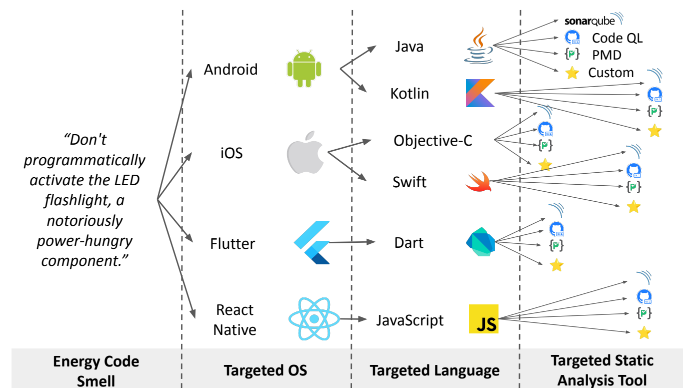

Figure 1: Ramification of a energy code smell at different levels of abstraction.

The heterogeneity of mobile device operating systems poses the initial challenge for developers, a challenge that is equally familiar to code smell experts, who must know the different platforms. The first hurdle entails choosing between developing a native application, requiring a decision between Google's Android and Apple's iOS platforms, which induces coding the app twice. Alternatively, developers may choose cross-platform technologies like Flutter or React Native if native functionality is not essential. These considerations are integral to the definition of energy code smells.

As illustrated in Figure 1, this complexity leads to four distinct pathways, each offering a potential approach to addressing a single energy code smell. However, it’s important to emphasize that this list is not exhaustive. In our example, the flashlight feature is available in native and cross-platform solutions.

Each mobile operating system or cross-platform technology is associated with specific programming languages. For Android, applications must be written in Java or Kotlin, while for iOS, developers use Objective-C or Swift. This diversity in programming languages adds further complexity to Figure 1, as each language introduces additional branches, resulting in many versions needed to express the energy code smell for mobile apps. In our scenario, when dealing with Android Java, turning on the flashlight with `CameraManager#setTorchMode(..., true)` must be avoided. Similarly, in iOS Swift, it is essential to refrain from activating the flashlight using `AVCaptureDevice#setTorchMode(level:)` or `AVCaptureDevice#torchMode`.

Static code analysis examines the source code of a program without executing it. Its purpose is to detect code smells previously defined for a specific language. Several tools are available for that purpose (cf. Section 2.2). It highlights the need to create multiple versions of a single mobile energy code smell to detect it across applications and static analysis tools. Figure 1 illustrates 24 distinct branches from a single code smell root.

This paper proposes an approach for defining mobile code smells at the highest abstraction level for maximum reuse. The detection rule for an energy code smell would be generated by selecting a mobile platform, the corresponding programming language, and a target static analysis tool. With the proposed approach, the detection rules can be automatically regenerated each time an energy code smell is created or updated, or as mobile technologies or code quality tools evolve. Our approach will adhere to the following requirements:

- R1: Energy code smells should be described at the highest abstraction level, independent of any specific programming language.
- R2: Energy code smells should be defined once and reused for as many mobile code translations as necessary.
- R3: Translations into a targeted language should be independent of any specific static analysis tool.
- R4: The definition of an energy code smell, its translation, and the code generation model for a static analysis tool should be achievable iteratively and by different actors.

To address these requirements, we propose two DSLs as stated in Figure 2. The first DSL will enable the definition of a catalogue of energy code smells following R1 and, transitively, R3. The second DSL will extend the first one to fulfil R2, allowing for distinctions such as Android Java or iOS Swift and more. Code to be utilized within a static analysis tool will be generated via Model-to-Code transformation, fulfilling R3. Furthermore, as depicted in Figure 2, R4 will be addressed by facilitating collaboration among experts, enabling end-users to analyze their code using the energy code smells catalogue. Ideally, a code smell expert will handle the task of describing and categorizing mobile energy code smells using the Energy Code Smell DSL. Mobile development experts will then translate these smells from the catalogue into specific mobile languages using the second DSL, which extends the first one. Finally,

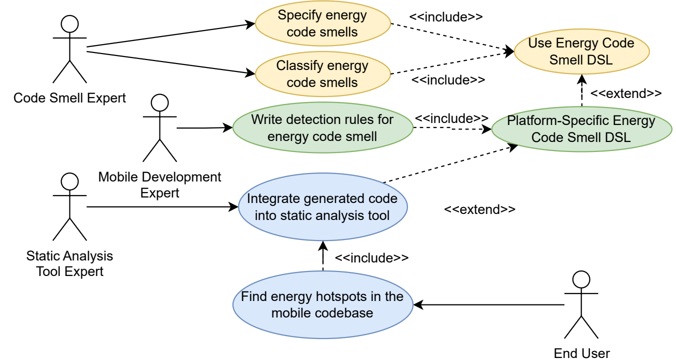

Figure 2: Use cases and actors in the proposed approach.

a static analysis tool expert will outline the template for generating any code smell within the targeted static analysis tool.

4 PROPOSED APPROACH

The implementation of energy code smells can result in multiple versions because of specific languages or platforms. In response to the heterogeneity of energy code smells, we propose two meta-models and a model-to-code transformation. This section describes the abstract and concrete syntax of these meta-models and the envisioned process for generating the final code.

4.1 Abstract Syntax

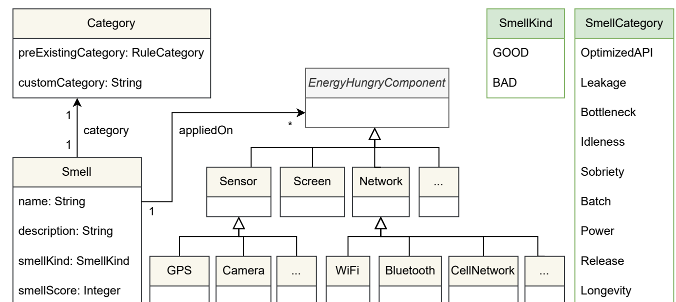

Figure 3: Energy Code Smell meta-model.

We have created two meta-models. The first one in Figure 3 is called the Energy Code Smell DSL. It permits the description of a mobile-specific smell *at the highest abstraction level, independent of any specific programming language (R1)*. Each smell has a name and a description, aligning with the two columns of Table 1. Although a smell is usually a synonym for bad practices, we argued that there could be good smells too (cf. [8]): a code smell can cause more or less damage (or benefit); hence, we can give a score which will allow classifying the importance of a smell compared to another.

One model conforming to the meta-model will contain one single smell, but several models compose a catalogue of smells. Consequently, we offer to specify a smell with two elements: the categories we have cited in Table 1 and the hardware-related components that are involved in battery drain.

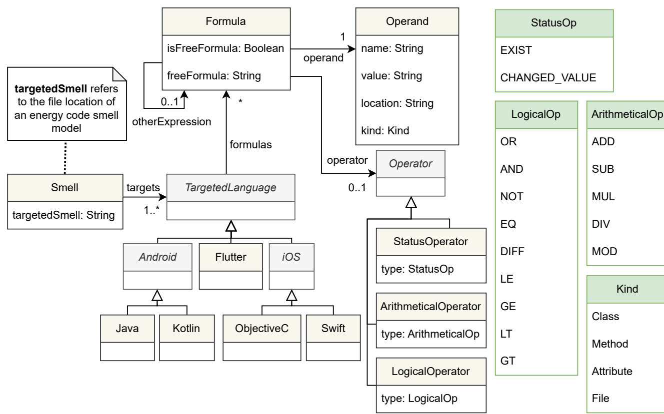

Figure 4: Platform-Specific Energy Code Smell meta-model.

```
1 Name: Torch-free
2 Description: Don t programmatically activate the LED
      flashlight, a notoriously power-hungry component
3 Score: -2
4 #Sobriety @Flashlight
```

Listing 1: Example of the Energy Code Smell meta-model concrete syntax.

Energy code smells should be defined once and reused for as many mobile code translations as necessary (R2), thus our second meta-model in Figure 4 is referring to a model of the first meta-model and it defines all the different syntax of the targeted languages (cf. Figure 1). Thanks to this solution, we separate the definition of the energy code smell from its mapping. We must nonetheless qualify the statement of R2 by noting that it is not possible to guarantee that two smells with the same purpose are not defined differently twice, which could create redundancy. However, our solution proposes to handle the translation of a single smell once rather than repeating the operation X times, which preserves R2.

The second meta-model, named the "platform-specific energy code smell DSL", serves as a bridge to translate a smell into various mobile programming languages. This mapping is achieved by directly referencing the smell via its file location in an attribute of type "String". The meta-model is designed to be versatile and allows targeting several operating systems and languages. The detection rules for energy code smells are to be written by mobile development experts. The *translations into a targeted language should be independent of any specific static analysis tool (R3)* and this aspect will be tackled in Section 4.3. Each target language can have multiple formulas associated with the detection, as there can be various ways to identify the same smell. These formulas can either be a "free formula," a simple string that needs to be searched within the files of an application, or a more complex formula composed of operands and operators for more intricate detection scenarios.

4.2 Concrete Syntax

We have implemented the textual concrete syntax of our meta-models using Xtext. Listing 1 shows an example of the concrete syntax of our energy code smell meta-model. It corresponds to

```
1 Smell: "/../energy-smell/Torch-free"
2 @Java {
3     Method."setTorchMode" in "android.hardware.
       camera2.CameraManager" == "true"
4 }
5 @Swift {
6     "AVCaptureTorchMode.on" OR "setTorchModeOn" OR "
       TorchMode.on"
7 }
```

Listing 2: Example of the Platform-Specific Energy Code Smell meta-model concrete syntax.

the example introduced in Section 3 and describes the Torch-free code smell (lines 1-2). This example has a score of -2. The negative sign means it is a bad smell, and the absolute value describes its estimated impact on the battery drain (line 3). This smell is part of the Sobriety category and applies to only one energy-hungry component: the flashlight (line 4).

Listing 2 is an instance of the platform-specific energy code smell meta-model. The first line of the listing is the location of the model to translate; here, it is the model of Listing 1.

This example illustrates the mapping with Android Java API and iOS Swift API, and we can easily understand why we need mobile development experts, as stated in Figure 2. In Android Java, we describe that we will look for a method invocation `setTorchMode`, on an instance of the class `CameraManager` when the argument is true. In the iOS Swift example, we do not give any details and only provide strings to search for in the application source code.

4.3 Model-to-Code Transformation

The definition of an energy code smell, its translation, and the code generation model for a static analysis tool should be achievable iteratively and by different actors (R4), the first part has to be done by a code smell expert according to Figure 2 and has already been tackled with the energy code smell meta-model. We have introduced our meta-model to translate the code smells in Section 4.1 and 4.2. In this section, we will explain the challenges for generating the final code for the detection rules that the static analysis tool expert of Figure 2 will integrate into static analysis tools.

The detection rules will be generated using the template language Acceleo and the difference between the two first steps is that we will need a template version that will be using the two meta-models per static analysis tools but also languages. In fact, the problem of heterogeneity arises again to illustrate our challenge: we will take the case of SonarQube and its Java API, which allows writing custom detection rules.

Unfortunately, when writing custom SonarQube rules, Swift code scanner is not available. Hence, the approach to analyzing and reporting issues in an Android Java app and iOS Swift one is completely different. Listing 3 shows an extract of the code necessary to track a Torch-free code smell in Android Java apps: the solution is based on an AST for Java. Listing 4 is the solution for iOS Swift apps, and we use a hybrid solution between the textual node of AST yielded by a tailor-made parser, and basic regular expressions.

This example demonstrates that the 24 branches of Figure 1 might have other hidden leaves and emphasize the need for abstraction and code generation.

```
1  public class TorchFreeRule extends
       ArgumentValueOnMethodCheck {
2    ...
3    public TorchFreeRule() {
4      super("setTorchMode", "android.hardware.camera2.
       CameraManager", true);
5    }
6    @Override
7    protected void checkConstantValue(Optional<Object>
       optionConstantValue, Tree reportTree, Object
       constantValueToCheck) {
8      if(optionalConstantValue.isPresent() && (
       optionalConstantValue.get().equals(
       constantValueToCheck) || ((Boolean)
       optionalConstantValue.get()))) {
9        reportIssue(reportTree, getMessage());
10 }}}
```

Listing 3: SonarQube rule of torch-free code smell detection for Android Java.

```
1  public class TorchFreeRule extends SwiftRuleCheck {
2    ...
3    @Override
4    public void apply(ParseTree tree) {
5      if (tree instanceof Swift5Parser.
       ExpressionContext) {
6        Swift5Parser.ExpressionContext id = (
       Swift5Parser.ExpressionContext) tree;
7        String expressionText = id.getText();
8        if (expressionText.contains("AVCaptureTorchMode
       .on") || expressionText.contains("setTorchModeOn"
       ) || expressionText.contains("torchMode=.on")) {
9          this.recordIssue(id.getStart().getStartIndex
       (), DEFAULT_ISSUE_MESSAGE);
10 }}}}
```

Listing 4: SonarQube rule of torch-free code smell detection for iOS Swift.

5 RELATED WORK

Many works discuss the definition and detection of energy code smells for mobile development [3, 10–12, 14], although they do not employ MBSE.

DECOR [9] addresses the definition of code smells and defines detection rules using a DSL. Despite its robust capabilities in identifying and fixing code smells, DECOR does not focus specifically on mobile applications or energy concerns. However, its methodologies and tools could integrate with or complement the research approach.

Furthermore, research exists on generating mobile applications through MBSE [1, 7, 13] but [2, 5] focus on generating energy-efficient applications. Although these approaches are compelling, it is of no help for pre-existing applications or for enhancing static analysis tools since they do not address code smells.

These related studies collectively highlight the importance of code smells and energy efficiency in software engineering, alongside the critical role of abstraction. This demonstrates the need for specialized tools to address these issues in the mobile application domain. By building on these principles, this research aims to bridge the gap and introduce a novel solution tailored to detect and address energy code smells in mobile applications.

6 CONCLUSION AND FUTURE WORK

Automated detection of mobile-specific energy code smells is still in its infancy, but the next big step is already to address time-to-market. Indeed, the diversity of mobile technologies, along with the diversity of static code analyzers, hinders a widespread adoption by mobile developers.

In this paper, we have presented an approach to large-scale mobile-specific energy hotspot detection, starting from the description of commonly shared energy smells to a variety of operational detection rules. We have proposed two meta-models and first steps towards code generation for integration into static analysis tools.

We are currently implementing our proposal and laying the groundwork for a future website that will store existing code smells described in categorized models. This initiative aims to foster collaboration among code smell experts and environment-friendly mobile developers, facilitating the integration of these detection rules into static analysis tools.

ACKNOWLEDGMENTS

We'd like to thank the core team of the "Green Code Initiative" (GCI) for their great work on the series of plugins for SonarQube.

REFERENCES

[1] Hanane Benouda, Redouane Essbai, Mostafa Azizi, and Mimoun Moussaoui. 2016. Modeling and Code Generation of Android Applications Using Acceleo. *International Journal of Software Engineering and Its Applications* 10 (03 2016), 83–94.
[2] Kowndinya Boyalakuntla, Marimuthu Chinnakali, Sridhar Chimalakonda, and Chandrasekaran K. 2022. eGEN: an energy-saving modeling language and code generator for location-sensing of mobile apps. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. ACM, 1697–1700.
[3] Luis Cruz and Rui Abreu. 2019. Catalog of energy patterns for mobile applications. *Empirical Softw. Engg.* 24, 4 (aug 2019), 2209–2235.
[4] Olivier Le Goaër. 2020. Enforcing green code with Android lint. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 85–90.
[5] Imre Kelényi, Jukka K. Nurminen, Matti Siekkinen, and László Lengyel. 2014. Supporting Energy-Efficient Mobile Application Development with Model-Driven Code Generation. In *Advanced Computational Methods for Knowledge Engineering*. Springer, 143–156.
[6] Olivier Le Goaer and Julien Hertout. 2023. ecoCode: a SonarQube Plugin to Remove Energy Smells from Android Projects. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. ACM, Article 157.
[7] Olivier Le Goaer and Sacha Waltham. 2013. Yet another DSL for cross-platforms mobile development. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages (GlobalDSL '13)*. ACM, 28–33.
[8] Olivier Le Goaër. 2024. *Mobile-specific Best Practices for Sustainable Software*. https://github.com/cnumr/best-practices-mobile
[9] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 36 (01 2010), 20–36.
[10] Ghulam Rasool and Azhar Ali. 2020. Recovering Android Bad Smells from Android Applications. *Arabian Journal for Science and Engineering* 45 (02 2020).
[11] Reeshti, Rajni Sehgal, Deepti Mehrotra, Renuka Nagpal, and Tanupriya Choudhury. 2021. *Code Smell Refactoring for Energy Optimization of Android Apps*. Springer, 371–379.
[12] Ana Ribeiro, João Ferreira, and Alexandra Mendes. 2021. EcoAndroid: An Android Studio Plugin for Developing Energy-Efficient Java Mobile Applications. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 62–69.
[13] Steffen Vaupel, Gabriele Taentzer, Jan Peer Harries, Raphael Stroh, René Gerlach, and Michael Guckert. 2014. Model-Driven Development of Mobile Applications Allowing Role-Driven Variants. In *Model-Driven Engineering Languages and Systems*. Springer, 1–17.
[14] Zhiqiang Wu, Xin Chen, and Scott Uk-Jin Lee. 2023. A systematic literature review on Android-specific smells. *J. Syst. Softw.* 201 (2023), 111677.