



olivierlegoaer



GEDR

Groupeement  
de recherche

GPL Génie de la programmation  
et du logiciel



IVERSITÉ  
DE PAU ET DES  
PAYS DE L'ADOUR

# The road to green code (with *Sonar*)

# The limits to (software) growth

How it started (2011)



**Mark Andreessen**  
founder of Netscape,  
renowned Venture Capitalist  
Andreessen-Horowitz

Software is eating the  
world, in all sectors

In the future every  
company will become a  
**software** company

How it's going (2024)



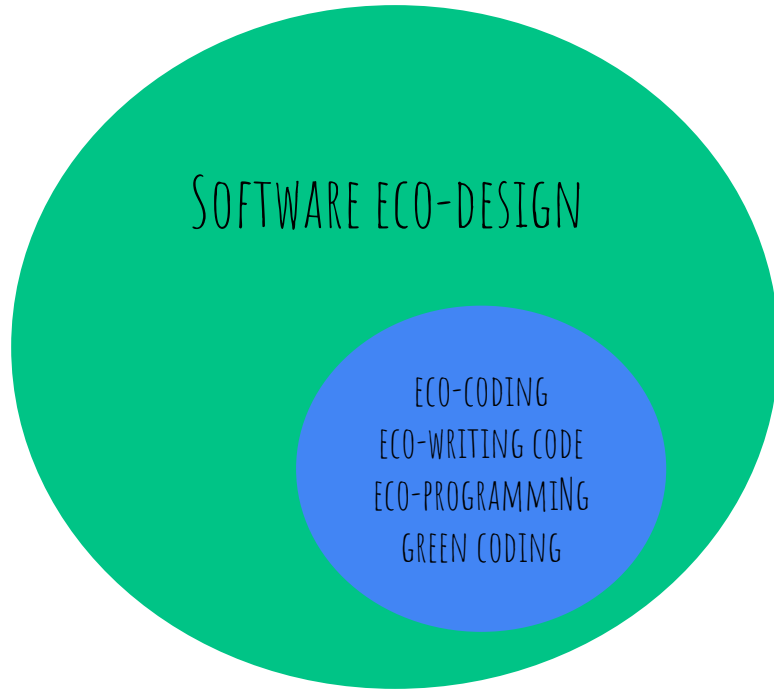
**Optimiser le logiciel  
d'un facteur 2 tous  
les 2 ans**

En optimisant le logiciel d'un facteur 2 tous les deux ans, on libère de la puissance informatique avec laquelle on peut inventer de nouveaux usages.

C'est comme la loi de Moore, mais **sans changer le matériel !**

©Tristan Nitot

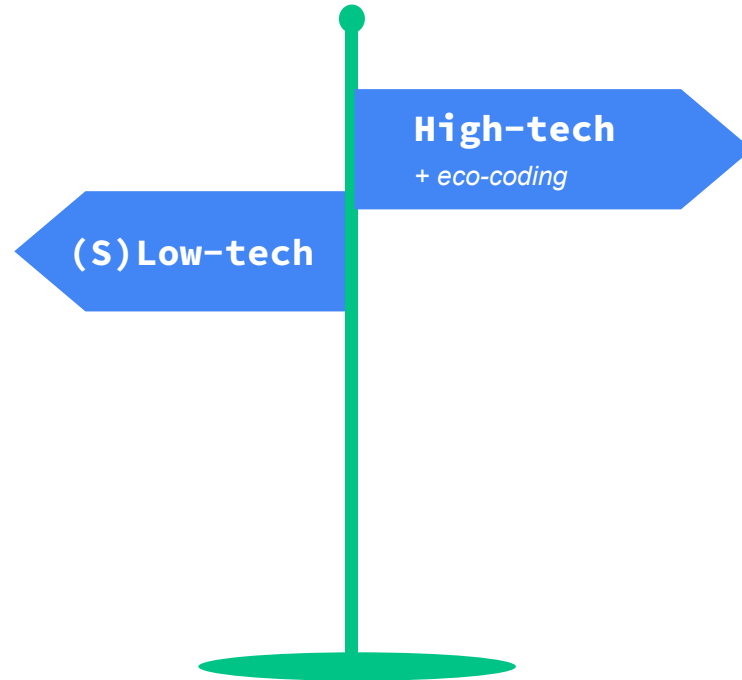
# The software eco-\*



●

*"the most responsible software is  
the one we don't build"*

# Fork in the road



## First law of eco-coding

$$\text{energy} = (\text{more code})^2$$

$$e = mc^2$$

# Energy versus Performance

Computer/Device	A	B
Energy (in joules)	30	20
Time (in seconds)	10	20

Energy-efficiency vs Run-time-efficiency

# Basic eco-coding incentives

## 💰 Money

The fewer resources SmartContracts\* consume, the lower the costs

```
contract ERC20 is Context, IERC20, IERC20Metadata {  
    mapping(address => uint256) private _balances;  
  
    mapping(address => mapping(address => uint256)) private _allowances;  
  
    uint256 private _totalSupply;  
  
    string private _name;  
    string private _symbol;
```

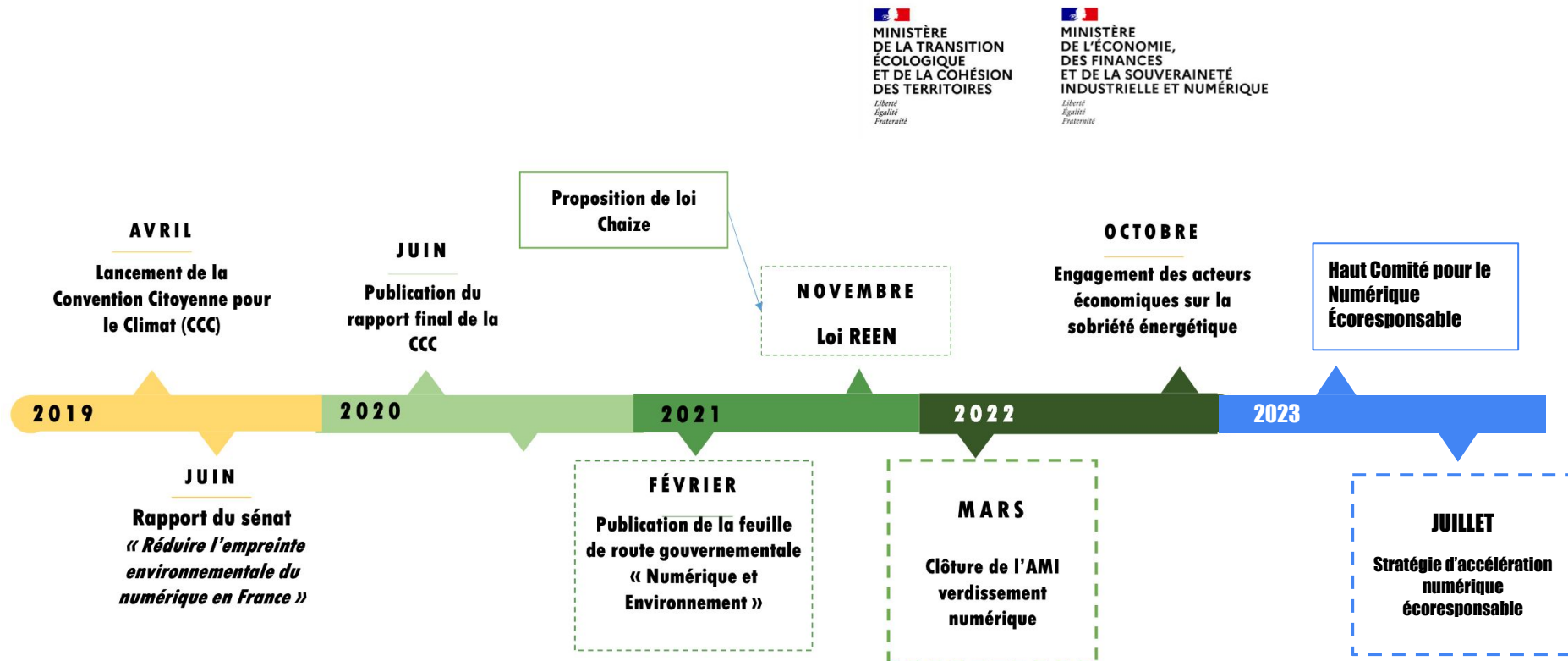
\*programs running within a Blockchain

## ★ Reputation

Bad reviews left on app stores can ruin your business



# French roadmap





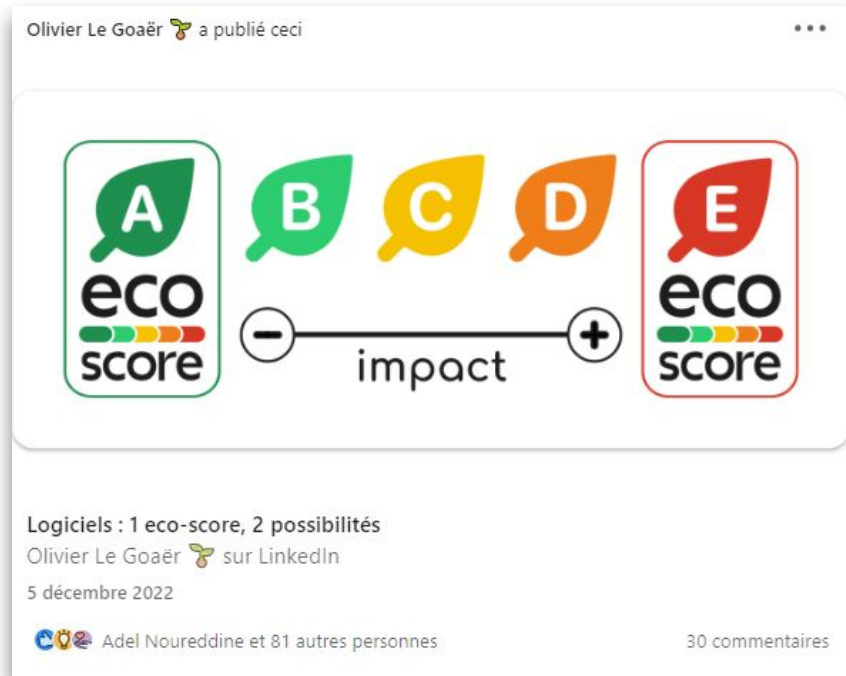
# Towards an eco-score...

## cyber-score (effective in oct. 2023)



[source](#)

## eco-score (elusive goal)

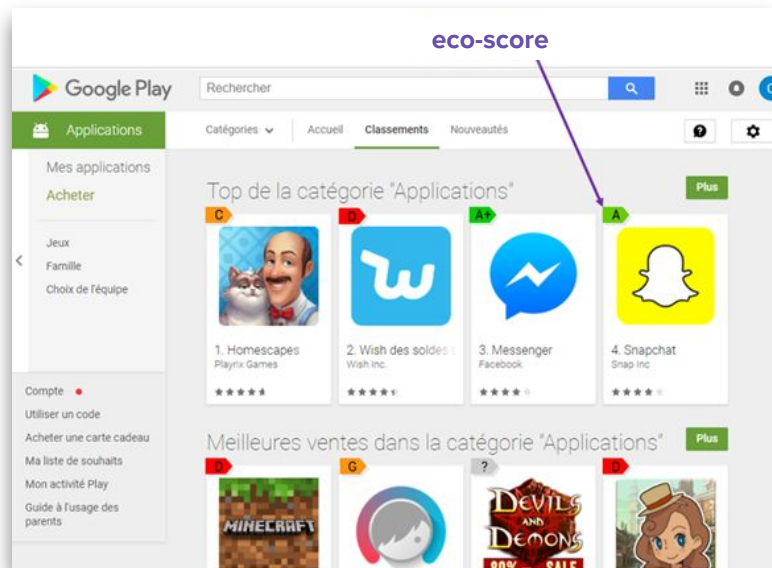


[source](#)

# What if an eco-score?

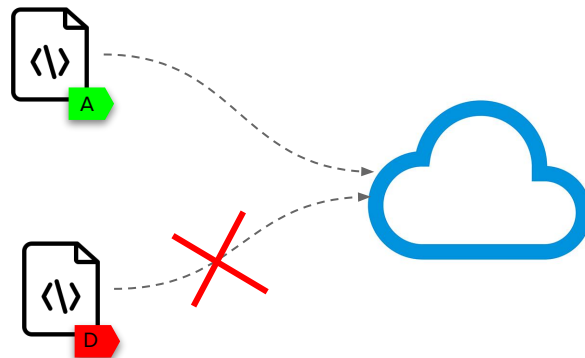
## Information

App stores display the eco-score to the end-users (and include it in their ranking algorithm)

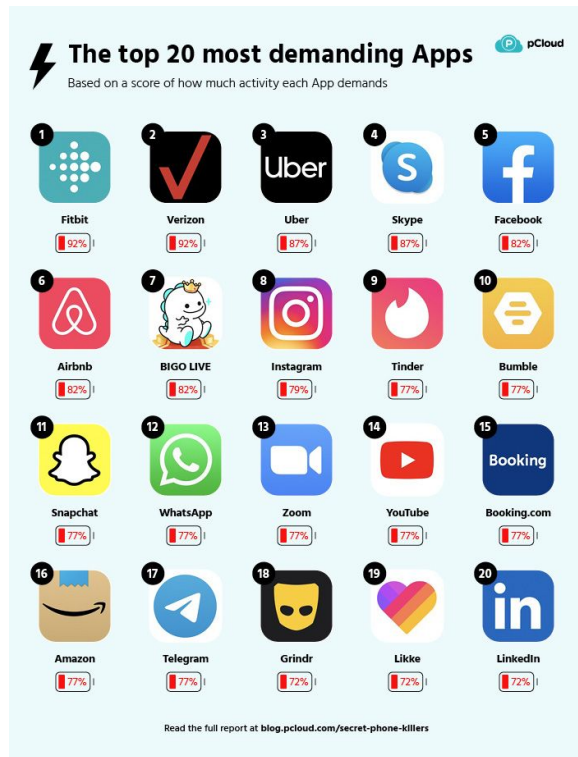


## Regulation

(Truly sustainable) Cloud providers refuse the deployment of program lower than D

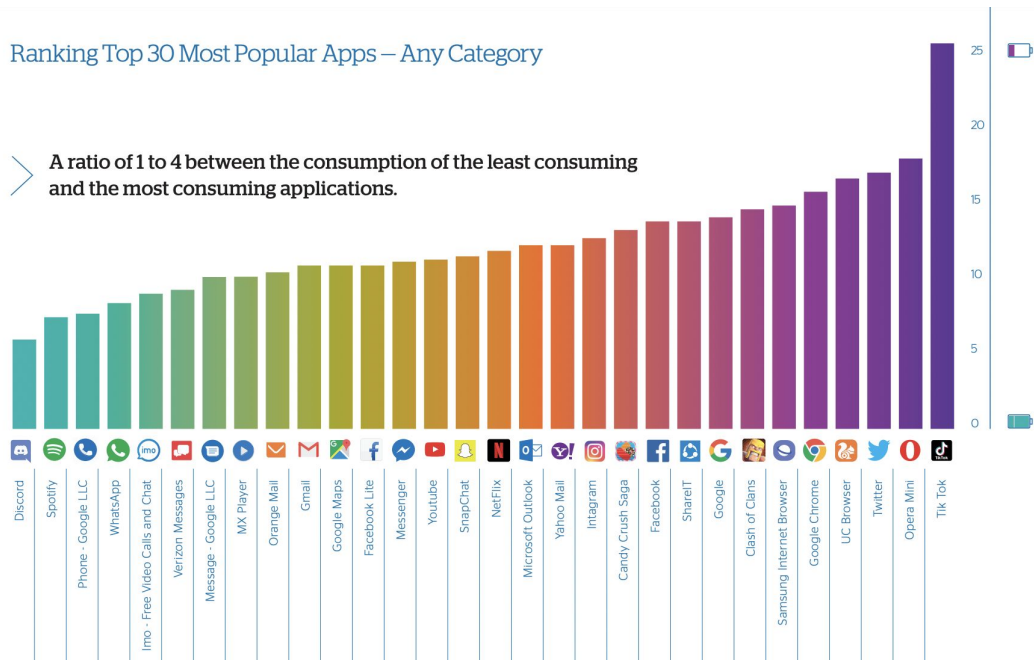


# Ranking eco-friendly apps

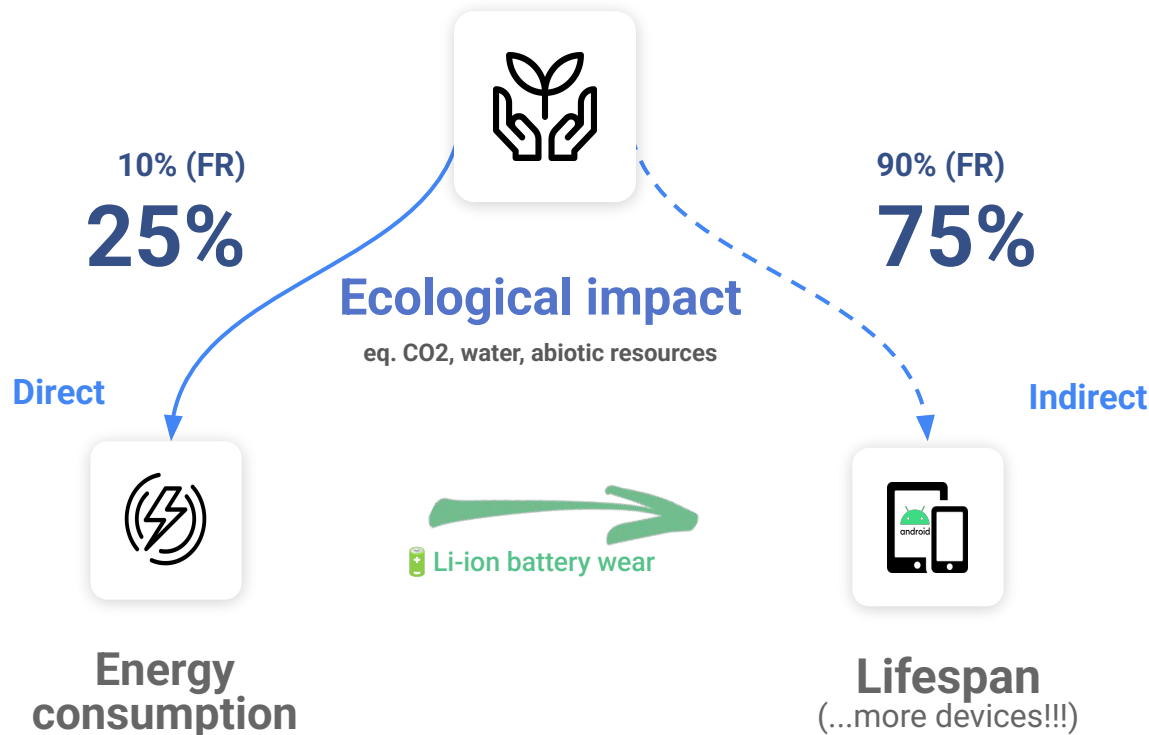


Ranking Top 30 Most Popular Apps – Any Category

A ratio of 1 to 4 between the consumption of the least consuming and the most consuming applications.



# Think global & mobile-first



**Limited impacts**

# Green programming languages?

You don't always have a choice!

What about the runtime?

Mobile apps are programs, but rarely algorithms\*

**Programming language: there is no silver bullet!**

	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.

\*mathematically provable object

## Anti-patterns and the energy efficiency of Android applications



# Code smells: The new challengers

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

Fabio Palomba<sup>a</sup>, Dario Di Nucci<sup>b</sup>, Annibale Panichella<sup>c</sup>, Andy Zaidman<sup>c</sup>,  
Andrea De Lucia<sup>d</sup>

<sup>a</sup>University of Zurich - Binzmühlestrasse

<sup>b</sup>Vrije Universiteit Brussel - Plein

<sup>c</sup>Delft University of Technology - Mithras

<sup>d</sup>University of Salerno - Via Giovanni

### Abstract

**Context.** The demand for green software is increasing, especially in the context of mobile devices, by battery life. Previous studies found a strong impact on the energy consumption of mobile applications.

**Objective.** Despite the efforts spent on the energy consumption of mobile applications, the energy consumption of mobile applications remains a challenge.

**Method.** To provide a wider overview of energy efficiency, in this paper we conduct an analysis of 9 Android-specific code smells in Android apps. In particular, we focus on the influence of 9 Android-specific code smells on the energy consumption of mobile applications, theoretically supposed to be related to energy consumption, such as performance and energy consumption.

**Results.** The results of the study highlight that some code smells, i.e., *Internal Setter*, *Leaking Slow Loop*, consume up to 87 times more energy than others.

**Conclusions.** Based on our findings, we designed automatic refactoring approaches to reduce energy consumption in all of the situations.

**Keywords:** Code Smells, Refactoring, Energy Consumption

### 1. Introduction

Energy efficiency is becoming a major concern for applications performing their activities on mobile devices. Although the problem is mainly concerned with battery life, previous researches have successfully demonstrated the impact of code smells on the energy consumption of mobile applications.

Preprint submitted to Information and Software

## Energy Refactorings for Android in the Large and in the Wild

Marco Couto  
HISLAB@INESC TEC

João Saraiva  
HISLAB@INESC TEC

João Paulo Fernandes  
CISUC

inho, Portugal  
inho@pt

Universidade de Coimbra, Portugal  
jp@dei.uc.pt

### Android Code Smells: From Introduction to Refactoring

Sarra Habchi<sup>a,\*</sup>, Naouel Moha<sup>b,1</sup>, Romain Rouvoy<sup>c,1</sup>

<sup>a</sup>University Of Luxembourg

<sup>b</sup>Université du Québec À Montréal

<sup>c</sup>University of Lille

### Abstract

Object-oriented code smells are well-known concepts in software engineering that refer to bad design and development practices commonly observed in software systems. With the emergence of mobile apps, new classes of code smells have been identified by the research community as mobile-specific code smells. These code smells are presented as symptoms of important performance issues or bottlenecks. Despite the multiple empirical studies about these new code smells, their diffuseness and evolution along change histories remains unclear.

We present in this article a large-scale empirical study that inspects the introduction, evolution, and removal of Android code smells. This study relies on data extracted from 324 apps, a manual analysis of 561 smell-removing commits, and discussions with 25 Android developers. Our findings reveal that the high diffuseness of mobile-specific code smells is not a result of releasing pressure. We also found that the removal of these code smells is generally a side effect of maintenance activities as developers do not refactor smell instances even when they are aware of them.

### 1. Introduction

Mobile apps have established themselves as mainstream software systems deployed at scale. Over the last few years, they successfully invaded the

\*Corresponding author

Email addresses: sarra.habchi@uni.lu (Sarra Habchi), moha.naouel@uqam.ca (Naouel Moha), romain.rouvoy@unil.fr (Romain Rouvoy)

Preprint submitted to Elsevier

October 15, 2020

improve energy consumption has already presented promising research results [4]–[13]. These results, however, have essentially been validated by testing code patterns individually and then in a small set of applications (sometimes only in one).

In this paper, we consider 11 energy-greedy code patterns obtained from the literature, described in detail in Section III. We conduct a study over a large-scale repository of 600+ Android applications to understand the frequency of occurrence of such patterns. Within the 200+ applications where the patterns were detected, we studied the impact that replacing them, individually and combined, by their documented alternatives has on the energy consumption. Moreover, as we consider all the possible combinations of the individual patterns, this resulted in 400+ refactored applications under analysis.

To perform our study, we developed an extensible, fully automated framework called *Chimera*, which is able to detect and refactor the patterns. Each pattern is considered individually and is also combined with all the other patterns. *Chimera* also measures the energy consumed by an application in different simulated usage scenarios, before and after refactoring.

In summary, the main contributions of this work are:  
1) An analysis of how energy-greedy patterns proposed in the literature are distributed over a large-scale repository of Android applications. This is described in Sections IV and V;  
2) A reusable prototype of a pattern-oriented testing framework (*Chimera*), described in Section VI-B, for the detection, filtering, and refactoring of patterns in Android applications; it can also run a set of usage scenarios on such applications, while collecting metrics such as energy consumption;

3) An empirical study, described in Section VI, to assess the energy impact of applying refactorings. We analyze, for each code pattern and combination of patterns, the test results for the Android applications on which they occur, and compare the obtained gains between each pattern/combination.

Using the results of the empirical study referred in 3), we aim at answering the following research questions:

RQ1: Do all individual refactorings consistently lead to energy savings?  
RQ2: Do all individual refactorings lead to energy savings of the same magnitude?

RQ3: What are the refactorings that, individually or when combined, produce the higher energy savings?

RQ4: When refactoring for energy efficiency, what approach should developers follow?

- Internal Setter
- Leaking Thread
- Leaking Inner Class
- Member Ignoring Method
- No Low Memory Resolver
- Hashmap Usage
- Init OnDraw

• • •



**Greater impacts**

# Android-specific matters

Battery-killers are nestled at the platform-level, not the language-level

Every Android project has a well-defined, meaningful structure

There are lots of interesting things to inspect:



# Energy-greedy components

Hardware-related Component	Avg. energy consumption (J)
display	139.784567875382
camera	84.1856142588254
microphone	81.8998646885348
gravity	71.3078291080087
magnetic_field	69.6877663025097
gyroscope	69.3777997221
accelerometer	67.9535327322522
cpu	66.6925401713931
room_database	66.0762976599094
speaker	65.6659164078901
gps	65.6478179873468
local_storage	64.5536233840085
ambientlight	63.0030057575923
networking	62.6477966616013

## A Framework for the Automatic Execution of Measurement-based Experiments on Android Devices

Ivano Malavolta<sup>1</sup>, Eoin Martino Grua<sup>1</sup>, Cheng-Yu Lam<sup>1</sup>, Randy de Vries<sup>1</sup>, Franky Tan<sup>1</sup>, Eric Zielinski<sup>1</sup>, Michael Peters<sup>2</sup>, Luuk Kaandorp<sup>1</sup>

<sup>1</sup> Vrije Universiteit Amsterdam, The Netherlands. l.malavolta@vu.nl, e.m.grua@vu.nl, c.lam@student.vu.nl, randy.de.vries@student.vu.nl, k.h.tan@vu.nl, e.a.zielinski@student.vu.nl, l.kaandorp@student.vu.nl

<sup>2</sup> M2mobi, The Netherlands. m.peters@m2mobi.com

### ABSTRACT

Conducting measurement-based experiments is fundamental for assessing the quality of Android apps in terms of, e.g., energy consumption, CPU, and memory usage. However, orchestrating such experiments is not trivial as it requires large boilerplate code, careful setup of measurement tools, and the adoption of various empirical best practices scattered across the literature. All together, those factors are slowing down the scientific advancement and harming experimenters' replicability in the mobile software engineering area.

In this paper we present Android Runner (AR), a framework for automatically executing measurement-based experiments on native and web apps running on Android devices. In AR, an experiment is defined once in a descriptive fashion, and then its execution is fully automatic, customizable, and replicable. AR is implemented in Python and it can be extended with third-party profilers.

AR has been used in more than 25 scientific studies primarily targeting performance and energy efficiency.

### 1 INTRODUCTION

Android is the leading mobile platform today and the majority of scientific contributions on mobile software engineering focuses on Android [1]. When dealing with quality properties like energy efficiency and performance, practitioners and researchers rely on the measurement of run-time metrics such as battery discharge, CPU and memory usage, number and type of network requests, etc. [7, 9, 10]. In this context, considerable effort and time are spent on setting up infrastructures and software pipelines for conducting measurement-based experiments. Moreover, when available, existing pipelines are either (i) ad-hoc for a specific experiment or (ii) tailored to one specific quality property (e.g., energy consumption).

This paper presents Android Runner (AR), a framework to automatically execute measurement-based experiments on native and web apps. The main goal of AR is to streamline the execution of measurement-based experiments involving Android devices. In AR,

experiments are defined in a descriptive fashion, and then their execution is fully automatic, customizable, and replicable. We designed AR with the following design drivers in mind:

- **Automation**: after an initial configuration, the experiment can be executed without any interaction from the user;
- **Incremental experiments**: AR always persists the intermediate results of the experiment and, if interrupted, it is able to resume it and continue with the remaining runs;
- **Usability**: users define the experiment in a descriptive manner, without writing boilerplate code or knowing the internals of AR;
- **Customizability**: users have the possibility to include their own business logic and automated testing tools [6] at specific points within the experiment execution (e.g., before the whole experiment begins, before or after each run, etc.);
- **Profiler independence**: in AR, run-time measures can be collected both via hardware (e.g., the Monsoon power monitor<sup>2</sup>) and software (e.g., Trepp<sup>3</sup>). Profilers can produce different data points and can interact with apps and the Android device in their own way; moreover, AR makes straightforward to use multiple profilers within a single experiment;
- **Experiments replicability**: given the configuration, subject apps, and available Android devices, AR can execute an already-performed experiment with low effort, even if the experiment has been performed by a third party.

We are aware that frameworks like AR must be as accurate as possible and that their accuracy must be independently verifiable. In order to facilitate the validation of AR, we created a set of 27 **benchmarking apps**, each of them stressing a specific hardware component of an Android mobile device, such as its accelerometer, camera, CPU, display, GPS, etc. We use those apps on a regular basis for validating the accuracy of AR. The full set of benchmarking apps is publicly available<sup>4</sup> (together with their source code) and can be reused by researchers in the area of mobile software engineering, also independently of AR.

The **target audience** of AR includes (i) researchers who need to conduct empirical evaluations of software engineering methods and techniques involving Android apps, (ii) researchers developing new run-time profilers for Android devices, and (iii) practitioners needing to quantitatively assess the quality of their own apps.

The remainder of the paper is organized as follows. Section 2 provides an overview of the proposed framework. Section 3 presents its

<sup>1</sup><https://github.com/S2-group/android-runner>

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 ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

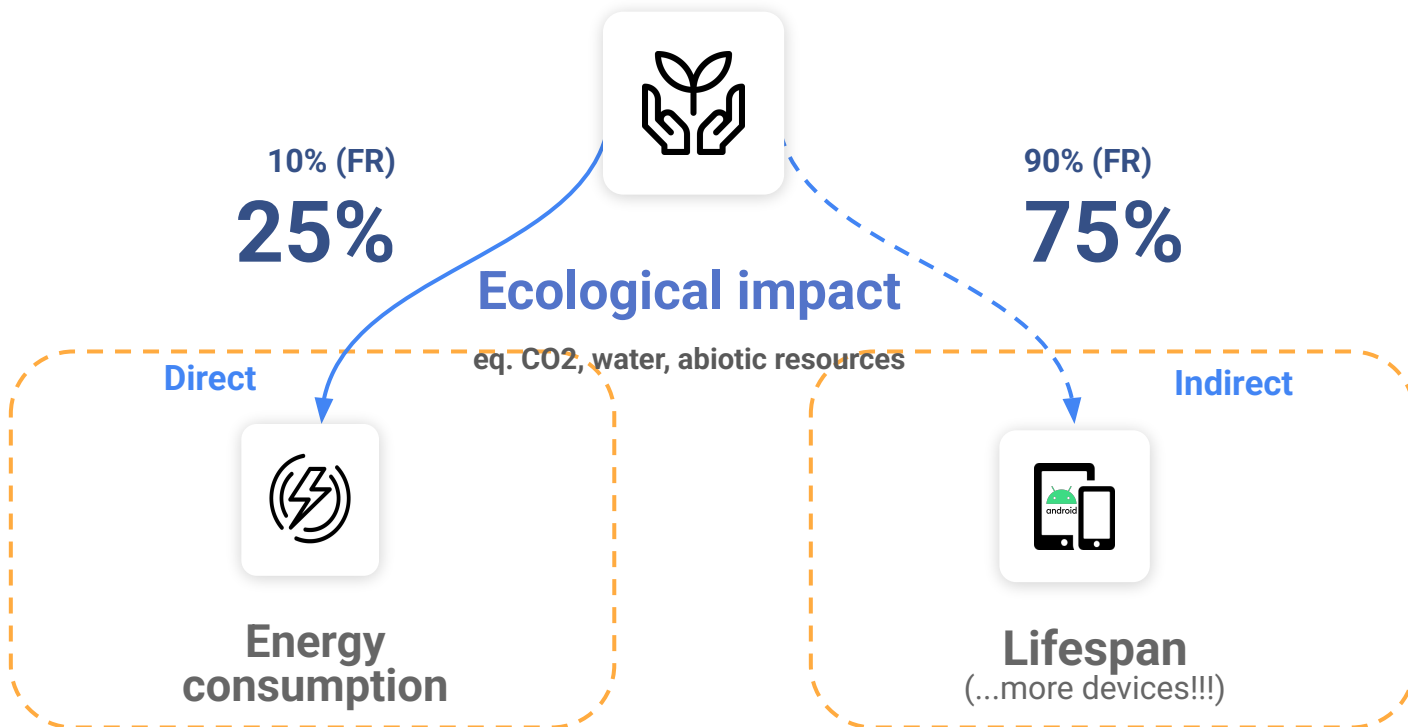
ACM 20: September 21–23, 2020, Virtual Event, Australia  
© 2020 Association for Computing Machinery.  
ACM ISBN 978-1-4503-4228-0/20/09...\$15.00  
<https://doi.org/10.1145/3471133.3422844>

<sup>2</sup><https://www.monsoon.com/high-voltage-power-monitor>

<sup>3</sup><https://developer.qualcomm.com/forums/software-trepp-power-profiler>

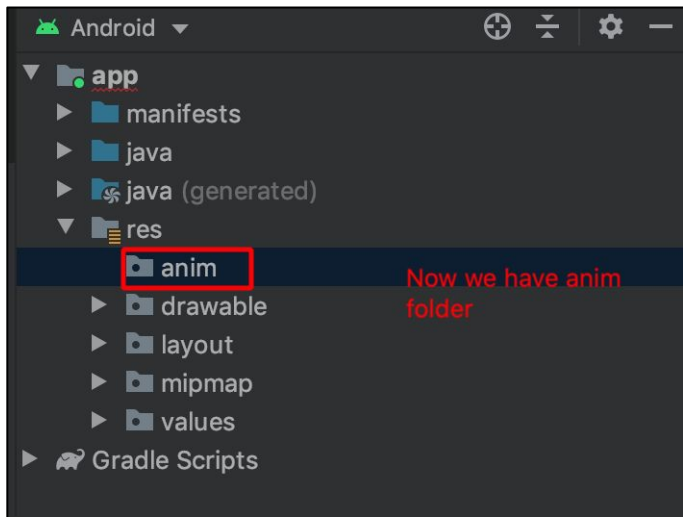
<sup>4</sup><https://github.com/S2-group/android-apps-benchmark>

# Back to the 2 scopes



# Scope #1: energy consumption

✂️ Avoid extraneous animation



zzz Avoid keep screen on

```
public class MainActivity extends Activity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);  
    }  
}
```



# Scope #2: device lifespan



Fight software obesity



```
android {  
    defaultConfig {  
        ...  
        minSdkVersion 15  
        targetSdkVersion 33  
        multiDexEnabled true  
    }  
    ...  
}  
  
dependencies  
    implementation "androidx.multidex:multidex:2.0.1"  
}
```



Support aging devices



```
<uses-sdk android:minSdkVersion="integer"  
          android:targetSdkVersion="integer"  
          android:maxSdkVersion="integer" />
```

**Green code  
smells**

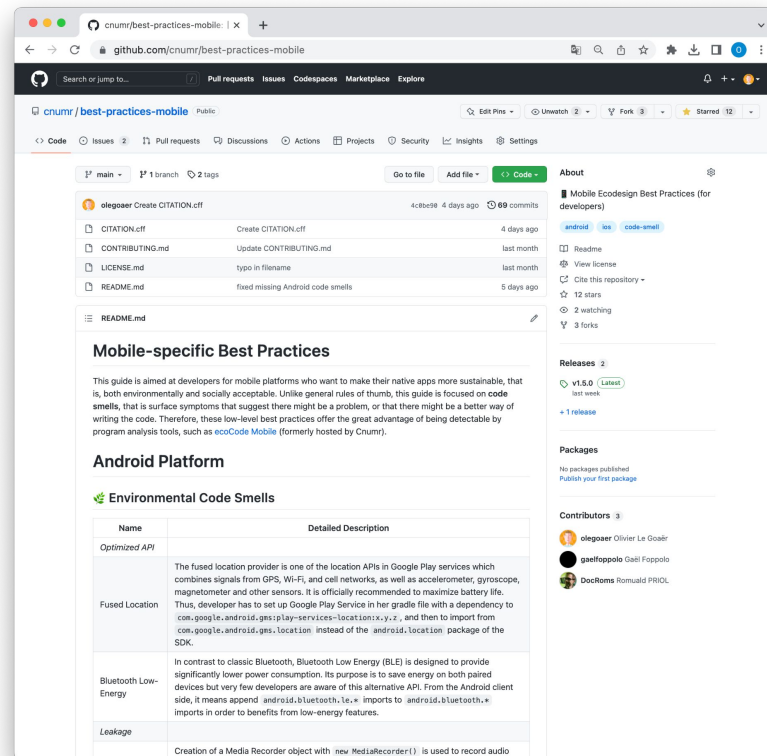
# Open source catalog

License CC BY-NC-ND

40+ code smells arranged in 9 categories,  
crosscutting scope 1 & 2

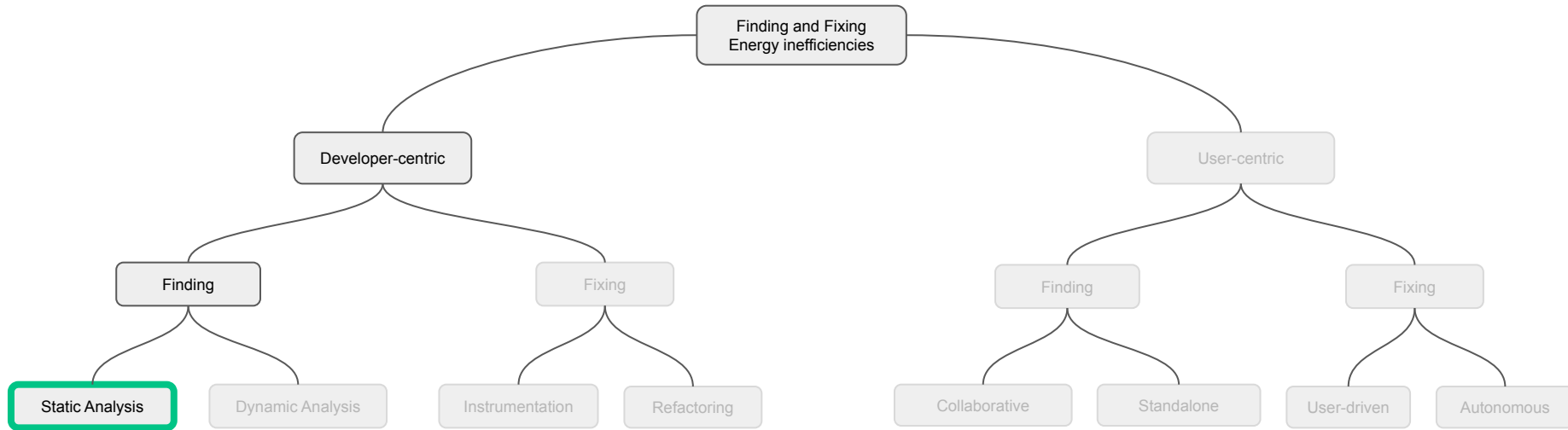
Description based on Java (but Kotlin-ready)

<https://github.com/cnumr/best-practices-mobile>



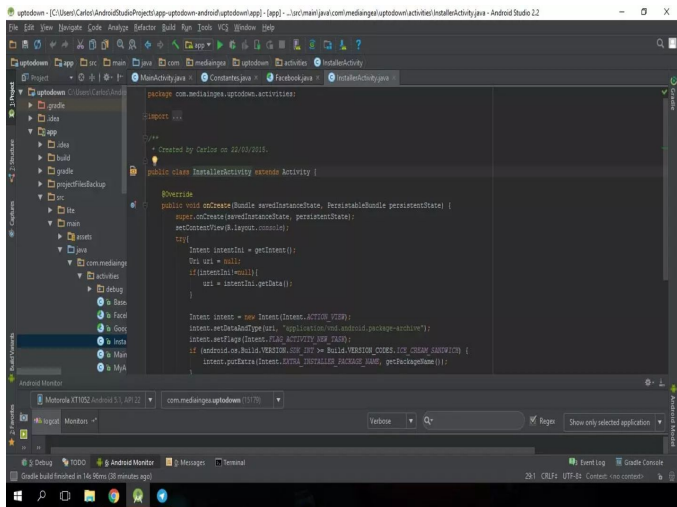


# R&D taxonomy



# Introducing *ecoCode*

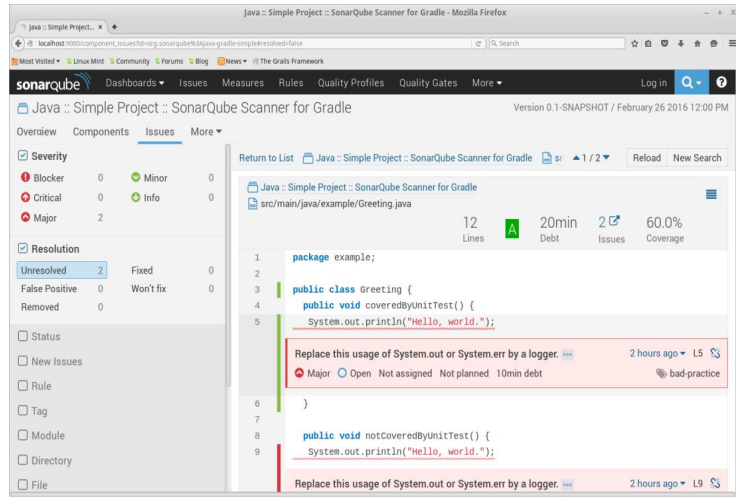
# Rationales



android  
studio



No guidelines on how to  
write energy-friendly apps



sonarqube



World-class solution to  
improve code quality

# “Green as You Code” sounds good

## 2-step implementation

It takes 2 simple steps to implement Clean as You Code.



### Quality Gate on New Code

A Quality Gate focused only on metrics for New Code – added or changed – prevents new issues from creeping in. Sonar sets this by default and aligns developers across the organization to deliver to that standard.



### don't release unless it's green

The only rule that needs to be applied is the common organizational understanding that no project will be released to production if it's failing its Quality Gate.

# Screenshots\*



The screenshot shows the 'Issues' page for a project named 'MyApp' in the SonarQube interface. The browser address bar shows the URL 'sonarqube.ecocode.io/project/issues?id=MyApp&resolved...'. The page header includes navigation tabs for 'Overview', 'Issues', 'Security Hotspots', 'Measures', 'Code', and 'Activity'. The 'Issues' tab is active, displaying a list of code quality issues. A sidebar on the left contains filters for 'Type' (CODE SMELL) and 'Severity' (Blocker, Minor, Critical, Info, Major). The main content area lists several issues, all of which are 'Code Smell' type and 'Major' severity. The first issue is 'Battery optimization should not be ignored.' with a severity of 'L8' and a creation time of '21 hours ago'. The second issue is 'Use com.google.android.gms.location instead of android.location to maximize battery life.' with a severity of 'L5' and a creation time of '21 hours ago'. The third issue is 'Use com.google.android.gms.location instead of android.location to maximize battery life.' with a severity of 'L6' and a creation time of '21 hours ago'. The fourth issue is 'Use com.google.android.gms.location instead of android.location to maximize battery life.' with a severity of 'L17' and a creation time of '21 hours ago'. The fifth issue is 'Use com.google.android.gms.location instead of android.location to maximize battery life.' with a severity of 'L8' and a creation time of '21 hours ago'.

The screenshot shows the 'Projects' page for a project named 'MyApp' in the SonarQube interface. The browser address bar shows the URL 'sonarqube.ecocode.io/projects'. The page header includes navigation tabs for 'Projects', 'Issues', 'Rules', 'Quality Profiles', and 'Quality Gates'. The 'Projects' tab is active, displaying a summary of the project's status. The main content area shows a 'Failed' status for the project, with a 'Last analysis: 5 minutes ago' timestamp. The project is associated with the 'Social' quality profile, which has a status of 'A' and a score of '0'. The project is also associated with the 'Environment' quality profile, which has a status of 'A' and a score of '6'. The project is associated with the 'Java, Xml' quality profile, which has a status of 'S' and a score of '1.5k'. The project is associated with the 'Overall Status' quality profile, which has a status of 'Failed' and a score of '0'.

\*self-hosted instance of the ecoCode SonarQube plugin

# Technical hurdles

Outdated








Ongoing







# Related works

## Academic

-  EcoAndroid [Ribeiro et al., 2021]
-  E-Debitum [Maia et al., 2020]
-  xAL [Fatimaa et al., 2020]
-  aDoctor [Iannone et al., 2020]
-  Green Android Lint [Le Goer, 2019]

## Non-Academic

-  Green Software Insights [CAST, 2023]
-  EcoSonar [Accenture, 2022]
-  Greensight Sonar [Capgemini, 2022]\*
-  ~~Ecoscan [Enedis, 2020]~~

\*joined the ecoCode project in 2024

# Digital commons

Avoid reinventing the wheel every time

Open Source improves IT sustainability. ecoCode cannot but be OSS

Build a community first (e.g., through hackathons). The lines of code will follow

Many to watch, few to make



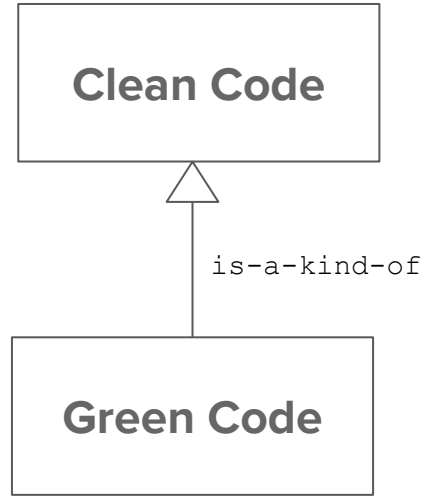
**Green Code Initiative (GCI)**

<https://github.com/green-code-initiative>

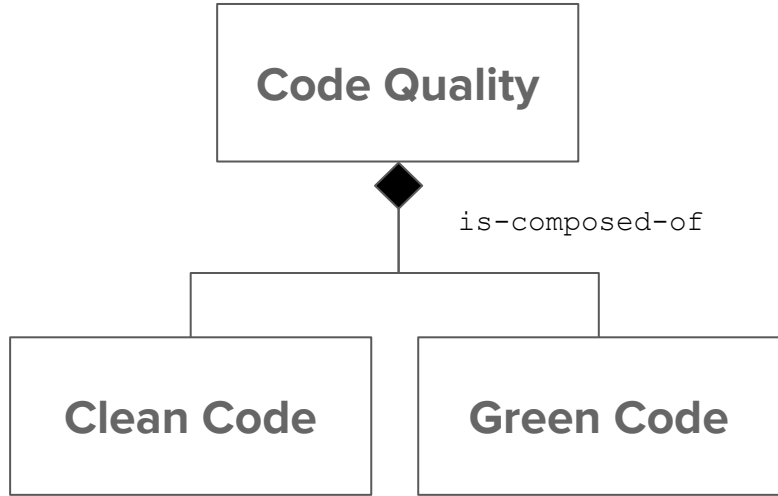


**Food for thought**

# How clean|green code relate?



*or*



# Green software supply chain

Greening the software is noble, but greening the software supply chain too

Motto: “Wherever there's code, eco-coding is possible !”

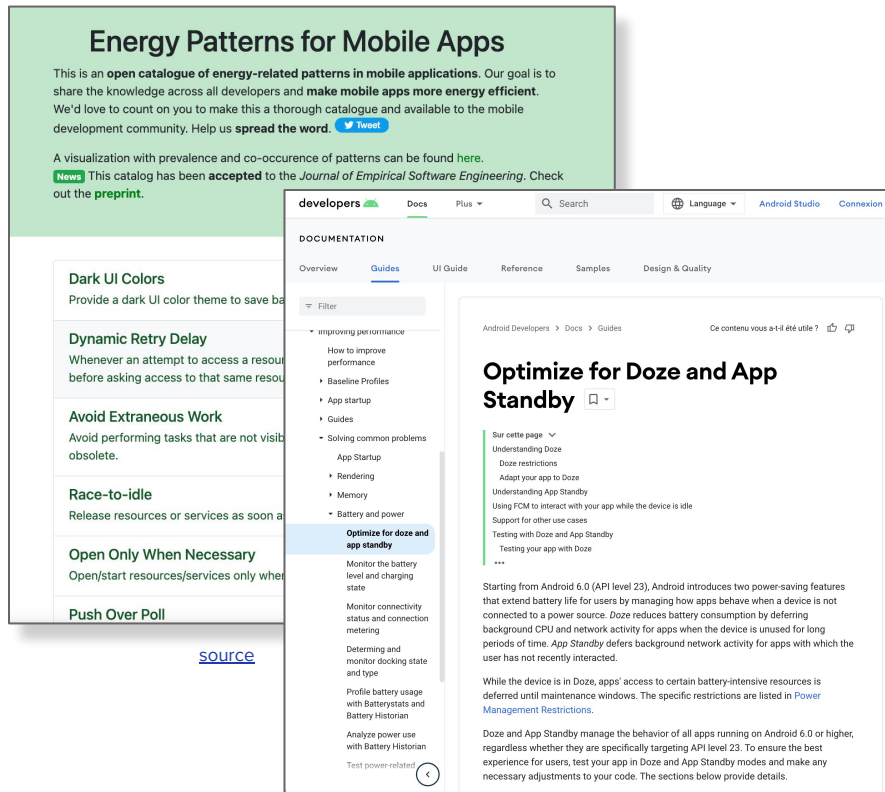
The “Everything-as-code” is a huge potential reservoir of green code smells :  
infrastructure-as-code, configuration-as-code, platform-as-code, ...

# Static analysis: The great filter

Very few general rules of thumb withstand the filter of static code analysis (“use cache”, “not too much videos”, etc.)

Pro Tips: Must be rooted at syntax-level

Bottom-up approach is the preferred way to find new rules/patterns/best-practices



# Static analysis: A noble art

## Challenges



Post-processing



Cross-scanning



False positive/negative

## Opportunities



Taint analysis



Call Graph/Control Flow Graph



Machine-Learning

# **Pain point: the evaluation. But...**

Unlike 90's OOP code smells, green code smells are still in their infancy

Do not expect green code to do what clean code has barely done

Sometimes common sense is enough

Ever-evolving mobile platforms makes things even more challenging

# Round-trip engineering



design-time



Joular ⚡

  
PowDroid

ETSdiff

run-time

# UX/UI

“Wow effect” is important to attract early-adopters

Our revamped UI was hard-coded. Tailoring the SonarQube UI to green-specific concepts would require diving deep

Developers can find green code burdensome. Gamification can help (to engage and reward)



**The end.**