Static code analysis for reducing energy code smells in different loop types: a case study in Java

Ram Prasad Gurung
Department of Software Engineering
LUT University
Lappeenranta, Finland
ram.prasad.gurung@student.lut.fi

Jari Porras
Department of Software Engineering
LUT University
Lappeenranta, Finland
jari.porras@lut.fi

Jarkko Koistinaho
AtoZ Oy
Tampere, Finland
jarkko.koistinaho@gmail.com

Abstract—**An increase in ICT devices and services has led to a rise in carbon emissions. As a result, there is a growing demand for energy-efficient software; however, this demand remains unmet due to the lack of knowledge regarding the best practices for reducing energy consumption in software. Unnecessary iterations and faulty looping conditions in different loops can consume high energy, and loops are considered as one of the most energy consuming entities. The purpose of this study is to detect and rectify energy code smells in different Java loop types by implementing static code analysis. Using the DSR approach, a Java Maven custom SonarQube plugin, *GreenForLoops*, was developed. The plugin underwent in-house testing as well as evaluation by professionals from industry. The professionals had provided feedback, which were later analyzed by using a qualitative method. For internal testing, 16 different open-source Java projects were selected. The results demonstrated considerable variations in the prevalence of energy code smells across the projects. Additionally, the plugin provided sample code suggestions to address each identified energy code smell. Finally, based on professional reviews, the plugin received an overall rating of Very Good. In conclusion, the plugin had successfully detected code smells and suggested code samples to rectify the detected code smells. However, it cannot be overlooked that the plugin may also generate false positives.**

Index Terms—**Energy code smell, SonarQube, Sustainability, Java, Energy-Efficient Software, Loop types.**

I. Introduction

Human activities have had a significant impact on the planet earth, influencing various aspects of environment, ecosystems, and climate. With the advancements made by humans in the field of technology, industry, and agriculture, there have been profound effects on the planet's health and balance. Due to this, there has been an increase in pollution, global carbon emission, and rise of temperature. Among all the sources of carbon emission, ICT is one of them. The contribution of ICT to Global Greenhouse Gas Emissions (GHGE) is projected to increase notably from approximately 1% in 2007 to 3-3.6% in 2020, with an expected annual growth of 5.6-6.9%, potentially surpassing 14% of the 2016-level worldwide GHGE by 2040 [1]. With the increase of our reliance on ICT devices and services, there has also been an increase in the need of energy, thereby multiplying the manufacture of energy via different sources to power these devices. These devices can be software-driven. Even though, software systems do not consume energy directly, they can have impact in hardware utilization, leading to indirect energy consumption [9]. Due to this, there is a need for energy efficient software. There are multiple ways to help in developing energy efficient software by estimating and calculating energy consumption of a software.

Software systems play a significant role in our daily lives, powering various applications and services. But the high energy consumption by these software has been a major concern [2]. As per the Wirth's law [3], *"Software is getting slower more rapidly than hardware becomes faster."* There are improvements in hardware in terms of performance, and the hardware resources–CPU, memory, storage–have become cheaper, but the results of refinements are neglected by software inefficiencies as software designers and developers have less concern on writing energy-efficient software to make the good use of those resources [4]. Due to this, there has been high energy consumption and carbon emission, which can have negative impact in environmental sustainability.

One of the popular ways to estimate energy consumption of software is static code analysis, which has less energy consumption estimation overhead than dynamic code analysis [5]. SonarQube[1] is one of the several tools available for static code analysis. This tool supports more than 30 programming languages including Java. In addition, it also helps in detecting code smells and assists in maintaining the quality of software code[2]. Fowler and Beck [6] introduced 'Code Smell' as a signal that points to potential deeper problems within a system. A 'Code Smell' refers to a section of code with a high likelihood of containing inherent errors or experiencing low performance, and it can be Duplicated Code, Long Method, Large Class, Long Parameter List, Temporary Field, Incomplete Library Class, Data Clump, and Message Chains [7]. In case of energy code smells, these are the patterns in software codes that might increase the energy consumption of a software. In SonarQube, all of these code smells are included as a different set of rules and referred as maintenance issues [8].

Java is one of the most widely used programming languages in various domains. While writing many Java applications, there can be multiple loops that execute repeatedly. Loops are considered as one of the most energy consuming entities

[1]https://www.sonarsource.com/products/sonarqube/
[2]https://docs.sonarsource.com/sonarqube/latest/

[5]. Inefficient coding practices like unnecessary iterations and flawed looping conditions can lead to high energy consumption and affect the overall energy efficiency of the software. The management of this situation hinges upon the proficiency of software developers.

The motivation behind exploring static code analysis techniques to detect and rectify the energy code smells within the different loop structures of Java is twofold. Firstly, it aims to help developers uncover energy-related issues and assess their impact on resource consumption, and apply appropriate optimizations or refactoring strategies. By this, it will lower energy consumption and carbon footprints. Secondly, optimizing energy efficiency in software has tangible benefits. Businesses can reduce their operational costs by installing energy-efficient applications [9]. Heavy resource usage resulting in fast battery drain is one of the reasons for poor app review in app stores [9]. So improved battery life on the devices can enhance user experiences. Due to this, addressing energy code smells in loop structures of Java can become crucial for both developers and end-users.

The rest of this paper is organized as follows. In Section II gives a background of this study. Further information regarding the methodology is available in Section III. The results of this research are described in Section IV, where results are divided as per the sub-research questions and feedback from the professionals from industry. In Section V, the results presented in this paper are discussed along with the feedback from professionals and threats to validity. Section VI reviews the related work. At last, Section VII concludes with the summary of this paper along with highlights to future work.

II. Background

A. Static Code Analysis

Louridas [12] defined static code analysis as the process of checking programs for errors without executing them and also compared the tendency of programs to attract defects similar to wool that attract dust and lint as it collects static electricity. The paper furthermore explained that static code analysis is employed after compilation and before testing. Static analysis as the analysis of program code to evaluate and reason about all potential behaviors that could occur during runtime, and the static analysis approach is meant to assess the source code by verifying its adherence to specific rules and examining the usage of arguments, among other aspects [13]. Additionally, static code analysis primarily revolves around examining the control flow and data flow of a program [14].

The static code analysis can be divided into two approaches: manual and automated [13]. In a manual approach, it is done by humans in the form of self-review, peer review, walk-through, and inspection or audit. In the case of automated, it is carried out by automated tools resulting in faster and efficient performance. The history and development of automated tools can be traced back to 5 decades earlier. *Lint*, a tool developed by Stephen Johnson at Bell Laboratories in the 1970s, has inspired many tools, both open source and commercial, for many programming languages and operating systems [12].

B. Understanding of Developers on Software Energy Consumption

This section describes how much the software developers and programmers are aware of energy consumption by software, and need of energy efficient software.

Pinto and Castor [9] highlighted that insufficient knowledge and inadequate tools are the two main energy related problems and provided an overview of current literature on how researchers in software engineering fields are addressing these issues. Pang et al. [10] also argued that the education, training, and knowledge do not mirror the demand for energy-efficient computing. This study conducted a survey, where 122 programmers took part, indicating that programmers lack knowledge of energy efficiency and the best practices to reduce software energy consumption, underscoring the importance of training on energy consumption. Software energy consumption research gaps exist not just in the industry but also within the field of academic research. Even though there are several empirical studies focused on investigating the cause of energy leaks in source code, there remains a little knowledge available in literature regarding the potential effects of *bad code smells* on energy consumption [11].

The two pieces of literature mentioned above highlight the lack of knowledge, tools, and training required to address high energy consumption. Due to this, some software developers do not know in which part of a software program is consuming more energy, and how codes can be refactored to minimize energy consumption. Changes in software programs can add efficiency on the changes made in hardware to reduce energy consumption. The recent studies showed the better results in energy saving outcomes after enabling and motivating software developers to take part in the energy consumption process [9].

C. SonarQube

SonarQube is a systematic code review tool that aids in the automated delivery of clean code or maintaining code quality. It supports 30+ different programming languages and can be integrated into Continuous Pipelines and DevOps platform[2]. The primary idea of SonarQube operates on the foundation of a source code analyzer component, which carries out fundamental analysis tasks like code line counting, alongside an application server that visually presents the analyzed data through a browser interface [16]. SonarQube is one of the most common open source automatic static code analysis tools used in both industry and academia [17]. To perform code analysis, SonarQube requires a set of defined rules.

When a project is analyzed in SonarQube, at first, it undergoes a parsing process, where the source code is transformed into an Abstract Syntax Tree (AST) [18] [19]. After the AST is constructed, SonarQube performs a traversal of the tree structure by the help of SonarSource Language Recognizer (SSLR) toolkit, which is provided for each language [19]. As SonarQube navigates this AST, it applies the rules to check various code issues, code standard violations, code smells, and security vulnerabilities in sample code. Furthermore, when source code is analyzed by SonarQube, it not only reports the

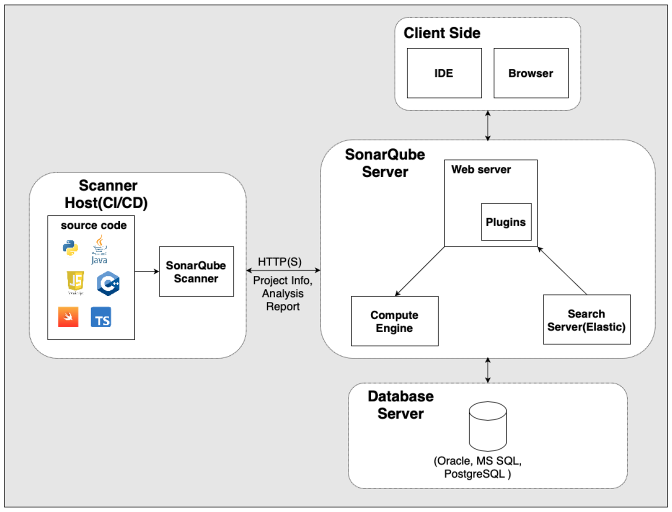


Fig. 1. SonarQube Architecture[3] [21]

code issues but also calculates the technical debt. SonarQube integrates Software Quality Assessment based on Life-cycle Expectations (SQALE) method to calculate the technical debt, which is shown in the SonarQube issues dashboard [20]. In addition, to carry out static code analysis, SonarQube is divided into multiple components.

The architecture of SonarQube is generally divided into 4 components: Database Server, Scanner Host, Plugin, and SonarQube Server[3] [21]. In addition, each programming language can have a different number of in-built rules in SonarQube. These rules help to detect 3 types of issues: Bug, Vulnerability, and Code Smell [23]. Furthermore, SonarQube categorizes rules into five different severity levels and assesses the severity based on their potential to negatively impact software quality. Those categories of severity levels are *BLOCKER, CRITICAL, MAJOR, MINOR, and INFO* [23].

III. Methodology

This section describes all the research methods and practices that are applied for this study. Firstly, the goal and research questions of this research are introduced, followed by explaining the research method followed for the research. After that, plugins rules for the SonarQube plugin and selection criteria for the Java open-source projects are defined.

A. Goal and Research Questions

In general, the goal of this research is to detect the energy code smells in different loop types and suggest the rectifying measures for those smells. The goal of the study is defined using the Goal-Question-Model technique [24], as shown in the Table I.

As per the goal, the following are the main research question and its sub-research questions along with its respective rationale:

TABLE I
Goal Definition

Analyze	Static code related to different loop types
Purpose	Detect and rectify code smells
With respect to	Energy consumption
View point	Developers
In the context of	Different Java projects
Goal Statement	
Analysis of static codes to detect and rectify energy code smells in different loop types from the view-point of developers in different Java projects.	

1) RQ 1: How can static code analysis contribute to the detection and rectification of energy code smells in different loop structures in Java?: The main rationale of this research question is to investigate how static code analysis can be leveraged in detecting and rectifying energy-related code issues in different loop types in Java.

The chosen method for this research involves developing the SonarQube Java Maven plugin,*GreenForLoops*[4], to assess its effectiveness in addressing the main question, with sub-research questions formulated accordingly.

Sub RQ 1.1: To what extent does the GreenForLoops plugin demonstrate effectiveness across a wide range of real world Java projects? As the plugin will be tested in different open source Java projects, this sub research question helps to find how this plugin performs in different Java projects of different size chosen from various owners.

Sub RQ 1.2: How does GreenForLoops plugin help to detect energy code smells in different loop types in Java, incorporating factors such as the category of code smells and their precise locations within the codebase? For this sub-research question, this shows how the SonarQube detects energy code smells in different loop structures in Java and showcases them as per the type of code smells, either major or minor, along with other metrics like fixing time, and the actual line of code where the energy code smell is detected.

Sub RQ 1.3: How does GreenForLoops plugin assist in rectifying energy related code smells related to different loop types in Java? The rationale behind this sub-research question is to explore how the SonarQube dashboard presents the rectifying code samples for the identified energy code smells based on the capabilities of the plugin.

B. Research Methods

This section explains the research method, Design Science Research (DSR), adopted for this study design solution. In order to design and construct the assessment of the artifact, qualitative methodologies are employed.

1) Design Science Research: Design science involves the creation and analysis of problem-solving artifacts that interact with their specific context to bring about improvement [25]. It focuses on creating practical solutions that can improve or solve real-world problems. Fundamentally, DSR is a problem-solving approach that aims to effectively accomplish the analysis, design, implementation, and utilization of information

[3]https://www.youtube.com/watch?v=VzuY0jyrF80

[4]https://github.com/ramgrg/greenForLoops.git

systems by generating ideas, practices, technical abilities, and products [26]. This study addresses the problem of high energy consumption associated with energy code smells in different loop types of Java programming language.

2) Qualitative Research: Qualitative process helps to get information about the feelings, opinions, and understanding of participants. This is an approach to study the social world that helps to describe and analyze the behaviour and culture of humans or groups from the viewpoint of those being studied [27]. In a way, it helps to understand the experience of people. It also helps in comprehending a requisition question from a humanistic or idealistic perspective [28]. In this research, the experts from industry tested the plugin and their experiences on working with the plugin were recorded. Through these experiences and feedback collected via Google Form, the efficiency and applicability of the plugin in a wide range of Java projects were defined. In addition, 1 to 5 grading system [29] available in Google form was also considered for providing extra assessment criteria. In this grading system, 1 is considered as *Unsatisfactory*, 2 as *Satisfactory*, 3 as *Good*, 4 as *Very Good*, and 5 as *excellent* [29].

C. RULES DEFINITION

To perform static code analysis by SonarQube, a set of rules should be defined. As per those rules, SonarQube analyses the source code, a very similar way antivirus software detect viruses [30]. In addition, if the source code violates these rules, SonarQube labels those codes with issue [8]. By considering the requirements of SonarQube, the following rules were finalized for the plugin:

- **Do not initiate an array inside a loop.**
  The process of initializing an array using a loop statement results in high energy consumption, mainly due to the repetitive execution of comparison operations and index modifications [7]. In the plugin, this rule is classified as *Minor*.
- **Do not concatenate string inside loops. If you have to, use StringBuilder instead** [18].
  String in Java is immutable; concatenation of string creates new string, whereas StringBuilder is mutable and provides a method to modify a string without creating a new string, thereby consuming less energy [31]. This rule is similar to one of a rule[5] defined by SonarQube. In the plugin, this rule is labelled as *Minor*.
- **Do not access the collection size attribute and array length in the body of a loop** [18].
  Initializing a loop without referring to array length consumes 10% less energy than a loop initialized by mentioning array length [32]. This rule is classified as *Minor*.
- **Do not access global variables inside for-loop; instead try to use local variables**.
  The loops keep on iterating until the escape condition is fulfilled. Inside the loop, if it is accessing some global variables, a problem may arise as global memory will be repeatedly accessed in each iteration [7]. In the plugin, this rule is classified as *Minor*.
- **If possible try to avoid nested loops to avoid time complexity**.
  In the nested loop, even though a simple operation is running, the number of executions of the operation multiplies thereby affecting in energy saving as well as runtime performance [7]. Therefore, refactoring of nested loops into a single loop is suggested. This rule is categorized as *Major*.

The plugin was developed by implementing these rules. To develop plugin, the guidelines were followed as mentioned in the Sonar-Java[6] and tutorial[7] provided by SonarQube for custom plugin development. While developing the plugin, unit testing and integration testing were performed so that bugs and defects can be identified at an early stage.

D. Java Projects Selection

The performance of the plugin was evaluated as per its applicability in a wide range of Java projects and detection of energy code smells. The criteria for selecting Java projects are defined in Table II.

TABLE II
SELECTION CRITERIA FOR JAVA PROJECTS

Inclusive Criteria	Rationale
The project should be open-source.	The source code and others information about the projects are easily accessible.
Total lines of code should be equal or more than 4500.	The code's lower bound is established for easy comparison between projects near this limit and those surpassing it.
The project should be either trending on GitHub or popular in developer communities.	This criterion is introduced so that the test results can have impact on developer communities.

E. Research Reproducibility

This section expounds the process to replicate this study. The *GreenForLoops* plugin is available in GitHub[4]. The replication package contains (i) the information about the plugin (ii) how to configure the plugin in a system (iii) details about the selected Java projects.

IV. RESULTS

In DSR, artifacts are evaluated for functionality, completeness, consistency, accuracy, performance, reliability, usability, and organizational fit [26]. This section presents research results, findings, and solutions addressing the study's problem. As an artifact, a custom Java Maven SonarQube plugin, *GreenForLoops*, was developed to detect and rectify energy code smells in various Java loop types. The research has only one main research question: **"How can static code analysis contribute to the detection and rectification of energy code smells in different loop structures in Java?".** This

[5]https://rules.sonarsource.com/java/RSPEC-1643

[6]https://github.com/SonarSource/sonar-java/blob/master/docs/CUSTOM_RULES_101.md/
[7]https://docs.sonarqube.org/latest/analyzing-source-code/languages/java/

main research question is answered by the three sub-research questions. The first sub-research questions explains how the SonarQube custom plugin performs while testing in the wide range of real world Java projects. Secondly, it inquires into how the plugin aids in detecting the energy code smells within various Java loop structures. Lastly, the third sub-research question examines how *GreenForLoops* plugin contributes in rectifying the detected energy code smells in different Java loop types.

A. Sub RQ 1: To what extent does the GreenForLoops plugin demonstrate effectiveness across a wide range of real world Java projects?

The objective of this sub-research question is to determine the effectiveness of applying the *GreenForLoops* plugin on various Java projects from different vendors for detecting energy code smells. As shown in the Table III, there are 16 selected projects from 10 different owners. All the Java projects were selected as per the criteria mentioned in Table II. Among the project owners, *Google* leads the way with 4 selected projects, closely followed by *Alibaba* with 3, and *Apache* with 2 selected projects. The remaining owners–*GeyserMC, Karate Labs, The Algorithms, Kekingcn, Jagrosh, MyBatis*, and *Spring*–each of them has 1 selected project showcasing a diverse range of projects from different owners.

TABLE III
TOTAL NUMBER OF PROJECTS PER OWNER.

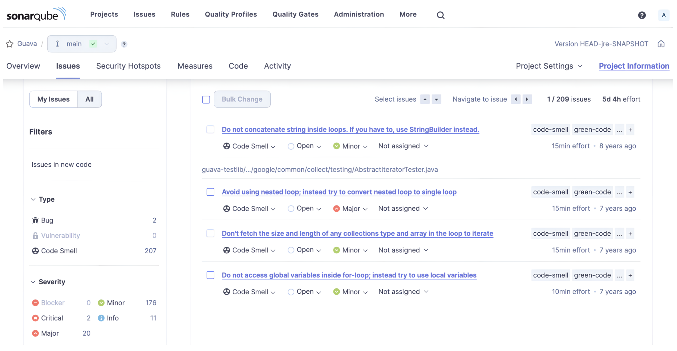
Owner	**Number of projects**
Alibaba	3
Apache	2
Google	4
GeyserMC	1
Karate Labs	1
The Algorithms	1
Kekingcn	1
Jagrosh	1
MyBatis	1
Spring	1
Total:	**16**

Table IV shows the list of all selected projects, its total lines of codes and GitHub URLs along with the total number of code smells detected during SonarQube analysis. Among the 16 selected projects, *JMusicBot, COLA,* and *Commons Validator* had the least number of lines of code with 4,785, 6,488, and 7,892 respectively, whereas *Guava* and *Error Prone* had the largest codebase with 124,119 and 101,557 lines respectively.

Furthermore, the feedback from the professionals described in Section IV-D also sheds light on the applicability of the plugin.

B. Sub RQ 2: How does GreenForLoops plugin help to detect energy code smells in different loop types in Java, incorporating factors such as the category of code smells and their precise locations within the codebase?

Table IV illustrates the overall number of energy-related code smells identified in each project, following the elimination of false positives, and subsequently categorizing them into major and minor energy code smells. In the *The Algorithms Java* project, which contains 27,121 lines, the maximum number of energy code smells - 547 in total, consisting of 184 major and 363 minor energy code smells - was detected. In contrast, the *COLA* project, with 6,488 lines of code, exhibited the detection of only 2 energy code smells: 1 major and 1 minor. In addition, *Google Guava*, with the largest codebase among the selected projects, had 160 energy code smells (16 major and 144 minor code smells). On the other hand, *JMusicBot*, with the least number of lines of code, had 16 energy code smells, all of which were minor energy code smells. Furthermore, Table IV also shows false positives detected in every selected projects. Among the projects, *Google Guava* had the highest number of false positives, with 36, followed by *MyBatis-3* with 21 false positives. On the other hand, *COLA* and *Spring Data REST* displayed the fewest false positives, each having only 1.



1 / 209 issues 5d 4h effort

Filters

Issues in new code

Type

Bug	2
Vulnerability	0
Code Smell	207

Severity

Blocker	0	Minor	176
Critical	2	Info	11
Major	20		

Do not concatenate string inside loops. If you have to, use StringBuilder instead. — code-smell green-code ... +
Code Smell · Open · Minor · Not assigned — 15min effort · 8 years ago

guava-testlib/.../google/common/collect/testing/AbstractIteratorTester.java

Avoid using nested loop; instead try to convert nested loop to single loop — code-smell green-code ... +
Code Smell · Open · Major · Not assigned — 15min effort · 7 years ago

Don't fetch the size and length of any collections type and array in the loop to iterate — code-smell green-code ... +
Code Smell · Open · Minor · Not assigned — 15min effort · 7 years ago

Do not access global variables inside for-loop; instead try to use local variables — code-smell green-code ... +
Code Smell · Open · Minor · Not assigned — 10min effort · 7 years ago

Fig. 2. A list of major and minor energy code smells

In SonarQube, the portal shows the total code smells detected in a particular project along with the list of energy code smells, which are separated as per the severity level of code smells. Fig. 2 shows the list of some of the energy code smells detected within the chosen projects, and the energy code smells are further divided as per the severity levels.

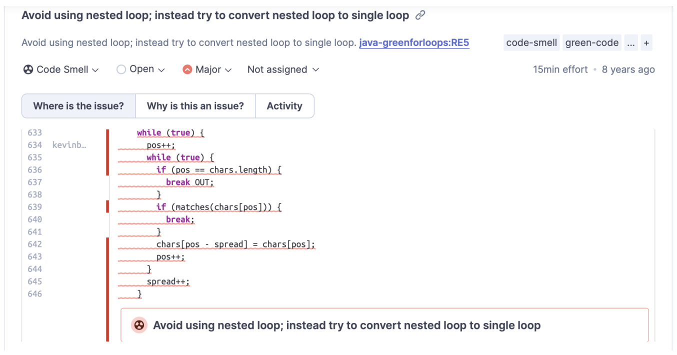
For every issue, SonarQube also locates where the issue is. If one of the issues as shown in Fig. 2 is clicked, it will open a window as shown in Fig. 3 or 4, and the source codes with the issue will be highlighted in red followed by the cause of the issue.

C. Sub RQ 3: How does GreenForLoops plugin assist in rectifying energy related code smells related to different loop types in Java?

SonarQube detects energy code smells and provides corresponding rectifying approaches. By following these approaches, the source code can be refactored and optimized to address the identified energy code smells. Fig. 5 and 6 show how to fix energy code smells related to nested loop and string concatenation respectively. While displaying how to correct code smells, SonarQube presents non-compliant and compliant solution code samples, which are defined in the

TABLE IV
LIST OF THE SELECTED PROJECTS AND FINDINGS.

Project Name	Total lines of code	Total code smells	Major code smells	Minor code smells	False Positives	Technical debts	GitHub URL
Bytekit	9,241	37	4	33	2	1 day	https://github.com/alibaba/bytekit.git
COLA	6,488	2	1	1	1	35 mins	https://github.com/alibaba/COLA.git
Commons Configuration	20,952	21	1	20	7	6 hrs 45 mins	https://github.com/apache/commons-configuration.git
Commons Validator	7,892	15	0	15	5	5 hrs	https://github.com/apache/commons-validator.git
EasyExcel	16,431	17	0	17	4	4 hrs 40 mins	https://github.com/alibaba/easyexcel.git
Error Prone	101,557	122	4	118	6	3 days 5 hrs	https://github.com/google/error-prone.git
Geyser	52,343	126	8	118	16	3 days 6 hrs	https://github.com/GeyserMC/Geyser.git
Google-java-format	10,457	23	4	19	3	6 hrs 10 mins	https://github.com/google/google-java-format.git
Guava	124,119	160	16	144	36	5 days 4 hrs	https://github.com/google/guava.git
Gooogle Guice	28,513	35	5	30	2	1 day	https://github.com/google/guice.git
Karate	38,393	41	3	38	3	1 day 3 hrs 10mins	https://github.com/karatelabs/karate.git
The Algorithms-Java	27,121	547	184	363	19	17 days	https://github.com/TheAlgorithms/Java.git
kkFileView	9,534	42	3	39	7	1 day 1 hr	https://github.com/kekingcn/kkFileView.git
JMusicBot	4,785	16	0	16	5	5 hrs 15 mins	https://github.com/jagrosh/MusicBot.git
MyBatis-3	22,410	50	6	44	21	2 days	https://github.com/mybatis/mybatis-3.git
Spring Data REST	13,250	3	0	3	1	1 hr	https://github.com/spring-projects/spring-data-rest.git


Avoid using nested loop; instead try to convert nested loop to single loop

Avoid using nested loop; instead try to convert nested loop to single loop. java-greenforloops:RE5 code-smell green-code ... +

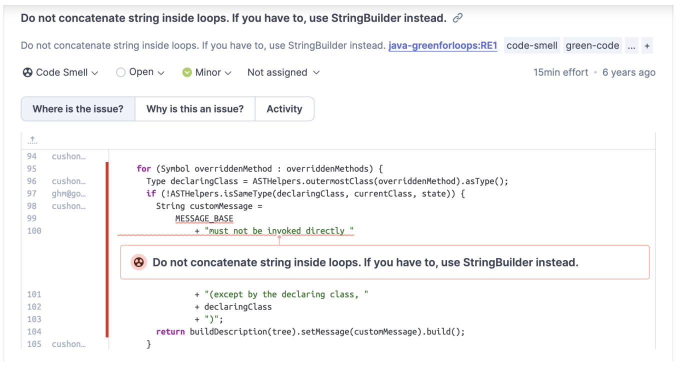
Code Smell · Open · Major · Not assigned · 15min effort • 8 years ago

Where is the issue? | Why is this an issue? | Activity

```
kevinb…
    while (true) {
      pos++;
      while (true) {
        if (pos == chars.length) {
          break OUT;
        }
        if (matches(chars[pos])) {
          break;
        }
        chars[pos - spread] = chars[pos];
        pos++;
      }
      spread++;
    }
```

Avoid using nested loop; instead try to convert nested loop to single loop

Fig. 3. SonarQube locating major energy code smell in source code


Do not concatenate string inside loops. If you have to, use StringBuilder instead.

Do not concatenate string inside loops. If you have to, use StringBuilder instead. java-greenforloops:RE1 code-smell green-code ... +

Code Smell · Open · Minor · Not assigned · 15min effort • 6 years ago

Where is the issue? | Why is this an issue? | Activity

```
cushon…
      for (Symbol overriddenMethod : overriddenMethods) {
cushon…  Type declaringClass = ASTHelpers.outermostClass(overriddenMethod).asType();
ghm@go…  if (!ASTHelpers.isSameType(declaringClass, currentClass, state)) {
cushon…    String customMessage =
               MESSAGE_BASE
                   + "must not be invoked directly "
```

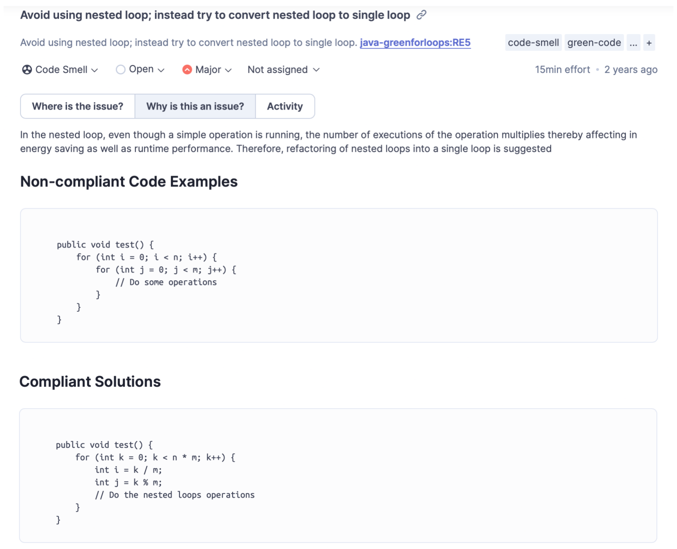
Do not concatenate string inside loops. If you have to, use StringBuilder instead.

```
                   + "(except by the declaring class, "
                   + declaringClass
                   + ")";
           return buildDescription(tree).setMessage(customMessage).build();
cushon…  }
```

Fig. 4. SonarQube locating minor energy code smell in source code

GreenForLoops plugin. The compliant solution example acts as a guideline to rectify energy code smells.

Furthermore, SonarQube also estimates the time it may take to fix each individual energy code smell. The cumulative sum


Avoid using nested loop; instead try to convert nested loop to single loop

Avoid using nested loop; instead try to convert nested loop to single loop. java-greenforloops:RE5 code-smell green-code ... +

Code Smell · Open · Major · Not assigned · 15min effort • 2 years ago

Where is the issue? | Why is this an issue? | Activity

In the nested loop, even though a simple operation is running, the number of executions of the operation multiplies thereby affecting in energy saving as well as runtime performance. Therefore, refactoring of nested loops into a single loop is suggested

Non-compliant Code Examples

```
public void test() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            // Do some operations
        }
    }
}
```

Compliant Solutions

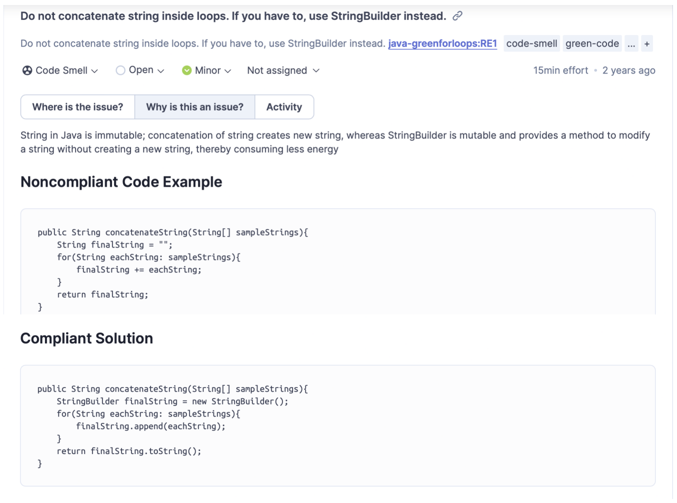
```
public void test() {
    for (int k = 0; k < n * m; k++) {
        int i = k / m;
        int j = k % m;
        // Do the nested loops operations
    }
}
```

Fig. 5. Rectifying measure for nested loop energy code smell

of all the time to fix each individual energy code smell gives the technical debt of the project. Table IV shows technical debt of each selected project. Among all projects, *Guava* had the maximum technical debt of 5 days and 4 hours, whereas *COLA* had the minimum technical debt, requiring only 35 minutes to amend the energy code smells.

D. Feedback from Professionals

6 professionals tested the plugin in Java projects and provided their feedback. The participants were coded as P1, P2,


Do not concatenate string inside loops. If you have to, use StringBuilder instead.

Do not concatenate string inside loops. If you have to, use StringBuilder instead. java-greenforloops:RE1 code-smell green-code ... +

Code Smell · Open · Minor · Not assigned · 15min effort • 2 years ago

String in Java is immutable; concatenation of string creates new string, whereas StringBuilder is mutable and provides a method to modify a string without creating a new string, thereby consuming less energy

Noncompliant Code Example

```
public String concatenateString(String[] sampleStrings){
    String finalString = "";
    for(String eachString: sampleStrings){
        finalString += eachString;
    }
    return finalString;
}
```

Compliant Solution

```
public String concatenateString(String[] sampleStrings){
    StringBuilder finalString = new StringBuilder();
    for(String eachString: sampleStrings){
        finalString.append(eachString);
    }
    return finalString.toString();
}
```

Fig. 6. Rectifying measure for string concatenation energy code smell

P3, P4, P5, and P6. P4 works in *SonarSource*[8], the company that developed SonarQube. Among the participants, three were Software Developers, while the others held positions as Principal Software Engineer, Software Engineer, and Lead Software Architect. P4 had the most experience with 35 years, followed by P1 with 15 years and P5 with 11 years. Both P2 and P3 had 5 years of experience, while P6 had the least experience with 4 years.

Out of 6 participants, only 50% of of the participants take energy consumption into account while programming, with the other half not considering it. In case of performance rating of plugin, half of the professionals had given *Excellent* rating, whereas two professionals had rated the plugin as *Very Good*, and the remaining one professional rated it as *Satisfactory*.

Furthermore, all of the participants had provided comments after the testings. Overall, P1, P2, P3, P5, and P6 had given *Excellent* rating. In the case of P3, the participant had some configuration issue but seemed satisfied with the easy installation features. Additionally, P5 and P6 suggested the inclusion of more rules in the plugin. Some of the reviews are mentioned below: *"Overall helpful suggestion from the plugin. Great work!"*

"Seems quite promising, however, I encountered some build issues during the configuration process. Overall, the tool shows great potential, particularly with its containerization feature, which enhances both accessibility and configurability."

"The plugin is working and fulfills the original needs. It's great to see the rationale of each developed rule. You also could add more rules."

As per P4, the results from static code analysis can be contradictory to some of the real-case scenarios. Furthermore, the participant emphasized that rules targeting architectural decisions have a significantly greater impact compared to rules focusing on low-level optimization: *"Not allowing to initiate an array inside a loop seems to be a good rule to force developers to reuse the same array in a loop.I think energy consumption, like performance, needs to be precisely measured to ensure a code pattern is more efficient than another. The results are too often counterintuitive, so good practice and rules should always be related to real-life studies. There are only some corner cases where it's true that string concatenation using '+' in Java is slow. For example: 'throw new IOException(Unexpected document root '" + elementName + "' instead of 'BugCollection'.");' is faster using the '+' operator than building a StringBuilder object. String concatenation using '+=' was so slow and so widely used that recent versions of the JVM make it faster than StringBuilder. So we should not rely on old measures but, again and again, collect evidence that all new JVMs still have this not-yet-optimized problem. One of the rules complains about 'while (result.is(Tree.Kind.PARENTHESIZED_EXPRESSION)) { ' but moving the enum element into a local variable makes the code harder to read. The rule related to time complexity in nested loops should provide some evidence that the complexity introduced by the single loop has a real energy consumption benefit. It can also raises a lot of false positives for loops that cannot be merged. In my view, energy consumption is well correlated to mono-thread performances, and it's easier to measure performance. Rules targeting architectural decisions are far more impactful than rules focusing on a low level of optimization."*

[8]https://www.sonarsource.com

V. Discussion

This chapter provides a thorough analysis of test results from diverse Java projects and industry professional's feedback. It begins by explaining how these findings address the research question and sub-questions, followed by a detailed analysis of industry specialist's feedback. The chapter concludes with an assessment of potential study validity threats.

A. Addressing the Research Question and Sub-research Questions

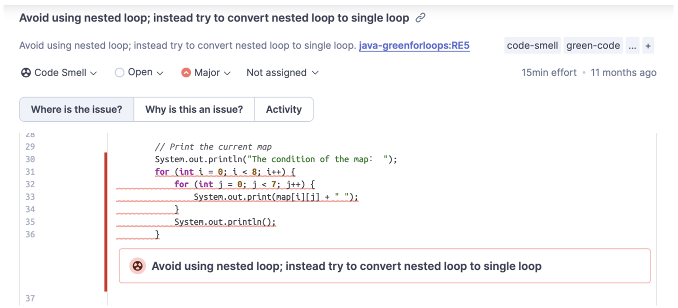
In this section, it addresses the research questions and sub-research questions as per the findings from the test results of different Java projects.

1) RQ 1: How can static code analysis contribute to the detection and rectification of energy code smells in different loop structures in Java?: In this research, among the available static code analysis tools, SonarQube was selected. For detecting and rectifying code smells in different loop structures in Java, a SonarQube custom Java Maven plugin, *GreenForLoops*, was created. To address this research question, the plugin was tested in 16 diverse Java projects, accompanied by a comprehensive presentation of the total energy code smells – categorized as major and minor – detected within each project, their corresponding locations in the source code, and a presentation of compliant source code sample to assist developers in effectively correcting the identified energy code smells.

Sub RQ 1.1: To what extent does the *GreenForLoops* plugin demonstrate effectiveness across a wide range of real world Java projects?
GreenForLoops plugin was tested in 16 different Java projects, and these projects were from 10 different owners. In addition, these Java projects have a total number of source code ranging from 4,785 to 124,119. This concludes that the selection of projects not only varies among different owners but also exhibits significant diversity in terms of overall size and complexity. Irrespective of the project size and vendor, the plugin detected energy code smells in the selected Java projects.

Sub RQ 1.2: How does *GreenForLoops* plugin help to detect energy code smells in different loop types in Java, incorporating factors such as the category of code smells and their precise locations within the codebase?
As shown in Table IV, the plugin had detected energy code smells in the selected projects. In addition to listing the energy code smells as shown in Fig. 2, every code smell was also located and highlighted in red in its corresponding source file. This helps developers identify the location of energy code smells and correct them according to the suggested code samples. Additionally, SonarQube categorizes the identified energy code smells based on their severity levels, enabling developers to prioritize and effectively address these issues.

While testing the plugin in the selected Java projects, false positives were detected. Those source codes were classified as false positives because they consisted of JUnit test codes, nested loops with a small number of iterations, and code that runs during debug mode. None of these code segments will affect the application in a production environment. It is already known that static code analysis tools can encounter issues with generating false positives [8]. SonarQube analyzes source code based on patterns and rules, which can sometime misinterpret complex logic, and it inspects code without a deep understanding of the overall program context, leading to false positives. The code sample shown in Fig. 8 was categorized as true positive as Sonarqube does not know the size of the array, whereas the code snippet exhibited in Fig. 7 was labelled as false positive since it involves only 56 iterations, resulting in a relatively small time complexity.


Avoid using nested loop; instead try to convert nested loop to single loop

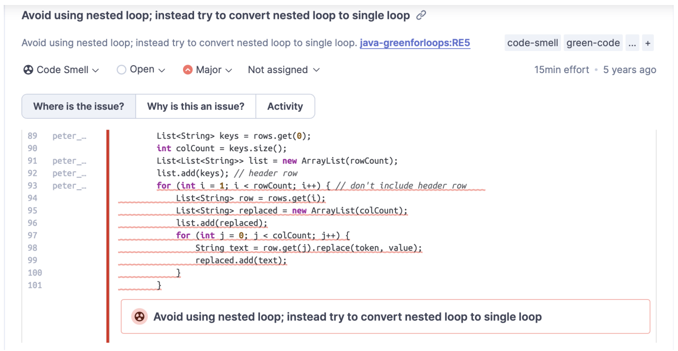
Avoid using nested loop; instead try to convert nested loop to single loop. java-greenforloops:RE5

```java
        // Print the current map
        System.out.println("The condition of the map:  ");
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 7; j++) {
                System.out.print(map[i][j] + " ");
            }
            System.out.println();
        }
```

Avoid using nested loop; instead try to convert nested loop to single loop

Fig. 7. Energy code smell categorized as false positive

Sub RQ 1.3: How does *GreenForLoops* plugin assist in rectifying energy related code smells related to different loop types in Java?


Avoid using nested loop; instead try to convert nested loop to single loop

Avoid using nested loop; instead try to convert nested loop to single loop. java-greenforloops:RE5

```java
        List<String> keys = rows.get(0);
        int colCount = keys.size();
        List<List<String>> list = new ArrayList(rowCount);
        list.add(keys); // header row
        for (int i = 1; i < rowCount; i++) { // don't include header row
            List<String> row = rows.get(i);
            List<String> replaced = new ArrayList(colCount);
            list.add(replaced);
            for (int j = 0; j < colCount; j++) {
                String text = row.get(j).replace(token, value);
                replaced.add(text);
            }
        }
```

Avoid using nested loop; instead try to convert nested loop to single loop

Fig. 8. Energy code smell categorized as true positive

With the help of the plugin, SonarQube had suggested rectifying code samples for each detected energy code smells. These suggested compliant solution samples can be the guidelines for developers to fix the energy code smells. The plugin does not automatically rectify the code smells like in *EcoAndroid* [34] and *SORALD* [35]. In addition, the plugin also forecasted the time required to address each individual energy code smell, as well as calculating the total technical debt of the entire project. Even though the time taken to rectify the issue was configurable and approximated according to the provided code sample, it is important to acknowledge that the actual time needed to fix a specific energy code smell may vary among developers depending on their experience and work efficiency. Nevertheless, SonarQube assists to provide an overall ballpark estimation to resolve the discovered energy code smells in a Java project. This feature empowers development teams with valuable insights into the potential effort and resources needed to enhance the overall energy efficiency and code quality of their software.

Furthermore, the plugin also provides an explanation for each identified energy code smell, clarifying why a particular line of code is flagged as such, which was not done in other plugins like *ecoCode* [18], *EcoAndroid*, and *SORALD*. For instance, in the Fig. 6, the reason behind "String concatenation" energy code smell is elucidated as: *"String in Java is immutable; concatenation of string creates new string, whereas StringBuilder is mutable and provides a method to modify a string without creating a new string, thereby consuming less energy"*. By providing the reason, the plugin serves to educate the developer thereby improving the programming skills of developers.

B. Feedback from Professionals

6 professionals took part in testing the plugin on Java projects. In qualitative analysis, there are no specific rules on fixing the sample size until the collected data is enough to answer the research question [36], but 5 to 50 participants are considered adequate [37]. From the feedback collection, the overall grading of 4.16 has been received. As per the grading system mentioned in Section III-B2, *Very Good* is the overall grade after analysis. Furthermore, in the case of feedback received from the participants, the majority expressed

satisfaction with the performance and effectiveness shown by the plugin. Both P5 and P6 suggested adding more rules in the plugins, which is one of the future plans.

However, P4 stated that the rules should always be backed by real-life studies. The participant further mentioned that sometimes the results of analysis can be contradictory and can introduce false positives. To explain this point, the participant provided some examples: string concatenation from ”+” in Java can be faster than building a *StringBuilder* object in some cases, and there can be some nested-loops that cannot be merged into one. In addition, the participant also stated that architectural decision-related rules have a greater impact compared to rules that emphasize low-level optimization.

All the rules implemented in *GreenForLoops* plugin are backed by scientific studies. All the rules along with its references are shown in Section III-C. SonarQube uses static code analysis to identify potential code issues. However, it may not have access to complete contextual information and intent of the code, leading to potential false positives. Developers may attribute the issue-list reported by SonarQube as false positive due to issues in checking mechanism, rule semantics (e.g., software engineering principles that may not apply in certain contexts), or conditions defining rule violations (e.g., specific complexity thresholds not universally applicable) [38]. Nevertheless, the plugin has rooms for improvement, and implementation of the rules in the plugin can be optimized for minimizing false positives.

C. Threats to Validity

To ensure the robustness and quality of this research, 3 different types of threats to validity are analyzed.

1) External Validity: Biasness in sample selecting the sample can be the potential threat to external validity. In our case, the Java projects were selected as per the criteria mentioned in Table II. 16 projects were selected from 10 different owners, and the size of the project ranges from 4,785 to 124,119. Random selection of different owners and its projects plus addressing the projects in terms of overall size and complexity allow not to introduce biases on selecting real-world Java projects.

However, it may not accurately reflect energy code smells patterns in projects developed in other programming languages or executed in different environments. Replicating the study in different programming languages and execution environments can help to mitigate this potential threat to validity.

2) Internal Validity: The quality and effectiveness of the plugin can pose a potential threat to internal validity. All the rules selected for detecting energy code smells are backed by the scientific studies, and references for each rule is given. However, there can be multiple rules other than 5 defined rules that can assist in detecting and rectifying energy code smells. The limited number of defined rules in the study can have impact in internal validity. However, adding more rules for future studies can minimize this threat to validity.

3) Construct Validity: Construct validity refers to the extent to which the SonarQube plugin accurately measures and identifies energy code smells in different loop types. From the professional feedback, the average rating of the plugin is 4.16. In the 1 (Unsatisfactory) to 5 (Excellent) grading scale, the average rating of 4.16 is pretty decent. Nonetheless, it is not possible to deny the chances of getting false positives. In every selected Java projects, false positives were detected. The participant, P4, also pinpointed the presence of some false positives in the feedback, thereby suggesting the rooms for improvement in the plugin

In addition, the study relies on static code analysis and lacks real execution data, potentially overlooking dynamic runtime optimizations that could have impact in energy consumption.

VI. Related Work

A. Analysis of energy consumption by static code analysis

Energy bugs in applications can cause high battery drainage in mobile phones resulting in serious issues with its durability and performance. Jiang et al. [39] proposed a static analysis technique called Static Application Analysis Detector (SAAD) to detect two types of energy bugs, resource leak and layout defect, in Android applications. In the research, it is mentioned that SAAD detects the energy bugs by considering components call relationship analysis, inter-procedural and interprocedural analysis, whereas for layout defects, SAAD uses Lint which performs some app analysis. This study performed experiments on 64 applications from Google Play Store, in which they found out that SAAD was able to detect resource leak and layout energy defect by 87.5% and 78.1% accuracies.

Pereira et al. [40] presented a tool called *jStanley*[9], a static analysis Eclipse plugin, which statically detects energy-inefficient Java collections and suggests more efficient Java collection alternatives. For this tool, the data about energy consumption and execution time for Java collections were provided through CSV files. As a result, the paper had presented improvement in energy gains between 2% and 17%, and reduction in execution time between 2% and 13%.

B. Plugins related to static code analysis for energy consumption

In case of energy consumption, there are multiple plugins created for static code analysis for tools like SonarQube, AndroidStudio[10].

Ribeiro et al. [34] created an AndroidStudio plugin, *EcoAndroid*, to help developers in building energy-efficient Android mobile applications. The plugin automatically applies energy patterns to the source codes written in Java. These energy patterns are typically illustrated by an anti-pattern demonstration and a set of instructions on how to modify the code to reduce energy consumption. In addition, the study has also indicated that apart from factors like 3G, WIFI, heavy graphic processing, and screen usage of an application, software codes–which are usually ignored by developers– can be also a reason for high energy consumption. Even though

[9]https://greensoftwarelab.github.io/jStanley/

[10]https://developer.android.com/studio

EcoAndroid is built for different tools than the plugin built for this study, the gist of the both plugin is the same: reduction of energy consumption in coding.

1) SonarQube Plugins for energy consumption: There are multiple papers about plugins related to SonarQube, which can be guidelines while developing the plugin for this research. Those papers have described the SonarQube architecture, plugin integrations, and some of their GitHub repositories contain guidelines on how to create a SonarQube plugin from scratch.

Someoliayi et al. [35] introduced a new system, *SORALD*, that uses metaprogramming templates to modify program abstract syntax trees and recommend solutions for static analysis alerts. The research claimed that the system had fixed 10 issues from SonarQube. To enhance the convenience of integrating SORALD into the workflow of developers, the article introduced *SORALDBOT* – a bot that monitors changes on GitHub repositories continuously, with the aim of detecting new violations that arise from new commits, fixes those violations, and propose the fixed version to the developers as the patched version, a pull request to the relevant repository. This plugin is very different to the plugin that is created for this study on the basis of objective. But this paper can provide insights on how different components are integrated in the SonarQube and also the structure of a SonarQube plugin.

Goaër and Hertout [18] introduced a SonarQube plugin called *ecoCode* that detects energy code smells of any native Android application written in Java. The research also mentioned that energy smells can be organized into 8 categories: optimized API, leakage, bottleneck, sobriety, idleness, power, batch, and release. In their experiment, they examined three types of files simultaneously. Those file types were Java files, XML files such as manifest and certain type of resources, and Gradle files that include information related to build, settings and properties. The *ecoCode* plugin is taken as the main reference for the plugin of this study. Two rules from the *ecoCode* plugin are adopted as rules for this research. The plugin created for this research will be for all general Java projects, but the *ecoCode* plugin is only limited to native Android projects written in Java even though is not just focused for only loop types of Java.

Maia et al. [42] presented a SonarQube plugin called *E-Debitum* to measure the energy debt of Android applications. The study claimed that the plugin utilizes a comprehensive, strong, and expandable lists of smells that are well enough to help in reduction of energy consumption. They assumed that the addition of new features on each releases will add new energy code smells resulting in the increase of energy debt. Furthermore, they introduced the direct relationship between the cost of maintaining energy code smells and the duration for which the release remains functional. While calculating the energy debt,the research also suggested to consider how frequent the code block that contains energy smells is executed and re-utilized. The rules in this plugin may not add value while listing the rules for the plugin designed for this study, but it introduces the effect of energy code smells that can have in software development cycle.

VII. Conclusion

In this research, the primary research question, *"How can static code analysis contribute to the detection and rectification of energy code smells in different loop structures in Java?"* was addressed through three sub-research questions. Firstly, the plugin was tested in multiple real-world Java projects to answer the first sub-research question. Subsequently, the second sub-research question was addressed by illustrating how the plugin detected energy code smells in the selected projects. Lastly, the third sub-research question was answered by recommending code samples to rectify the identified energy code smells.

To test the plugin, 16 open-source Java projects were selected out of 10 different owners. Among these projects, the highest number of energy code smells, specifically 547 energy code smells consisting of 184 major and 363 minor energy code smells, were detected in *The Algorithms Java*. On the other hand, only 2 energy code smells, comprising 1 major and 1 minor energy code smells, were identified in *COLA*. In addition, for every detected energy code smell, the plugin had suggested code samples to fix it along with the rationale behind the energy code smell. Subsequently, professionals reviewed the plugin, and it received an overall rating of *Very Good* based on the feedback received.

In conclusion, the plugin effectively detected code smells and suggested code samples to rectify the detected code smells. Nevertheless, it cannot be denied that the plugin may not immune to generating false positives.

A. Future Work

The next steps involve expanding the plugin with additional rules to detect more energy code smells in Java loop types. Additionally, there are plans to extend plugin support to other commonly used programming languages. Currently, the plugin provides sample code suggestions for fixing detected issues; the next phase focuses on developing automated code refactoring tools based on plugin analysis results to enhance the efficiency of energy-consuming loops. Further refinement and performance optimization will be done to improve accuracy and efficiency, followed by collaboration with the industry.

Acknowledgment

The authors would like to thank all the industrial experts who generously volunteered their time and expertise to test the plugin. Those feedback really add value to this study. This research work is designed as a Master's thesis for the Erasmus Mundus programme, Software Engineers For Green Deal(SE4GD).

References

[1] Lotfi Belkhir and Ahmed Elmeligi. *Assessing ICT global emissions footprint: Trends to 2040 & recommendations. Journal of Cleaner Production*, 177:448–463, 2018. Elsevier.

[2] Anne E Trefethen and Jeyarajan Thiyagalingam. *Energy-aware software: Challenges, opportunities and strategies. Journal of Computational Science*, 4(6):444–449, 2013. Elsevier.

[3] Niklaus Wirth. *A plea for lean software*. *Computer*, 28(2):64–68, 1995. IEEE.

[4] Giuseppe Procaccianti. *Energy-efficient software*. *Amsterdam: Vrije Universiteit*, 2015.

[5] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Anjum Naveed, Kwangman Ko, and Joel JPC Rodrigues. *A case and framework for code analysis–based smartphone application energy estimation*. *International Journal of Communication Systems*, 30(10):e3235, 2017. Wiley Online Library.

[6] Martin Fowler and Kent Beck. *Refactoring: Improving the design of existing code*. In *11th European Conference. Jyväskylä, Finland*, 1997.

[7] Doohwan Kim, Jang-Eui Hong, Ilchul Yoon, and Sang-Ho Lee. *Code refactoring techniques for reducing energy consumption in embedded computing environment*. *Cluster Computing*, 21:1079–1095, 2018. Springer.

[8] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. *Are SonarQube rules inducing bugs?*. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 501–511, 2020. IEEE.

[9] Gustavo Pinto and Fernando Castor. *Energy efficiency: a new concern for application software developers*. *Communications of the ACM*, 60(12):68–75, 2017. ACM New York, NY, USA.

[10] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. *What do programmers know about software energy consumption?*. *IEEE Software*, 33(3):83–89, 2015. IEEE.

[11] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. *On the impact of code smells on the energy consumption of mobile applications*. *Information and Software Technology*, 105:43–55, 2019. Elsevier.

[12] Panagiotis Louridas. *Static code analysis*. *IEEE Software*, 23(4):58–61, 2006. IEEE.

[13] Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. *An overview on the static code analysis approach in software development*. *Faculdade de Engenharia da Universidade do Porto, Portugal*, 2009.

[14] Herbert Prähofer, Florian Angerer, Rudolf Ramler, Hermann Lacheiner, and Friedrich Grillenberger. *Opportunities and challenges of static code analysis of IEC 61131-3 programs*. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–8, 2012. IEEE.

[15] Alexandru G Bardas and others. *Static code analysis*. *Journal of Information Systems & Operations Management*, 4(2):99–107, 2010. Romanian-American University.

[16] Javier García-Munoz, Marisol García-Valls, and Julio Escribano-Barreno. *Improved metrics handling in SonarQube for software quality monitoring*. In *Distributed Computing and Artificial Intelligence, 13th International Conference*, pages 463–470, 2016. Springer.

[17] Ping Yu, Yijian Wu, Jiahan Peng, Jian Zhang, and Peicheng Xie. *Towards Understanding Fixes of SonarQube Static Analysis Violations: A Large-Scale Empirical Study*. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 569–580, 2023. IEEE.

[18] Olivier Le Goaer and Julien Hertout. *ecoCode: a SonarQube Plugin to Remove Energy Smells from Android Projects*. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

[19] SonarSource S.A. *Adding coding rules*. Available online at: https://docs.sonarsource.com/sonarqube/latest/extension-guide/adding-coding-rules/

[20] María José Salamea and Carles Farré. *Influence of developer factors on code quality: A data study*. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 120–125, 2019. IEEE.

[21] *Architecture and Integration*. Available online at: https://scm.thm.de/sonar/documentation/architecture/architecture-integration/

[22] Ifeanyi Rowland Onyenweaku, Michael Scott Brown, Michael Pelosi, and MH Shahine. *A SonarQube Static Analysis of the Spectral Workbench*. *International Journal of Natural Science and Reviews*, 6:16, 2021.

[23] SonarSource. *Issues*. Available online at: https://docs.sonarsource.com/sonarqube/latest/user-guide/issues/

[24] Victor R Basili Caldiera and H Dieter Rombach. *The goal question metric approach*. In *Encyclopedia of Software Engineering*, pages 528–532, 1994.

[25] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[26] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. *Design science in information systems research*. *Management Information Systems Quarterly*, 28(1):6, 2008.

[27] Bubaker F Shareia. *Qualitative and quantitative case study research method on social science: Accounting perspective*. *International Journal of Economics and Management Engineering*, 10(12):3849–3854, 2016.

[28] Vibha Pathak, Bijayini Jena, and Sanjay Kalra. *Qualitative research*. *Perspectives in Clinical Research*, 4(3), 2013. Medknow Publications & Media Pvt. Ltd.

[29] Ranjit Biswas. *An application of fuzzy sets in students' evaluation*. *Fuzzy Sets and Systems*, 74(2):187–194, 1995. Elsevier.

[30] Danilo Nikolić, Darko Stefanović, Dušanka Dakić, Sran Sladojević, and Sonja Ristić. *Analysis of the tools for static code analysis*. In *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6, 2021. IEEE.

[31] Cristian Mateos, Ana Rodriguez, Mathias Longo, and Alejandro Zunino. *Energy implications of common operations in resource-intensive Java-based scientific applications*. In *New Advances in Information Systems and Technologies*, pages 739–748, 2016. Springer.

[32] Ding Li and William GJ Halfond. *An investigation into energy-saving programming practices for Android smartphone app development*. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, pages 46–53, 2014.

[33] Leticia Duboc, Stefanie Betz, Birgit Penzenstadler, Sedef Akinli Kocak, Ruzanna Chitchyan, Ola Leifler, Jari Porras, Norbert Seyff, and Colin C Venters. *Do we really know what we are building? Raising awareness of potential Sustainability Effects of Software Systems in Requirements Engineering*. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 6–16, 2019. IEEE.

[34] Ana Ribeiro, João F Ferreira, and Alexandra Mendes. *Ecoandroid: An Android Studio plugin for developing energy-efficient Java mobile applications*. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 62–69, 2021. IEEE.

[35] Khashayar Etemadi Someoliayi, Nicolas Yves Maurice Harrand, Simon Larsen, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom, and Martin Monperrus. *Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations*. *IEEE Transactions on Dependable and Secure Computing*, 2022. IEEE.

[36] Sara L Gill. *Qualitative sampling methods*. *Journal of Human Lactation*, 36(4):579–581, 2020. SAGE Publications Sage CA: Los Angeles, CA.

[37] Shari L Dworkin. *Sample size policy for qualitative studies using in-depth interviews*. *Archives of Sexual Behavior*, 41:1319–1320, 2012. Springer.

[38] Davide Falessi and Alexander Voegele. *Validating and prioritizing quality rules for managing technical debt: An industrial case study*. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 41–48, 2015. IEEE.

[39] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. *Detecting energy bugs in Android apps using static analysis*. In *Formal Methods and Software Engineering: 19th International Conference on Formal Engineering Methods, ICFEM 2017, Xi'an, China, November 13-17, 2017, Proceedings*, pages 192–208, 2017. Springer.

[40] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. *jstanley: Placing a green thumb on Java collections*. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 856–859, 2018.

[41] JM Cardoso, José Gabriel F Coutinho, and Pedro C Diniz. *Chapter 5-Source code transformations and optimizations*. In *Embedded Computing for High Performance*, pages 137–183, 2017.

[42] Daniel Maia, Marco Couto, João Saraiva, and Rui Pereira. *E-debitum: Managing software energy debt*. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, pages 170–177, 2020.