Green Software

TACTICS, PATTERNS & SMELLS

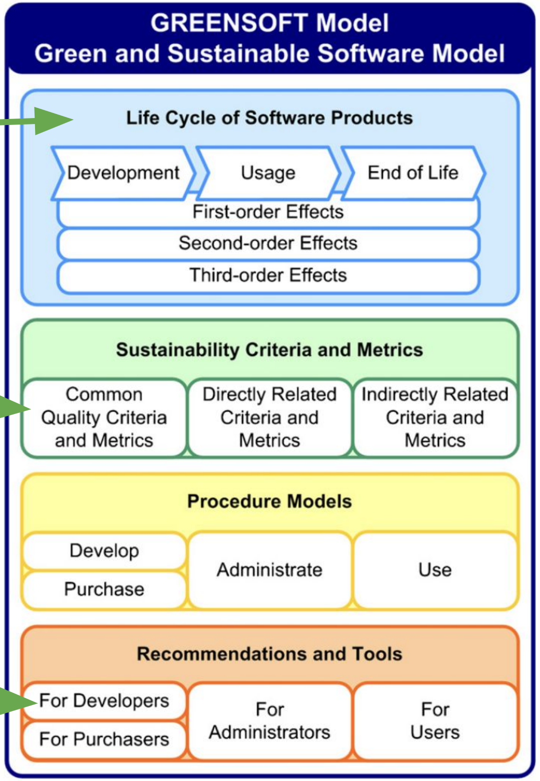
Dr. Olivier Le Goaër, LIUPPA

An “old” story...

🌐 A **holistic** approach to green software

✨ Sustainability as a **quality** attribute of software (i.e. non-functional property)

🛠️ An increasing demand for **tooling** from software practitioners (mainly devs)



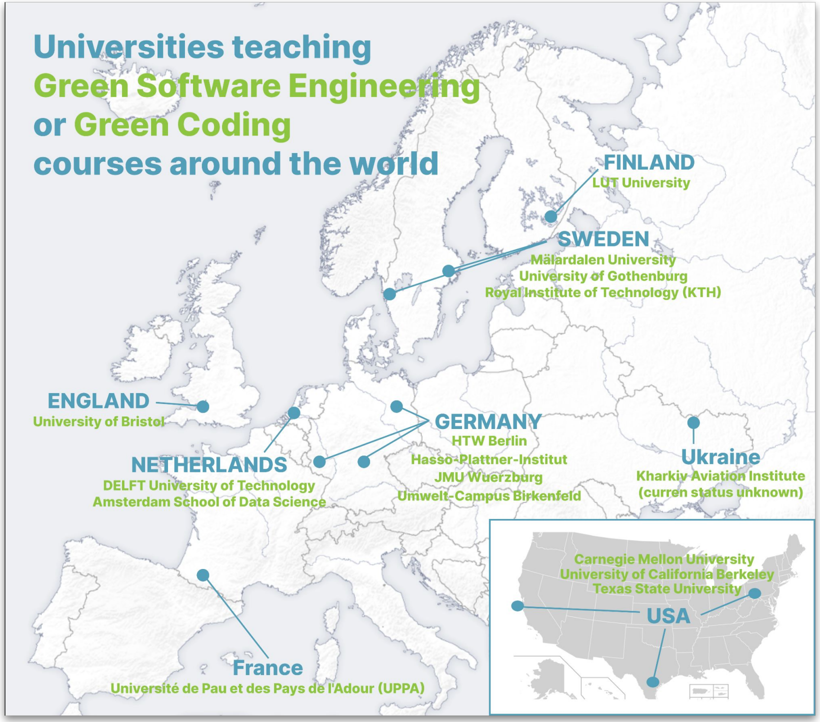
Stefan Naumann *et al.*, “The GREENSOFT Model: A reference model for green and sustainable software and its engineering”, *Sustainable Computing: Informatics and Systems*, Volume 1, Issue 4, 2011, pages 294-304

…more acute than ever

"Green Coding is the act of **designing**, **developing**, **maintaining**, and **(re-)using** software systems in a way that requires as little **energy** and **natural resources** as possible. Green Coding **methods** or **practices** thus mean any action or use of technology intended and suitable to further this."

Dennis Junger *et al.*, Potentials of Green Coding, *INFORMATIK 2023 - Designing Futures: Zukünfte gestalten*, pp. 1289-1299

Green Software Engineering



Dennis Junger *et al.*, Potentials of Green Coding, *Intermediate Report of the Project "Potentials of Green Coding" for the ISOC Foundation*, 2024

Topics at GREENS'25 (co-located ICSE)

9th International Workshop on Green and Sustainable Software (GREENS'25)

Official website of the International Workshop on Green and Sustainable Software (GREENS)

Co-Located with:	Date:	Location:	Submission:
ICSE25	April 27-May 3 2025	Ottawa (Canada)	November 11, 2024

Theme & Goals

Engineering green software-intensive systems is critical in our drive towards a sustainable, smarter planet. The goal of green software engineering is to apply green principles to the design and operation of software-intensive systems. **Green and self-greening software systems have tremendous potential to decrease energy consumption.** Moreover, software can and should be rethought to address sustainability issues, for instance, innovative business models, new processes, and incentives. *Monitoring* and *measuring* the greenness of software is critical to the notion of sustainable software. Demonstrating improvement is paramount for users to achieve and effect change. Analysis of the sustainability of a specific software system requires software that aids developers in weighing the four dimensions of sustainability – economic, social, environmental, and technical – with their attendant trade-offs. The software engineering community must assume leadership in this important challenge. In this workshop, we explore the theme of *"green software engineering for software sustainability"* with the goal of creating actionable outcomes that will affect how software engineering is practiced and taught in the future to help organizations prioritize their sustainability objectives.

Topics of Interest

GREENS 2025 seeks contributions addressing, but not limited to, the following topics related to sustainable software systems and green software engineering:

- Practices for sustainability-aware software engineering
- Metrics and measures for sustainability-aware software engineering
- Teaching and training of skills and competencies in sustainability-aware software engineering
- Sustainable computing from a software engineering and software-intensive system perspective
- Applied, or experimented with, sustainability-aware software engineering methodologies at all levels (from requirements engineering and architecture design to coding, testing, and maintenance)
- Energy-efficient choices for architecture, including design patterns, algorithms, data structures, programming languages, language runtime, and infrastructure.
- Architectural implications (architectural tactics, architectural styles, design patterns and anti-patterns) for green and sustainable software
- Sustainability-aware architectures in context (e.g., cloud-edge continuum)
- Meta-analyses and syntheses of studies to build theories on green and sustainable software
- Conceptual reflections related to software sustainability
- Progress on the various dimensions of software sustainability and their interplay
- Software adaptation and evolution for sustainability
- Tools to support sustainability-aware decision-making
- Sustainability of emerging computing technologies (AI and generative AI, cloud-fog-edge, quantum computing)
- Green AI, lighter, less data-intensive, and less energy-consuming AI models and architectures
- Sustainable Large Language Models (LLMs)
- Reduction of software organizations' compute-heavy workloads
- Cloud and energy efficiency
- Standards on the environmental sustainability of software and AI software

https://greensworkshop.github.io/

Burn your idols

Green language is no silver bullet

Mobile end users are very sensitive to battery life, and the world's leading platforms are spending 💰 to improve the energy efficiency of their ecosystems.

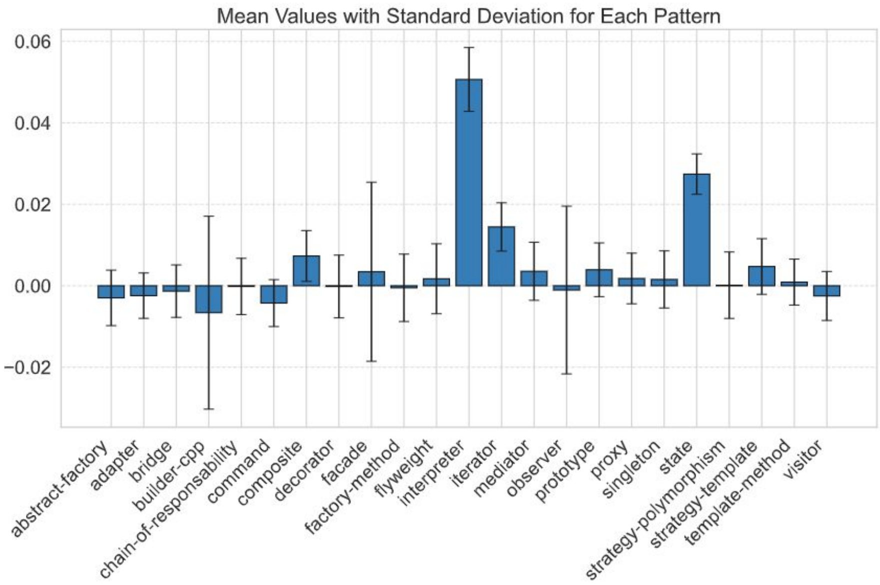
Yet they are pushing Kotlin (formerly Java) and Swift (formerly Objective-C) to write software at large 🤔

⇒**The answer rather lies in your design choices**

	Energy (J)
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

Rui Pereira *et al*. "Ranking Programming Languages by Energy Efficiency". Science of Computer Programming, volume 205. Elsevier, 2021.

OOP Design patterns



⇒**Variations below 5% = negligible**

Investigating the Impact of Software Design Patterns on Energy Consumption

Abstract—**GoF's design patterns are popular structures designed for flexible and reusable object-oriented software. However, their energy impact is not well understood, with contradictory existing results. In this paper, we aim to answer the relation between design patterns and energy consumption through a multi-platform and multi-software stack empirical study, covering all design patterns, and multiple hardware, OS, languages and compilers. We found, with a high confidence level, that none of the GoF's design patterns has a significant impact on energy consumption. Therefore, they can be considered energy-neutral, allowing eco-friendly developers to continue using them. We argue that software energy efficiency should rather be considered within the business logic embedded by a design pattern, and ultimately raised to a higher perspective of software development.**

Index Terms—**Energy, Power, Design Patterns, Empirical Study, Replication Study, Software Engineering**

I. Introduction and Motivation

The Gang of Four (GoF) design patterns are well-known best practices for the design of object-oriented systems. GoF design patterns is the collection of 23 design patterns, grouped into three categories: creational, structural and behavioral [1]. They were proposed in 1995, and are still relevant today in modern software engineering. Many development frameworks rely heavily on them, and researchers are still exploring the impacts of these patterns. Adopting these patterns has significantly improved the quality of large-scale software projects with a particular focus on maintainability, an important non-functional requirement.

In parallel, the growth of large software projects came at an increased requirement for hardware, and an overall increase in financial costs and energy consumption. For the latter, researchers estimate that information and communication technologies (ICT) consumes today around 7% of global electricity, and is expected to continue rising [2]. Software practitioners consider writing green software a major concern [3], but they lack in knowledge and tools to understand the energy impacts of their code and how to make it more energy-efficient [4]. This makes energy efficiency another important non-functional requirement.

Today, crafting and building energy-efficient software is faced with many challenges: from current and upcoming regulations and norms, to rising energy costs on servers hosting software services, to the rising awareness of users and businesses on the need for IT and software to be *greener*, and to the lack of training, knowledge, and tools to build green software.

GoF design patterns take full advantage of object-orientation mechanisms, notably object instantiation, encapsulation, method overriding and late binding thereof (aka run-time polymorphism, where 19 out of the 23 GoF patterns use it). Therefore, existing studies often consider that the adoption of design patterns comes at an additional energy cost. However, it is difficult to estimate the impact of design patterns on energy consumption, as the literature is divergent and contradictory. Beyond patterns, existing studies on the overhead of object-oriented programming style [5] or for embedded systems [6], have shown that there is an increase in energy consumption. As the vast majority of developers and practitioners have adopted object-oriented programming and design patterns, we aim to analyze the energy impact of design patterns.

Existing studies are either partial (not all patterns studied), outdated and out of sync with modern software stacks and energy measurement approaches, or use a limited experimental and measurement protocol that do not allow for clear and generalized conclusions. Therefore, we argue that it is necessary today to conduct a solid empirical study, including replicating existing studies and design pattern codes. Can we trust the results of existing studies in terms of energy measurements? Are their results consistent in modern hardware and modern software stack?

We aim to conduct a multi-platform study, exploring the energy impact of patterns on x86 computers along with ARM-based systems (such as macOS or smartphones). For the latter, we decided to exclude it from our study because existing energy estimation approaches [7], [8] are not reliable in measuring specific software code. Also, a design pattern code cannot be run in isolation as it requires being part of a larger component-based reactive system that might skew energy measurements.

In this paper, we aim to answer this research question:

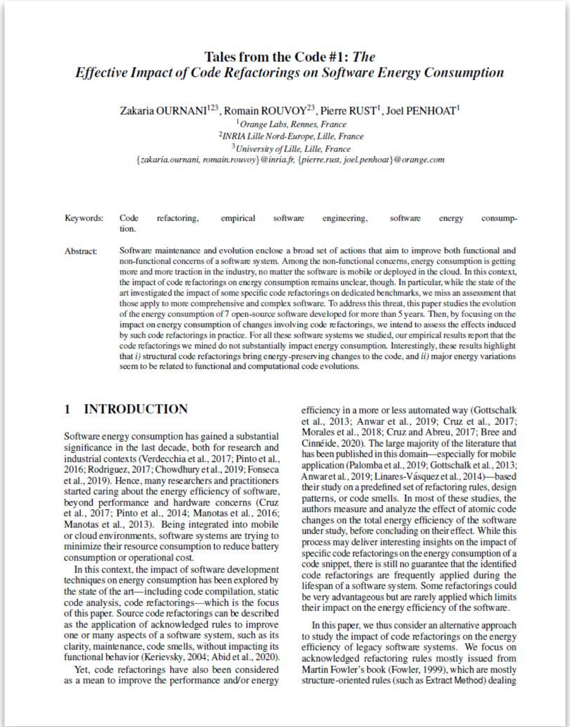
Do design patterns impact software energy consumption?

To answer our research question, we propose a multi-platform and multi-software stack empirical replication study. Therefore, 22 + 1 design patterns are studied, with implementation variants written in different languages and executed on different platforms, CPU architectures, operating systems and compiled with different compiler versions.

The remainder of this paper is organized as follows: Section II explore existing studies and their limitations, highlighting the need for a multi-platform empirical replication study. In Section III, we detail our experimental protocol, describing the platforms and software stacks, the replication methodology along with the additional experiments we conducted. The

A. Noureddine & O. Le Goaer, "Investigating the Impact of Software Design Patterns on Energy Consumption". - *under submission* -, 2024

OOP Refactorings



Tales from the Code #1: *The Effective Impact of Code Refactorings on Software Energy Consumption*

Zakaria OURNANI[123], Romain ROUVOY[23], Pierre RUST[1], Joel PENHOAT[1]
[1]*Orange Labs, Rennes, France*
[2]*INRIA Lille Nord-Europe, Lille, France*
[3]*University of Lille, Lille, France*
{*zakaria.ournani, romain.rouvoy*}@*inria.fr*, {*pierre.rust, joel.penhoat*}@*orange.com*

Keywords: Code refactoring, empirical software engineering, software energy consumption.

Abstract: Software maintenance and evolution enclose a broad set of actions that aim to improve both functional and non-functional concerns of a software system. Among the non-functional concerns, energy consumption is getting more and more traction in the industry, no matter the software is mobile or deployed in the cloud. In this context, the impact of code refactorings on energy consumption remains unclear, though. In particular, while the state of the art investigated the impact of some specific code refactorings on dedicated benchmarks, we miss an assessment that those apply to more comprehensive and complex software. To address this threat, this paper studies the evolution of the energy consumption of 7 open-source software developed for more than 5 years. Then, by focusing on the impact on energy consumption of changes involving code refactorings, we intend to assess the effects induced by such code refactorings in practice. For all these software systems we studied, our empirical results report that the code refactorings we mined do not substantially impact energy consumption. Interestingly, these results highlight that *i)* structural code refactorings bring energy-preserving changes to the code, and *ii)* major energy variations seem to be related to functional and computational code evolutions.

1 INTRODUCTION

Software energy consumption has gained a substantial significance in the last decade, both for research and industrial contexts (Verdecchia et al., 2017; Pinto et al., 2016; Rodriguez, 2017; Chowdhury et al., 2019; Fonseca et al., 2019). Hence, many researchers and practitioners started caring about the energy efficiency of software, beyond performance and hardware concerns (Cruz et al., 2017; Pinto et al., 2014; Manotas et al., 2016; Manotas et al., 2013). Being integrated into mobile or cloud environments, software systems are trying to minimize their resource consumption to reduce battery consumption or operational cost.

In this context, the impact of software development techniques on energy consumption has been explored by the state of the art—including code compilation, static code analysis, code refactorings—which is the focus of this paper. Source code refactorings can be described as the application of acknowledged rules to improve one or many aspects of a software system, such as its clarity, maintenance, code smells, without impacting its functional behavior (Kerievsky, 2004; Abid et al., 2020).

Yet, code refactorings have also been considered as a mean to improve the performance and/or energy efficiency in a more or less automated way (Gottschalk et al., 2013; Anwar et al., 2019; Cruz et al., 2017; Morales et al., 2018; Cruz and Abreu, 2017; Bree and Cinnéide, 2020). The large majority of the literature that has been published in this domain—especially for mobile application (Palomba et al., 2019; Gottschalk et al., 2013; Anwar et al., 2019; Linares-Vásquez et al., 2014)—based their study on a predefined set of refactoring rules, design patterns, or code smells. In most of these studies, the authors measure and analyze the effect of atomic code changes on the total energy efficiency of the software under study, before concluding on their effect. While this process may deliver interesting insights on the impact of specific code refactorings on the energy consumption of a code snippet, there is still no guarantee that the identified code refactorings are frequently applied during the lifespan of a software system. Some refactorings could be very advantageous but are rarely applied which limits their impact on the energy efficiency of the software.

In this paper, we thus consider an alternative approach to study the impact of code refactorings on the energy efficiency of legacy software systems. We focus on acknowledged refactoring rules mostly issued from Martin Fowler's book (Fowler, 1999), which are mostly structure-oriented rules (such as Extract Method) dealing

Table 2: The observed impact of mined refactoring rules

Refactoring	Count	CountxCommits	IC	WIC	$\delta\%(r)$	$\delta\lvert\%\rvert(r)$	RI
add method annotation	10120	**80960**	**30.77%**	**43.41%**	**1.13%**	**2.14%**	**7.34**
change variable type	101	**606**	16.67%	14.95%	0.24%	1.32%	1.17
rename parameter	45	**180**	**33.33%**	**71.69%**	-0.07%	**1.82%**	**5.12**
change parameter type	42	**168**	11.76%	17.07%	-0.03%	1.20%	0.81
change attribute type	26	**130**	16.67%	9.39%	0.12%	1.35%	0.63
add class annotation	63	**216**	**33.33%**	**63.53%**	**1.30%**	**2.20%**	**2.77**
move class	40	**120**	**30.00%**	**54.28%**	**0.77%**	**2.21%**	**3.55**
change return type	28	**112**	14.81%	19.93%	0.14%	1.11%	0.88
move method	33	**99**	21.43%	19.10%	0.59%	1.76%	1.00
rename variable	21	**84**	25.00%	18.24%	0.46%	1.44%	1.04
move attribute	18	**54**	25.00%	18.81%	-0.07%	1.92%	1.06
extract method	37	**37**	20.00%	71.87%	0.08%	1.24%	0.88
pull up method	32	**32**	33.33%	38.90%	0.03%	1.97%	0.75
rename class	6	**24**	25.00%	13.71%	**1.14%**	**1.51%**	0.82
add attribute annotation	8	16	20.00%	15.12%	0.64%	1.14%	0.34
rename attribute	5	15	30.00%	8.77%	0.55%	1.62%	0.42
add parameter	6	12	16.67%	6.55%	0.82%	1.47%	0.19
merge parameter	6	6	100.00%	100.00%	6.00%	6.00%	6.00
extract class	2	4	33.33%	11.14%	0.72%	2.62%	0.57
extract variable	3	3	11.11%	10.52%	0.49%	0.91%	0.10
remove method annotation	1	1	11.11%	0.77%	0.71%	1.40%	0.01
rename method	1	1	11.11%	2.20%	0.32%	1.10%	0.02
modify method annotation	1	1	33.33%	7.99%	2.50%	2.50%	0.20
move & rename method	1	1	20.00%	13.17%	-0.32%	2.32%	0.30
merge attribute	1	1	100.00%	100.00%	6.00%	6.00%	6.00

⇒ **fairly neutral**

Zakaria Ournani, Romain Rouvoy, Pierre Rust, Joel Penhoat. Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption. *ICSOFT 2021 - 16th International Conference on Software Technologies,* Jul 2021, Virtual, France.

OOP Code smells

V. Wohlgemuth, D. Kranzlmüller, M. Höb (Editors): EnviroInfo 2023,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2023 111

Influence of Static Code Analysis on Energy Consumption of Software

Christoph Brosch[1]

Abstract: In recent years, the rise of mobile devices, such as smartphones, smartwatches, or tablets, has led to an increased demand for energy-efficient software. In order to achieve this, developers can use static code analysis tools, such as Pylint, to detect potential issues in their code. This paper investigates how the usage of static code analysis influences the energy consumption of software. More specifically, we used the programming language Python and the general-purpose static code analysis tool Pylint [Py22]. For this purpose, we measured the energy consumption for algorithms implemented in the *Benchmarks Game* [Go22] before and after implementing the annotations and compared the results. Our findings suggest that resolving the annotations can have a negative impact on energy consumption. This was the case in 3 out of 8 algorithms. The remaining cases showed no significant difference. We assume that the increased energy consumption is due to the multitude of possibilities to implement annotations, leading to a possibility for worsening performance. Further research and experimentation are needed to objectively evaluate the impact of Pylint and static code analysis by extension, on energy consumption.

Keywords: Static code analysis; Linter; Programming; Energy consumption; Efficiency; Python

Built-in code smells of PyLint

Brosch, Christoph (2023): Influence of Static Code Analysis on Energy Consumption of Software. EnviroInfo 2023.

⇒ **Effective or counter-productive?**

Who's right?

Empirical Evaluation of the Energy Impact of Refactoring Code Smells

Roberto Verdecchia[†*], René Aparicio Saez[*], Giuseppe Procaccianti[*], Patricia Lago[*]
[†]Gran Sasso Science Institute, L'Aquila, Italy
[*]Vrije Universiteit Amsterdam, The Netherlands
roberto.verdecchia@gssi.it, [g.procaccianti, p.lago]@vu.nl

***Abstract*—Software energy efficiency has gained the increasing attention of the research community. How to improve it, however, still lacks evidence. Specifically, the impact of code smell refactoring on energy efficiency has been scarcely investigated. In the exploratory study here reported, we investigate the impact on performance and energy consumption of refactoring well-known code smells on Java software applications. In order to understand if software metrics can be used as indicators of the energy impact of refactoring, we also measured the variation caused by refactoring on a set of well-established software metrics. We conducted a controlled experiment using state-of-the-art power measurement equipment. Statistical hypothesis testing and effect size estimation were performed on the experimental results, which show that in one out of three applications, refactoring each smell significantly impacted power- and energy consumption. E.g., refactoring Feature Envy and Long Method smells led to a 49% energy efficiency improvement. No software metric, however, significantly correlated with execution time, power or energy consumption. In conclusion, refactoring code smells resulted to be a viable process to significantly improve software energy efficiency. The magnitude of the impact may depend on application properties, e.g. size or age. Further research is needed to understand the relationship between software metrics and energy efficiency.**

***Keywords*-energy efficiency; code smells; refactoring; empirical experiment.**

I. Introduction

Computing devices have become a major part of our everyday life. The number of these devices is predicted to globally increase in the coming years. Not only do people own more devices themselves, they also increasingly rely on services provided by Cloud providers, which are typically hosted in large-scale data centers. This brings up the issue of their environmental impact: the carbon footprint of Information and Communications Technology (ICT) accounts for 2% of global emissions [1] and is expected to keep on growing [2]. Data centers alone account for around 1.1% to 1.5% of global energy consumption according to a report from 2010 [3]. These numbers indicate the global scale reached by ICT, and shows the need for energy efficient ICT solutions. While hardware solutions have been thoroughly researched, the same cannot be said for software. As shown by Pinto et al. [4], only since recent years the energy efficiency of application software is taken into consideration [5]. It is expected that increasing software energy efficiency can cause major changes in the energy consumption of ICT, thanks to the global scale of software usage [6]. Therefore, the next step in finding significant energy efficient improvements in ICT will most likely be software related.

There are a number of empirical studies showing how software engineering best practices can improve energy efficiency [7], [8]. Code refactoring is probably the most common approach to re-engineer software applications in order to improve non-functional attributes. Refactoring activities are typically aimed at removing *code smells* [9], that can be defined as "certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality" [10]. Detection and refactoring of some code smells can be automated by using special-purpose tools, among which the Eclipse plugin JDeodorant [11] results to be the one which is most commonly utilized [12].

In many studies, refactoring code smells was found to have a positive impact on software maintainability [13], [14]. However, the impact on energy efficiency is only marginally investigated [15], [16], [17], [18].

In this study, we aim to perform an exploratory analysis of the impact of code smell refactoring on energy consumption and performance in software applications. We selected five different code smells (Feature Envy, Type Checking, Long Method, God Class and Duplicated Code) that we automatically detected and refactored in three open-source, ORM-based Java software applications. The refactoring was applied both in isolation (i.e. on all the occurrences of a single smell) and in combination (i.e. on all the occurrences of all smells). We then performed a benchmark of each version of the application by means of an automated test scenario and we collected energy consumption and performance metrics in a controlled environment. We also investigate whether well-established object-oriented software metrics can be used as indicators of the impact of refactoring the smells. This would allow developers to use such metrics as proxies for identifying and refactoring code smells with a high energy impact, thus removing the need of performing dedicated measurements and benchmarks. To the best of our knowledge, this is the largest study on the energy impact of code smells.

The structure of the paper is as follows. Section II will present related work on this topic. Section III and IV discusses the definition and planning of our empirical experiment, along with subject selection, hypothesis formulation and instrumentation. Section V describes our experiment design and

Feature Envy
Type Checking
Long Method
God Class
Duplicated Code

Verdecchia, Roberto & Saez, René & Procaccianti, Giuseppe & Lago, Patricia. (2018). Empirical Evaluation of the Energy Impact of Refactoring Code Smells.

Let's dive in green

Green Software Engineering: granularity levels

Green tactics

Build or not to build. That is the ~~question~~ tactic

Motto: “The software (or feature) with the least impact is the one you don't build” (avoided carbon footprint)

Build only software (or feature) “Useful, Usable and Used” (the 3U’s)

🚦 *Go/no go* from high-level specifications

⇒ Agile Methodology, Lean ICT, …

All-purpose & domain-specific tactics

General software system (a.k.a practices)

- Avoid use of byte-code
- Batch I/O
- Code migration
- Compiler optimization
- Decrease algorithmic complexity
- Efficient GUI
- Free or unmap unneeded memory
- Keep 3rd party software up-to-date
- Lazy loading
- Less frequent or avoiding polling
- Put application to sleep
- Reduce data redundancy
- Reduce memory leaks
- Reduce QoS dynamically
- Reduce transparency and abstractions
- Static GUI
- Use asynchronous I/O
- Use efficient queries
- Use JIT compiler
- Use low-level programming

Procaccianti, G., Fernández, H., & Lago, P. (2019). Green Software in Practice: Empirical Validation and Assessment of Best Practices for Writing Energy-Efficient Software

ML-based software system

Green Architectural Tactics for ML-Enabled Systems

Data-centric	Algorithm design	Model optimization	Model training	Deployment	Management
T1: Apply sampling techniques **T2**: Remove redundant data **T3:** Reduce number of data features **T4:** Use input quantization **T5:** Use data projection	**T6:** Choose an energy-efficient algorithm **T7**: Choose a lightweight algorithm alternative **T8:** Decrease model complexity **T9:** Consider reinforcement learning for energy efficiency **T10:** Use dynamic parameter adaptation **T11:** Use built-in library functions*	**T12:** Set energy consumption as a model constraint **T13:** Consider graph substitution **T14:** Enhance model sparsity **T15:** Consider energy-aware pruning **T16**: Consider transfer learning **T17:** Consider knowledge distillation	**T18:** Use quantization-aware training **T19:** Use checkpoints during training **T20:** Design for memory constraints*	**T21:** Consider federated learning **T22:** Use computation partitioning **T23:** Apply cloud fog network architecture **T24:** Use energy-efficient hardware **T25:** Use power capping **T26:** Use energy-aware scheduling **T27**: Minimize referencing to data*	**T28:** Use informed adaptation* **T29:** Retrain the model if needed **T30:** Monitor computing power

H. Järvenpää, P. Lago, J. Bogner, G. Lewis, H. Muccini and I. Ozkaya, "A Synthesis of Green Architectural Tactics for ML-Enabled Systems," *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, Lisbon, Portugal, 2024, pp. 130-141.

Cloud software system

Tactic	*Quality Attribute*	*Rationale*
Consolidation	Availability	During VM migration some services may not be available.
	Security	Live VM migration over the network requires to transfer application code, metadata and workloads, making them vulnerable to attacks.
Energy Modeling	Modifiability	Energy Connectors are component-specific and therefore must be reimplemented if the architecture changes.
Service-Adaptation	Flexibility	The orchestrator concentrates all service composition logic in a single node.

Procaccianti, Giuseppe & Lago, Patricia & Lewis, Grace. (2014). A Catalogue of Green Architectural Tactics for the Cloud. *Proceedings - Working IEEE/IFIP Conference on Software Architecture 2014, WICSA* 2014. 41-44.

Developer choice as a tactic

When it comes to writing a mobile app, you choose either the native SDK or a cross-platform SDK. **There's no going back**.

Development	App size (KB)		Data transfer (KB)		Energy (mAh)	
	Android	iOS	Android	iOS	Android	iOS
Swift	N/A	216	N/A	?	N/A	8,59
Kotlin	1200	N/A	944	N/A	21,60	N/A
KMM	3600	1600	932	?	21,66	?
Flutter	17500	18000	1190	?	18,27	9,19
React Native	27300	13100	706	?	19,45	11,62

⇒Cross-platform is a challenger

Ecological Impact of Native versus Cross-Platform Mobile Apps: a Preliminary Study

Vincent Frattaroli, *Inside App*, Paris, France, vincent.frattaroli@insideapp.fr
Olivier Le Goaer, *E2S UPPA, LIUPPA*, *Universite de Pau et des Pays de l'Adour*, Pau, France, olivier.legoaer@univ-pau.fr
Olivier Philippot, *Greenspector*, Nantes, France, ophilippot@greenspector.com

***Abstract*—What are the best mobile development approaches to cut the carbon footprint? To answer this question, this experience paper provides a life-size comparison of native versus cross-platform frameworks prevailing in the mobile software industry at the time of writing, namely Kotlin Multiplatform Mobile, React Native and Flutter. To do this, we collected metrics related to the package size, network traffic and battery drain issued by a boilerplate application developed following the different approaches. Our preliminary findings tend to show that the cross-platform solutions perform quite well.**

***Index Terms*—android, ios, kotlin, react, flutter, carbon**

I. INTRODUCTION

Within a decade, the mobile software sector has seen tremendous success. The landscape has also reorganised, leading to the overwhelming dominance of 2 mobile platforms that now share the market: almost 71% for Android (Google) and 27% for iOS (Apple). However, this market fragmentation is still a concern for mobile developers. Either they opt for native development, but have to write the app twice, or they opt for cross-platform development to write a single code base. The pros and cons of each development method are regularly debated, whether from a time-to-market or user experience perspective [8]. But as climate change rises up the global economic and political agenda, more and more (mobile) developers are also concerned about the sustainability of the software they create. It is therefore useful to compare development practices from an environmental perspective until the decarbonization of software becomes mainstream practice.

Unfortunately, the everyday mobile developers often finds himself alone when facing this challenge. In [11], a survey of experienced developers showed that they are genuinely interested in the energy consumption of software, despite the fact that little knowledge is available. Authors of [18] pinpointed the energy-related questions posed in Stack Overflow by mobile developers, anxious to learn about power-related problems that are encountered by others.

From the trenches, at the implementation stage, eco-friendly mobile developers may refer to catalogues of code smells inherited from embedded systems [12] or mobile-specific green patterns [2]. More recently, they may use a lint-like tool to automatically clean their codebase of energy code smells [4], [7]. Before that, the choice of programming language itself can have a small impact on energy consumption [5] in particular contexts. But in the case of Android for example, this choice is obviously limited, and it has been shown that migrating from Java to Kotlin has no significant impact on the energy efficiency of the app [1]. However, an even earlier choice that the development team has to make (and therefore the hardest to change later) is the choice between native and cross-platform development methods. Therefore, this paper investigates whether this key design decision will have an ecological impact once the mobile application is deployed on a potentially large number of devices.

To this end, we have formulated the following 3 research questions:

- **RQ1**: *Does the development method affect the size of the application archive file?*
- **RQ2**: *Does the development method affect the amount of data the application exchanges over the network?*
- **RQ3**: *Does the development method influence the energy consumption of the app?*

By answering RQ1, we are fighting the "fatware" syndrome, i.e. the inflation of software size over the last decades (e.g. TikTok on iOS is now 400Mb!), which marginalises owners of low-end devices. In fact, the number of bytes downloaded to install the application and its subsequent updates is constantly increasing. This is particularly salient on mobile platforms, where updates are frequent and automatic, regardless of new differential download techniques. Answering RQ2 leads to pinpoint how much the client side, network infrastructure and server side are stressed over the Internet. Indeed, it is not unreasonable to assume that the more data that is exchanged and processed, the more energy is likely to be consumed on a global scale. Last but not least, by answering RQ3, we are looking at the battery drain. The direct effect of this is an increased demand for electricity (and its source production[1]) to charge the handheld device. The indirect effect is to shorten the life of the device, since its lithium-ion battery has a limited number of charge/discharge cycles. It is worth recalling that the manufacture of new user device remains the main source of greenhouse gas emissions in the ICT sector [6].

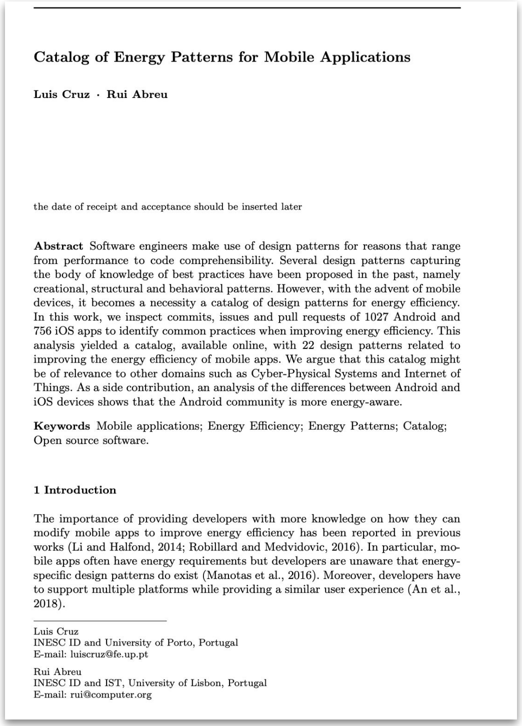
[1]Depending on the country, electricity production can be more or less decarbonised. See https://app.electricitymaps.com/

V. Frattaroli, O. Le Goaer, O. Philippot, "Ecological Impact of Native versus Cross-Platform Mobile Apps: a Preliminary Study." *The 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2023

🍃 Green patterns 🍃

Patterns for mobile apps

- Dark UI Colors
- Decrease Rate
- Dynamic Retry Delay
- Avoid Extraneous Work
- Race-to-idle
- Open Only When Necessary
- Push Over Poll
- Power Save Mode
- Sensor Fusion
- No screen interaction
- Avoid Extraneous Graphics and Animations
- Reduce Size
- User Knows Best
- Inform Users
- Enough resolution
- Batch Operations
- Suppress Logs
- WiFi Over Cellular
- Power Awareness
- Kill Abnormal Tasks
- Manual Sync - On Demand



Catalog of Energy Patterns for Mobile Applications

Luis Cruz · Rui Abreu

the date of receipt and acceptance should be inserted later

Abstract Software engineers make use of design patterns for reasons that range from performance to code comprehensibility. Several design patterns capturing the body of knowledge of best practices have been proposed in the past, namely creational, structural and behavioral patterns. However, with the advent of mobile devices, it becomes a necessity a catalog of design patterns for energy efficiency. In this work, we inspect commits, issues and pull requests of 1027 Android and 756 iOS apps to identify common practices when improving energy efficiency. This analysis yielded a catalog, available online, with 22 design patterns related to improving the energy efficiency of mobile apps. We argue that this catalog might be of relevance to other domains such as Cyber-Physical Systems and Internet of Things. As a side contribution, an analysis of the differences between Android and iOS devices shows that the Android community is more energy-aware.

Keywords Mobile applications; Energy Efficiency; Energy Patterns; Catalog; Open source software.

1 Introduction

The importance of providing developers with more knowledge on how they can modify mobile apps to improve energy efficiency has been reported in previous works (Li and Halfond, 2014; Robillard and Medvidovic, 2016). In particular, mobile apps often have energy requirements but developers are unaware that energy-specific design patterns do exist (Manotas et al., 2016). Moreover, developers have to support multiple platforms while providing a similar user experience (An et al., 2018).

Luis Cruz
INESC ID and University of Porto, Portugal
E-mail: luiscruz@fe.up.pt

Rui Abreu
INESC ID and IST, University of Lisbon, Portugal
E-mail: rui@computer.org

Cruz, Luís & Abreu, Rui. (2019). Catalog of energy patterns for mobile applications. *Journal of Empirical Software Engineering*. 24.

Patterns for web apps

Table 1: Energy patterns with applicability to web, classified to client (C) or Server (S), description, and examples.

Pattern	Applicability	C/S	Description
Avoid Extraneous Graphics and Animations	✔	S	Use battery-intensive graphics or animations with moderation. *e.g.*, A website not loading heavy graphics until users interact with them.
Avoid Extraneous Work	✔	S	Present only relevant data or perform tasks that have a direct impact on the user experience. *e.g.*, Mozilla's API in Listing 1 informs users for the page visibility to let audio/video pause.
Batch Operations	✔	S	Combine multiple operations to perform batch processing. *e.g.*, Web API from Microsoft to group several operations into a single HTTP request [12].
Cache	✔	C	Utilize caching mechanisms to reduce network load. *e.g.*, A code example to cache an API response in the local storage [31].
Dark UI Colors	✔	C	Provide a web application with the dark UI color theme. *e.g.*, Facebook provides an option on the website to switch to a dark theme.
Decrease Rate	✔	S	Increase the time interval between requests to the backend. *e.g.*, Library website refreshes the book availability only a few times a day.
Dynamic Retry Delay	✔	S	Use a systematic retry increasing time interval after each failed attempt to a resource, such as a database, or network. *e.g.*, In the Fibonacci series utilize a retry mechanism API to handle abnormal conditions [31].
Enough Resolution	✔	S	Provide high-accuracy data only when strictly necessary. *e.g.*, *AVIF* and *WebP* image formats reduces file sizes in browsers [8, 14].
Inform Users	partially	C	Inform users of the energy-intensive operations on the website. *e.g.*, Autoplay feature on YouTube consumes a significant amount of energy, but the user is not informed.
Kill Abnormal Tasks	✔	S	Provide means of interrupting energy-hungry operations. *e.g.*, A timeout to interrupt an abnormal operation [31].
Manual Sync – On Demand	✔	S	Perform tasks exclusively when requested by the user. *e.g.*, YouTube, with Autoplay feature off, plays song only when user clicks on it.

...

Energy Patterns for Web: An Exploratory Study

Pooja Rani
rani@ifi.uzh.ch
University of Zurich
Zurich, Switzerland

Jonas Zellweger
jonas.zellweger@uzh.ch
University of Zurich
Zurich, Switzerland

Veronika Kousadianos
veronika.wu@students.unibe.ch
University of Bern
Bern, Switzerland

Luis Cruz
L.Cruz@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Timo Kehrer
timo.kehrer@unibe.ch
University of Bern
Bern, Switzerland, Switzerland

Alberto Bacchelli
bacchelli@ifi.uzh.ch
University of Zurich, Switzerland
Zurich, Switzerland

ABSTRACT

As the energy footprint generated by software is increasing at an alarming rate, understanding how to develop energy-efficient applications has become a necessity. Previous work has introduced catalogs of coding practices, also known as energy patterns. These patterns are yet limited to Mobile or third-party libraries. In this study, we focus on the Web domain—a main source of energy consumption. First we investigated whether and how Mobile energy patterns can be ported to this domain and found that 20 patterns could be ported. Then, we interviewed six expert web developers from different companies to challenge the ported patterns. Most developers expressed concerns for antipatterns, specifically with functional antipatterns, and were able to formulate guidelines to locate these patterns in the source code. Finally, to quantify the effect of Web energy patterns on energy consumption, we set up an automated pipeline to evaluate two ported patterns: 'Dynamic Retry Delay' (DRD) and 'Open Only When Necessary' (OOWN). With this, we found no evidence that the DRD pattern consumes less energy than its antipattern, while the opposite is true for OOWN. Data and Material: https://doi.org/10.5281/zenodo.8404487

CCS CONCEPTS

• **Software and its engineering → Empirical software validation.**

KEYWORDS

Green Software Engineering, Energy patterns, Web applications, Software sustainability, Coding Practices, Energy consumption

ACM Reference Format:
Pooja Rani, Jonas Zellweger, Veronika Kousadianos, Luis Cruz, Timo Kehrer, and Alberto Bacchelli. 2024. Energy Patterns for Web: An Exploratory Study. In *Software Engineering in Society (ICSE-SEIS'24), April 14–20, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3639475.3640110

ICSE-SEIS'24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0499-4/24/04.
https://doi.org/10.1145/3639475.3640110

LAY ABSTRACT

The information technology sector significantly affects the climate. With our increasing online activities, from chatting to accessing medical history, software powering these services requires to be energy-efficient. Researchers in software engineering have been exploring green coding practices, or energy-specific design patterns (aka energy patterns) to make software more eco-friendly. While such energy practices have been explored for other domains including Mobile, Web applications have been somewhat overlooked, despite our daily heavy internet use. We focused on the existing energy patterns from Mobile applications to Web applications. To validate these ported energy patterns, we interviewed six professional web developers from various companies. Then, we tested some patterns to see if these energy patterns indeed save any energy. Our results showed that developers are unaware of the energy practices and some patterns did not make a noticeable difference, while others consume more energy than their counterpart. In a nutshell, our work highlights the knowledge gap between green coding research and industry and emphasize the need to understand the trade-offs in energy practices for sustainable digital future.

1 INTRODUCTION

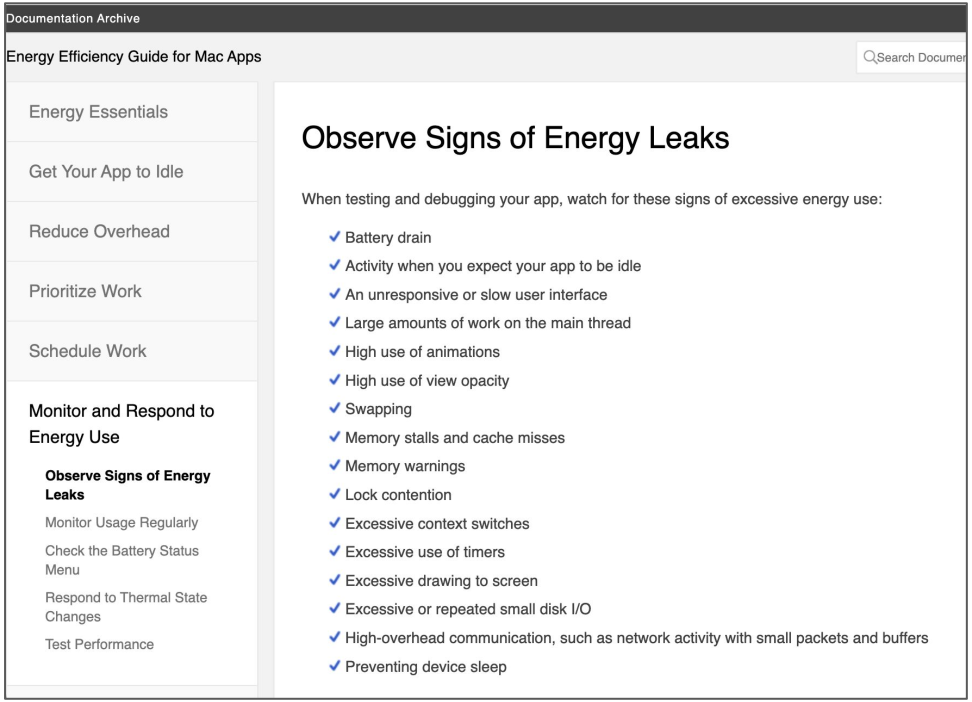
The ICT sector is estimated to generate up to 5.5% of world carbon emissions and to consume 20% of all electricity [3]. Indeed, from healthcare to communication, every industry prominently runs on software, thus understanding and developing energy-efficient software is urgent.

In this context, the Software Engineering (SE) research community has started investigating *green coding* and energy patterns for source code [17, 24]. Energy-specific design patterns for source code (henceforth, *Energy Patterns*) are best practices developers use to make their source code energy-efficient [24]. While researchers have developed catalogs of energy patterns for Mobile applications [10] and for deep learning libraries [35], some domains are still yet to be covered, prominently the Web domain, which is particularly relevant as its energy consumption is ever increasing [19].

Our goal is to gather and evaluate Web-specific energy patterns. To this aim, we first attempt to port existing Mobile energy patterns [10] to the Web domain. Then, to challenge our ported patterns, we discuss them with six professional Web developers, by means of in-depth structured interviews. In particular, we discuss how understandable these patterns are, how they are perceived, and whether they can be located in source code of Web applications.

P. Rani *et al.*, "Energy Patterns for Web: An Exploratory Study," 2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), Lisbon, Portugal, 2024, pp. 12-22.

Patterns for desktop apps



Documentation Archive

Energy Efficiency Guide for Mac Apps

Search Documen

- Energy Essentials
- Get Your App to Idle
- Reduce Overhead
- Prioritize Work
- Schedule Work
- Monitor and Respond to Energy Use
  - **Observe Signs of Energy Leaks**
  - Monitor Usage Regularly
  - Check the Battery Status Menu
  - Respond to Thermal State Changes
  - Test Performance

Observe Signs of Energy Leaks

When testing and debugging your app, watch for these signs of excessive energy use:

- ✓ Battery drain
- ✓ Activity when you expect your app to be idle
- ✓ An unresponsive or slow user interface
- ✓ Large amounts of work on the main thread
- ✓ High use of animations
- ✓ High use of view opacity
- ✓ Swapping
- ✓ Memory stalls and cache misses
- ✓ Memory warnings
- ✓ Lock contention
- ✓ Excessive context switches
- ✓ Excessive use of timers
- ✓ Excessive drawing to screen
- ✓ Excessive or repeated small disk I/O
- ✓ High-overhead communication, such as network activity with small packets and buffers
- ✓ Preventing device sleep

Energy Efficiency Guide for Mac Apps

Green Software Foundation’s patterns

Web patterns

- Avoid chaining critical requests
- Avoid an excessive DOM size
- Avoid tracking unnecessary data
- Defer offscreen images
- Deprecate GIFs for animated content
- Enable text compression
- Keep request counts low
- Minify web assets
- Minimize main thread work
- Optimize image size
- Remove unused CSS definitions
- Serve images in modern formats
- Use server-side rendering for high-traffic pages

Cloud patterns

- Cache static data
- Choose the region that is closest to users
- Compress stored data
- Compress transmitted data
- Containerize your workloads
- Delete unused storage resources
- Encrypt what is necessary
- Evaluate other CPU architectures
- Use a service mesh only if needed
- Terminate TLS at border gateway
- Implement stateless design
- Match utilization requirements of virtual machines (VMs)
- ...

IA patterns

- Optimize the size of AI/ML models
- Use efficient file formats for AI/ML development
- Run AI models at the edge
- Select a more energy efficient AI/ML framework
- Use energy efficient AI/ML models
- Use sustainable regions for AI/ML training
- Leverage pre-trained models and transfer learning for AI/ML development
- Select the right hardware/VM instance types for AI/ML training
- Adopt serverless architecture for AI/ML workload processes

🍃 Green smells 🍃

Pioneering smells for mobile apps

Investigating the Energy Impact of Android Smells

Antonin Carette[1], Mehdi Adel Ait Younes[1], Geoffrey Hecht[1,2], Naouel Moha[1], Romain Rouvoy[2,3]
[1] Université du Québec à Montréal, Canada
[2] University of Lille / Inria, France
[3] IUF, France
antonin.carette@gmail.com, ait_younes.mehdi_adel@courrier.uqam.ca, geoffrey.hecht@inria.fr, moha.naouel@uqam.ca, romain.rouvoy@inria.fr

***Abstract*—Android code smells are bad implementation practices within Android applications (or apps) that may lead to poor software quality. These code smells are known to degrade the performance of apps and to have an impact on energy consumption. However, few studies have assessed the positive impact on energy consumption when correcting code smells. In this paper, we therefore propose a tooled and reproducible approach, called HOT-PEPPER, to automatically correct code smells and evaluate their impact on energy consumption. Currently, HOT-PEPPER is able to automatically correct three types of Android-specific code smells: Internal Getter/Setter, Member Ignoring Method, and HashMap Usage. HOT-PEPPER derives four versions of the apps by correcting each detected smell independently, and all of them at once. HOT-PEPPER is able to report on the energy consumption of each app version with a single user scenario test. Our empirical study on five open-source Android apps shows that correcting the three aforementioned Android code smells effectively and significantly reduces the energy consumption of apps. In particular, we observed a global reduction in energy consumption by 4,83% in one app when the three code smells are corrected. We also take advantage of the flexibility of HOT-PEPPER to investigate the impact of three picture smells (bad picture format, compression, and bitmap format) in sample apps. We observed that the usage of optimised JPG pictures with the Android default bitmap format is the most energy efficient combination in Android apps. We believe that developers can benefit from our approach and results to guide their refactoring, and thus improve the energy consumption of their mobile apps.**

***Keywords*—Android, energy consumption, code smells, picture**

I. Introduction

Mobile devices have known a huge success along the last five years, for example Android's sales increased by more than 500% since 2011 [8]. With more than 50% of devices sold worldwide, Android has become one of the most popular operating system [6]. As the number of devices has increased, the number of applications (or *apps*) also grew rapidly along the last years. Therefore, the number of mobile developers also increases. Apps are mostly written using popular *Object-Oriented* (or OO) programming languages like Java, Objective-C, Swift or C#. Yet, mobile development is not as similar as traditional software development [54] and developers must consider the mobile specificities. Also, the user demand keeps increasing and forces mobile developers to add new features and maintain their apps as quickly as possible. Unfortunately, this pressure leads developers to adopt bad implementation practices, also known as *code smells* [28]. Code smells can lead to cause resources leaks in CPU, memory, battery, etc [25]. Leaks may deteriorate the quality of the app in terms of stability, user experience, maintainability, etc. It is also important to note that more than 18% of Android apps exhibit code smells [43]. Our previous studies have investigated the impact of code smells on performance and concluded that the correction of code smells improves the app performance [34]. In particular, the major code smells that impact the performance of Android apps are *HashMap Usage* (HMU), *Internal Getter/Setter* (IGS), and *Member Ignoring Method* (MIM) [2], [20], [34].
Like performance, the battery lifespan or energy consumption of an app is a critical quality criteria [1]. D. Li and W. Halfond [39] have proven that two of the three performance code smells listed above also have an energy impact on fictive app. However, these experiments were not performed on a real user app.

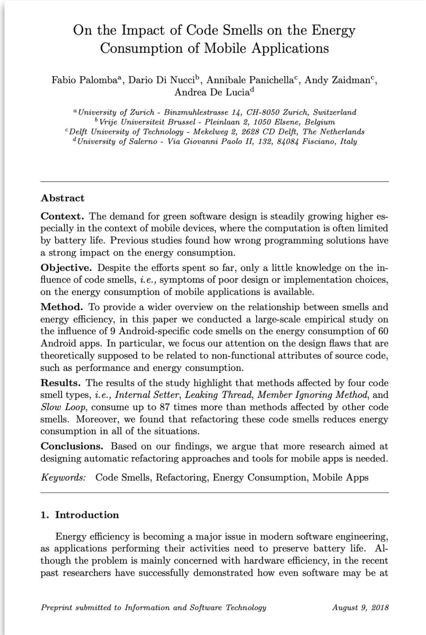
In this paper, we therefore propose an automated approach, called HOT-PEPPER, supported by a framework for Android developers that allows them to assess and improve the energy consumption of their Android apps. Concretely, HOT-PEPPER enables developers to detect and correct code smells, and evaluate their impact in terms of energy consumption in Android apps. HOT-PEPPER relies on PAPRIKA, a static analysis tool dedicated to the detection and correction of code smells in Android apps.

For the impact evaluation of code smells, HOT-PEPPER relies on the tool NAGA-VIPER, which uses a physical measurement device and monitors energy-related metrics (execution time, intensity, and voltage) on Android apps. For the validation of HOT-PEPPER, we performed an empirical study that allow us to answer to the following two research questions:

RQ$_1$: *Does the correction of Android code smells improve the energy consumption of the mobile phone?*
Finding: Yes, the correction of Android code smells improves the energy consumption of the mobile phone. We observed that the correction of at least one code smell reduces the energy consumption of the mobile phone. Moreover, the correction of all code smells reduces the energy consumption even more significantly.
RQ$_2$: *Do picture smells have an impact on the energy consumption of the mobile phone?*
Finding: Yes, studied picture smells have a bad impact on the energy consumption of the mobile phone. We observed that

HashMap Usage
Internal Getter/Setter
Member Ignoring Method
+
“picture smell” 💡

A. Carette, M. A. A. Younes, G. Hecht, N. Moha and R. Rouvoy, "Investigating the energy impact of Android smells," *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Klagenfurt, Austria, 2017, pp. 115-126,



On the Impact of Code Smells on the Energy Consumption of Mobile Applications

Fabio Palomba[a], Dario Di Nucci[b], Annibale Panichella[c], Andy Zaidman[c], Andrea De Lucia[d]

[a] *University of Zurich - Binzmuhlestrasse 14, CH-8050 Zurich, Switzerland*
[b] *Vrije Universiteit Brussel - Pleinlaan 2, 1050 Elsene, Belgium*
[c] *Delft University of Technology - Mekelweg 2, 2628 CD Delft, The Netherlands*
[d] *University of Salerno - Via Giovanni Paolo II, 132, 84084 Fisciano, Italy*

Abstract

Context. The demand for green software design is steadily growing higher especially in the context of mobile devices, where the computation is often limited by battery life. Previous studies found how wrong programming solutions have a strong impact on the energy consumption.

Objective. Despite the efforts spent so far, only a little knowledge on the influence of code smells, *i.e.,* symptoms of poor design or implementation choices, on the energy consumption of mobile applications is available.

Method. To provide a wider overview on the relationship between smells and energy efficiency, in this paper we conducted a large-scale empirical study on the influence of 9 Android-specific code smells on the energy consumption of 60 Android apps. In particular, we focus our attention on the design flaws that are theoretically supposed to be related to non-functional attributes of source code, such as performance and energy consumption.

Results. The results of the study highlight that methods affected by four code smell types, *i.e., Internal Setter, Leaking Thread, Member Ignoring Method,* and *Slow Loop*, consume up to 87 times more than methods affected by other code smells. Moreover, we found that refactoring these code smells reduces energy consumption in all of the situations.

Conclusions. Based on our findings, we argue that more research aimed at designing automatic refactoring approaches and tools for mobile apps is needed.

Keywords: Code Smells, Refactoring, Energy Consumption, Mobile Apps

1. Introduction

Energy efficiency is becoming a major issue in modern software engineering, as applications performing their activities need to preserve battery life. Although the problem is mainly concerned with hardware efficiency, in the recent past researchers have successfully demonstrated how even software may be at

Preprint submitted to Information and Software Technology *August 9, 2018*

Internal Setter
Leaking Thread
Member Ignoring Method
Slow Loop

Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, Andrea De Lucia, On the impact of code smells on the energy consumption of mobile applications, *Journal of Information and Software Technology*, Volume 105, 2019, Pages 43-55,

Mobile-specific code smells: a taxonomy

🔄 Alternative
Check battery-efficient APIs that have been specifically designed to substitute regular APIs

🚰 Leakage
Make sure that an acquired resource is always released, to avoid unnecessary battery drain

⌛ Bottleneck
Avoid accumulation of data or operations that will require an energy peak to be processed

🪫 Power
Operations driven by the battery status help prolong the battery life

⚖️ Sobriety
Make reasonable accommodations between user experience and more energy-efficient variants

💤 Idleness
When the app enters an idle state, reduce the workload accordingly

📦 Batch
Grouping individually costly operations allows saving energy globally

⚙️ Release
Favor the compile-time tasks that decrease the energy footprint of the deployment of the app

👴 Longevity
Pay attention to old and low-end devices

https://github.com/cnumr/best-practices-mobile (40+ green code smells for Android)

Bottleneck code smell example

👃 **Internet-in-the-loop (IITL)**
“Internet connection should not be opened in loops to preserve the battery”

🩹 **Repair**

- Quick fix: use advanced Android API components to do the job (e.g. DownloadManager)
- Long-term fix: rethink the dialogue between the client and the server (e.g. REST API endpoints)

🔎 **Occurrence**

You think nobody does that? Yet, around 4%* of OSS Android projects have this issue

Android code snippet

```
HttpURLConnection con;
URL myURL = null;

try {
  for (int i=0; i<20; i++) {
    myURL = new URL("http://myserver.com/file" + i);
    con = (HttpURLConnection) myURL.openConnection();
    //coof stuff here
    con.disconnect();
  }
} catch() {
  e.printStackTrace();
}
```

*preliminary results of an empirical study currently underway (LaBRI × LIUPPA)

🕹️ live demo

Smell detection & hotspots

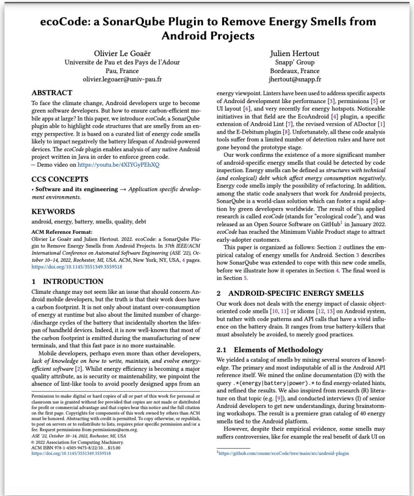
A (bad) green code smell is a **potential energy-inefficiency issue**. It has to be spotted and then discussed by the development team, and ultimately, fixed

The purpose of a lint-like tool is to highlight 🔎 code smells in a codebase

Enforcing Green Code With Android Lint

Olivier Le Goaër
Computer Science Laboratory (LIUPPA)
University of Pau
Pau, France
olivier.legoaer@univ-pau.fr

Olivier Le Goaër. 2021. Enforcing green code with Android lint. *In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 85–90.



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

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)*. Association for Computing Machinery, New York, NY, USA

Food for thought

Good smell, Bad Smell

By nature, code quality tools are solely focused on **bad** code smells (penalties in the quality score)

Sustainability calls for supporting also **good** code smells, as **rewards** 🍭 to raise the climate-consciousness of development teams

Incidentally, good points may mitigate bad points in the final quality score

Model-driven green code smell detection

Problem

Help green code smells face technological **heterogeneity** at both levels: mobile platforms and static code analysis tools

Solution (PoC)

“*write once, ~~run~~ detect everywhere*”

DSL & code generation (MBSE principles)

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.

MODELS Companion ’24, September 22–27, 2024, Linz, Austria

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

Léa Brunschwig and Olivier Le Goaër. 2024. Cross-Detection of Mobile-specific Energy Hotspots: MBSE to the Rescue. In Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24). ACM, New York, NY, USA, 518–522.

The era of AI-powered detection

On Android, “Use a cache” is a well-known best practice, but has various meanings and seems **undetectable** because it is implementation-free:

- cached HTTP requests
- memoized functions
- recycled views (a.k.a. “ViewHolders”)
- …

Visiting nodes of an abstract syntax tree is doing things the old way. Artificial Intelligence (LLM) is **the future of code smell detection** (and fix thereof) 🧠

ChatOpenAI

ecoCode + brain = ecoBrain

```
ggrousset@MacBook-Pro-de-Gilles  ~/Desktop/ecobrain  poetry run ecobrain check FibonacciCache.java    ✓  10320  12:04:12
import java.util.HashMap;
import java.util.Map;

public class FibonacciCache {  // [ecobrain] The caching system is initialized here as a HashMap. This will store computed Fibonacci numbers to avoid redundant computations.
    private Map<Integer, Long> cache;

    public FibonacciCache() {
        cache = new HashMap<>();
    }

    public long getFibonacci(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Negative numbers are not allowed.");
        }
        if (n <= 1) {
            return n;
        }
        if (cache.containsKey(n)) {
            return cache.get(n);
        }
        long result = getFibonacci(n - 1) + getFibonacci(n - 2);
        cache.put(n, result);
        return result;  // [ecobrain] Here, if the Fibonacci number has already been computed (i.e., it exists in the cache), it is returned from the cache instead of being recomputed.
    }  // [ecobrain] This line is where the cache is accessed to retrieve the already computed Fibonacci number.

    public static void main(String[] args) {  // [ecobrain] If the Fibonacci number hasn't been computed before, it's computed and then stored in the cache for future usage.
        FibonacciCache fibonacciCache = new FibonacciCache();
        int n = 50; // You can change this value to test with other numbers
        long startTime = System.nanoTime();
        long fibonacciValue = fibonacciCache.getFibonacci(n);
        long endTime = System.nanoTime();
        System.out.println("Fibonacci of " + n + " is " + fibonacciValue);
        System.out.println("Computed in " + (endTime - startTime) / 1_000_000 + " milliseconds");
    }
}
```

Successful attempt to automatically detect a memoized Java method (here Fibonacci). Thanks Gilles G. :)   Source code

Conclusion

Takeaways for green software practitioners

Building Green Software is not building software **as usual**

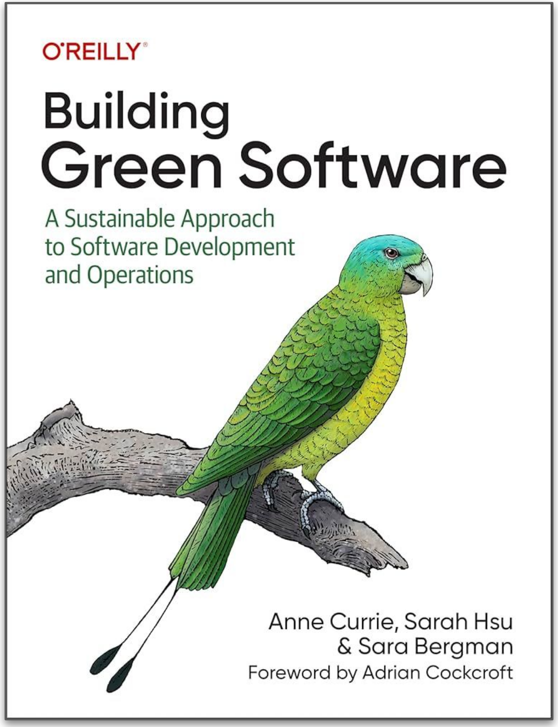
Green Software Engineering (GSE) must provide **actionable practices**

Green tactics, patterns and smells embody **3 levels** of best practices

Green software only has an impact at **scale**, and **automation** is key

Greener Is Coming...

🇫🇷 Government task force (2023)



🇺🇸 Potential bestseller (2024)