INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING AND COMPUTER SYSTEMS
E-ISSN: 2180-0650
VOLUME 11, ISSUE 2, 2025, pp. 176 – 187
DOI: https://doi.org/10.15282/ijsecs.11.2.2025.14.0146

RESEARCH ARTICLE

Investigation of code smells in Eclipse Framework using SonarQube: An empirical analysis

Simon Kawuma[1*], David Sabiiti Bamutura[2], Enock Mabberi[1], Moreen Kabarungi[1], Dickson Kalungi[3], Aggrey Obbo[1], Evarist Nabaasa[2]

[1]Software and Informatics Engineering Department, Mbarara University of Science and Technology, Uganda
[2]Computer Science Department, Mbarara University of Science and Technology, Uganda
[3]Information Technology Department, Mbarara University of Science and Technology, Uganda

Abstract - The Eclipse Framework provides both public and internal Application Programming Interfaces (APIs). Public APIs are widely supported and encouraged for use, while internal APIs are considered immature and subject to frequent changes. However, the quality of these APIs is not guaranteed, with many users reporting code smells, which could lead to application failures if unresolved. While some studies indicate that not all code smells can be easily fixed, users often face the challenge of either addressing these issues themselves or abandoning problematic APIs. To address this, we conducted an empirical investigation using the SonarQube static code analysis tool on 28 major Eclipse releases, aiming to identify code-smell-free APIs. Our study provides a dataset of 218K code-smell-free public APIs and 321K internal APIs. We found that 87.3% of public APIs and 91.5% of internal APIs in the analysed releases are free from code smells, highlighting the importance of using these cleaner alternatives for application stability and long-term usability. Furthermore, we have discovered that the number of code smells proportionately increases as the Eclipse Framework evolves. The average number of code smells and technical debt is 147K and 2,744 days, respectively, in all the studied Eclipse releases. Results from this study can be used by both interface providers and users as a starting point to recognize code smell-free interfaces and estimate efforts needed to fix code smells in each version of Eclipse.

Article History
Received : 19 March 2024
Revised : 25 March 2025
Accepted : 04 August 2025
Published : 26 January 2026

Keywords
Eclipse
Public APIs
Internal APIs
Code Smell
Software Quality

1. Introduction

Frameworks and libraries serve as the foundation for application development [1]. This approach in application development encourages functionality reuse [2] and boosts productivity [3]. This is the rationale behind major application frameworks, such as Eclipse [4], jBPM [5], JDK [6] and JUnit [7], consistently offering public (stable) Application Programming Interfaces (APIs) for developer use. In addition to public APIs, all these frameworks offer internal APIs. Eclipse is a popular and widely accepted application framework. Eclipse is a vast and complex open-source software system used by thousands of application developers. Eclipse has evolved for more than two decades, with over 28 major and 55 minor versions. Eclipse, jBPM, and jUnit follow the convention of internal interfaces by using the substring internal in their package names, but JDK's internal API packages begin with the substring sun [6].

Framework developers encourage the use of public APIs because they are considered stable, mature, and supported, whereas internal APIs are discouraged because they are unstable, unsupported, immature, and subject to change or removal without notice [3], [4]-[8]. Despite the fact that internal APIs are discouraged, they are widely used. Businge et al. [9] discovered that around 44% of the 512 Eclipse plug-ins employ internal APIs. Hora et al. [10] found that 23.5% of 9702 Eclipse client projects stored on GitHub relied on internal APIs. Experienced application developers stated that using internal APIs is a better option than creating their own APIs from scratch [11]. Usage of both public APIs and internal APIs by developers is inevitable because, when used, development time is reduced, and thus the application can reach its market within a shorter period of time. Although interface providers claim that public APIs are supported, in contrast, internal APIs are unsupported; there is no guarantee that these interfaces are well tested because several code smells are reported by interface users [12].

Code smells reveal something wrong with the underlying code of the application, which can lead to the eventual failure of an application or kill its performance. They include duplicate code, long methods, comments, long parameter lists, unnecessary primitive variables, dead code, and data clumps [13]. These can affect the speed of development and maintenance activities and may also become a detriment to an application program. The study of code smells is gaining importance to analyse different aspects of software quality because these can point to problems related to reusability, testability [14], structure, efficiency, maintainability, and readability of code [15]. Code smells can lead to maintenance difficulties in software systems, and classes that have code smells are more prone to change [16]. Code smells in the source code of a program are certain structures caused by design flaws or bad coding idioms that indicate a deeper problem, impact software quality factors [17], and hinder software evolution and maintainability [18]. Gradivsnik and Hericko [19] look at code smells as “code that does its purpose but does not seem to be just right”. These smells appear in software as a result of rushed implementation to meet deadlines, also referred to as technical debt, and rushed maintenance activities that give priority to feature delivery more than code quality [14]. Apart from schedule pressure,

*CORRESPONDING AUTHOR | Simon Kawuma| ✉ simon.kawuma@must.ac.ug
© The Author(s) 2025. Published by Universiti Malaysia Pahang Al-Sultan Abdullah Publishing. This is an open access article under the CC BY-NC 4.0 International license

these smells can also be caused unintentionally by careless or incompetent developers or even intentionally through lack of verification and poor processes to speed up development [19].

Applications that might use code-smelly interfaces risk failing if the code smells are not fixed. This implies that the application developer must be ready to fix the code smells themselves. Code smell fixing can be done by the framework developers on behalf of the user, following a typical process of code smell fixing, which may include refactoring the code [13], [20]. Before refactoring, the initial process includes identifying code fragments that violate the semantic properties or structure, for instance, the complexity or coupling [21]. Then the code smell can be assigned to a developer who does the fixing. This is followed by reviewing the fixed code to verify if the code smell is resolved. Previous studies [21], [22] have found that software maintainers spend at least 60% of their time comprehending the code, and maintenance accounts for around 80% of software cost. Therefore, fixing and resolving code smells is characterised by a long period; hence, the user might wait indefinitely for a solution from the developer.

Although software with code smells is not synonymous with the presence of bugs, it is characterized by slowed processing [23], poor quality code, and susceptibility to the introduction of bugs during refactoring as the framework evolves, which increases the risk of software failures and technical debt. Therefore, API users are left with no choice but to either fix the code smells themselves or abandon the interfaces because "code-smelly" API reuse leads to longer development time and high maintenance costs. As a solution to avoid huge costs, waiting indefinitely for solutions from interface developers, or getting involved in code smell fixing, users should use code smell-free interfaces. Unfortunately, these users may be unaware that the Eclipse Framework has code smell-free interfaces.

Since API users manually search for the functionality they require in the Eclipse Framework [11] interface users may first encounter code-smelly interfaces rather than code smell-free interfaces due to the fact that Eclipse is a vast and complex software framework. Therefore, the goal of this paper is to empirically investigate the presence of code smell-free interfaces in 28 Eclipse Framework major releases and recommend the code smell-free interfaces to application developers. To obtain raw data, we used SonarQube [24], a static code analyser tool, to investigate and retrieve code smell-free interfaces. To achieve the study goal, we developed five research questions:

- RQ1: What is the number of code smells in Eclipse Frameworks?
- RQ2: What is the technical debt needed to fix code smells in Eclipse Frameworks?
- RQ3: What is the percentage of code smell-free internal interfaces in Eclipse Framework?
- RQ4: What is the percentage of code smell-free public interfaces in Eclipse Framework?
- RQ5: What is the most common code smell in Eclipse Frameworks?

Although several studies have been conducted on code smell detection and analysis, no study has been carried out to ascertain the existence of code smell-free interfaces in Eclipse Frameworks; therefore, the contributions of this study are threefold:

1) We provide a dataset of 218K and 321K code smell-free public API and internal API classes, respectively, to Eclipse interface providers and users. Interface providers can use this dataset to estimate the effort needed to remove code smells. Users can look up code smell-free interfaces they want to use when developing their applications.
2) The Eclipse interface providers claim that public APIs are good and stable interfaces [4]. Indeed, this research study has empirically confirmed that public APIs are good since we have discovered that over 87.3% of public APIs are code smell-free in all studied Eclipse releases.
3) Internal APIs are discouraged by interface providers because they are often immature and unsupported; however, this research study has empirically confirmed that not all internal APIs are bad since over 91.5% of the internal APIs were code smell-free in all studied Eclipse releases, and thus users can use them when developing their applications.

The remainder of the paper is organized as follows: Section 2 discusses related work; Section 3 presents the Research Methodology, and Section 4 discusses experimental results. Finally, Section 5 concludes the paper and suggests some areas for future research.

2. Related Work

One of the conclusions of prior studies by Businge et al. [11] was that interface users are always utilizing unstable interfaces, and the reason for this is that there are no alternative stable interfaces that provide the same functionality. Indeed, Kawuma et al. [25] demonstrated that less than 1% of APIs provide the same or similar functionality as non-APIs. In a recent study, Businge et al. [3] used a clone detection technique to investigate the stability of the internal interface as the Eclipse Framework evolved. They detected 327K stable internal interfaces and suggested them as potential candidates for promotion. Hora et al. [10] revealed that 7% of 2,277 internal interfaces were promoted to public interfaces and the promotion rate was too low. Kawuma et al. [26] confirmed that the rate at which internal APIs are promoted to public APIs is slow. The earlier studies by Businge et al. both looked at public API and internal API interfaces, but none looked at finding code smell-free interfaces as compared to our study.

Guerrouj et al. [27] conducted research on 30 versions of three projects: ANT, ArgoUML, and Hibernate, to determine the association between lexical smell and software quality, as well as their interaction with design smells. They detected 29 smells, including 13 design smells and 16 lexical smells. Seref and Tanriover [28] carried out a survey on existing literature about software code maintainability. In their survey, they discovered that all authors stated that maintainability increases the quality of software and is one of the most important attributes. Jafari et al. [29] investigated dependency smell, which concerns developers who want to stay up to date with the latest features and fixes while ensuring backward

compatibility. They examined the commit data for a dataset of 1,146 active JavaScript repositories to catalogue, quantify, and understand dependency smells and conducted a series of surveys with practitioners who identified and quantified seven dependency smells with varying degrees of popularity. They also discovered that two or more distinct smells appear in 80 percent of JavaScript projects and that dependency smells cause security threats, runtime errors, and dependency breakage.

Previous survey studies have examined the issues of code smells, including definitions, detection methodologies, and refactoring tools [30]-[32]. Several research [33]-[36] examined developers' awareness of code smells, perceptions, and motivations for deleting them, while other studies [30], [37]-[39] focused on refactoring activities and challenges associated with them. Others [20], [40]-[41] examined the methods and algorithms utilized in detection and refactoring tools. Furthermore, Tufano et al. [42] studied the evolution of code smells in 200 open-source Java systems, and they discovered that code smells are introduced in the code by both new and experienced developers at the start of the project. They also noted that 80% of them survive across the evolution of the project, and of the 20% removed, only 8% are due to refactoring. This implies that code smells persist over the evolution of software ecosystems and are not easily addressed. Johannes et al. [12] examined the effect of code smells on the fault-proneness of JavaScript server-side projects; from their study, they concluded that code smells should be detected and addressed before the next release.

Several machine learning and deep learning techniques, such as classical Machine Learning algorithms and Deep Learning algorithms, have been proposed in previous work to detect code smells [43]-[46]. Zhang and Dong [47] presented a code smell detection approach that combines the residual network (ResNet) with the metric-attention mechanism. To evaluate their approach, they labelled a large-scale dataset by applying a heuristic-based tool and found out that their method outperforms the traditional practice of training a statistical ML model on a set of code metrics. Mhawish and Gupta [48] presented a method using different ML algorithms and software metrics to detect code smells based on 74 software systems. The experimental results showed that ML techniques have high potential in predicting code smells but have not been compared with rule-based systems such as SonarQube.

Several systematic surveys on code smells have been conducted; for example, Fawad et al. [49] carried out a systematic literature review between November 2007 and December 2023 to provide a comprehensive summary of techniques, tools, and approaches used for detecting and refactoring code smells in Android applications. They discovered a total of 237 code smells that were identified using 51 techniques and tools, and the code smell "Durable Wakelock" was among the most studied. Furthermore, Wu et al. [50] conducted a systematic literature review on Android-specific code smells and highlighted their significance and impact on end-user experience. They analysed 4,820 papers published until 2021 and selected 35 studies. They discovered that several proposed approaches focused on detecting performance-related smells. Hurtado Alegría et al. [51] performed a systematic literature review on 145 studies until March 2018 to explore, analyse, and identify different techniques to detect software smells at a design level. The primary objective was to analyse all tools and techniques, regardless of their implementation status, and detect code and design smells. They observed that 57% of the studies did not use any performance measure, and 41% overlooked the details of the targeted programming languages.

Some of the above studies look at public APIs and internal interfaces by identifying and recommending internal interfaces for promotion. Other studies look at code smell definition, detection tools, evolution, and machine learning algorithms for detection of code smells, but none of the above authors studied code smell composition, distribution, remediation efforts (Technical Debt) to fix them, and also the existence of code smell-free interfaces in the Eclipse Framework as compared to our study. Furthermore, there has not been any empirical investigation on code smells in Java Frameworks like Eclipse, except work done on JavaScript applications and Android applications, but limited to energy usage.

3. Materials and Methods

This section describes the experimental setup used to collect data for the study questions.

3.1 Eclipse Releases Collection

This section explains the data sources for our investigation. Our investigation was based on 28 Eclipse SDK major releases from the Eclipse project archive website [52], [53]. Table 1 presents the different Eclipse major releases that were considered in this research. The first and fifth columns show the major releases, while the second and sixth columns show their corresponding release date. The third and seventh columns show the Java lines of code (LOC) in each major Eclipse release, while the fourth and eighth columns show the total number of Java classes in each Eclipse major release. This research study chose Eclipse as a topic of study because it is a widely used and embraced open-source framework that will continue to draw new developers. The Eclipse Framework is constantly changing, with a new version being released every three months. This provides an opportunity to examine code smell evolutionary tendencies as the framework evolves. This study focused on Eclipse major versions because, as the framework evolves from one major version to the next, new projects, sub-projects, packages, classes, interfaces, fields, and methods are either introduced, updated, or removed.

Table 1. Eclipse major releases and their corresponding release dates

Major Release	Release Date	Java LOC	Java Classes
E-1.0	07-Nov-01	449K	4,608
E-2.0	27-Jun-02	769K	6,751
E-2.1	27-Mar-03	959K	7,911
E-3.0	25-Jun-04	1.3M	10,634
E-3.1	27-Jun-05	1.6M	12,299
E-3.2	29-Jun-06	2M	14,941
E-3.3	25-Jun-07	2.1M	16,036
E-3.4	17-Jun-08	2.5M	18,800
E-3.5	11-Jun-09	2.6M	19,169
E-3.6	08-Jun-10	2.7M	20,922
E-3.7	13-Jun-11	2.75M	21,104
E-3.8	27-Jun-12	2.8M	22,477
E-4.0	27-Jul-10	2.6M	20,498
E-4.1	20-Jun-11	2.7M	21,234
E-4.2	27-Jun-12	2.8M	22,443
E-4.3	05-Jun-13	2.9M	22,798
E-4.4	06-Jun-14	3.1M	23,880
E-4.5	03-Jun-15	3.14M	23,920
E-4.6	06-Jun-16	3.2M	23,936
E-4.7	28-Jun-17	3.3M	25,900
E-4.8	27-Jun-18	3.39M	26,180
E-4.9	19-Sept-18	3.4M	26,363
E-4.11	Mar-19	3.5M	27,448
E-4.12	Jun-19	3.51M	27,784
E-4.13	Sept-19	3.52M	27,904
E-4.14	Dec-19	3.53M	27,976
E-4.15	Mar-20	3.55M	28,500
E-4.16	Jun-20	3.6M	28,135

3.2 Code Smell Collection and Extraction Using SonarQube Tool

In this section, we describe how data required for answering research questions RQ1-RQ5 was extracted. We used the SonarQube tool (version 8.2) [24] to extract information about code smells in the different Eclipse releases. We relied on this tool because it is broadly used by thousands of users in academic research settings [54]-[56] and in industry [57]-[59]. The tool is actively maintained and can support commonly used development languages such as Java, Python, and C++ [60].

However, just like any other static code analysis tool, it doesn't have 100% precision, implying that it is possible that results could differ if a different tool is used. Nevertheless, the tool is actively maintained and widely used compared to other tools. We configured and ran SonarQube on a local computer using all 432 maintainability rules that cover code smell detection [61]. When any of the rules are violated, that source code manifests as a code smell. We investigated the total number of code smells, the code smell remediation effort to fix all code smells, and collected information about the most dominant code smell type, i.e., the most violated rule, for every Eclipse major release. It is important to note that SonarQube estimates the code smell remediation effort in days, and an 8-hour day is assumed [24].

Figure 1 illustrates the procedure we followed to detect and extract code smell information in all the analysed Eclipse releases. SonarQube takes source directories containing Java files as input to detect possible code smells at specific points in the class. Then it produces output reports for each Eclipse release, which can be accessed from the SonarQube server via the URL: http://localhost:9000. An excerpt of a sample output report for Eclipse-4.16 obtained from SonarQube is shown in Figure 1. Each file in the report has a Maintainability Rating (MR) assigned by SonarQube depending on the nature and number of code smells found in the class of source files under investigation. For example, as illustrated in Figure 1, the file in the last row has an MR rating of 'A,' indicating that it is free from code smells. The tool counts the number of code smells reported in each release and the Technical Debt, as shown in the report.

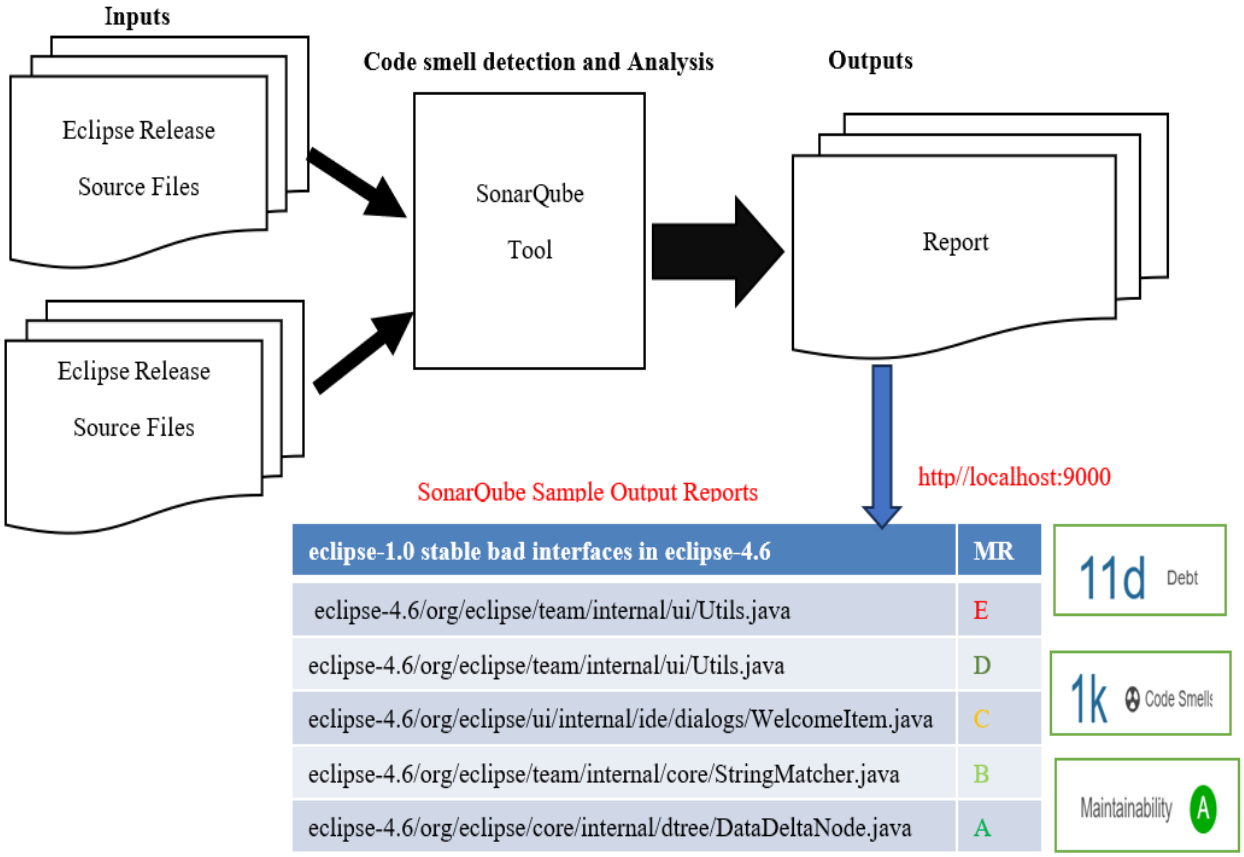


Figure 1. SonarQube code smell detection tool

3.3 Data Extraction for Number of Code Smells and Technical Debt in Eclipse Releases

In this section, we present the procedure we used to extract data for research questions: RQ1: What is the number of code smells in Eclipse Frameworks? and RQ2: What is the Technical Debt needed to fix code smells in Eclipse Frameworks? Information about the code smells and technical debt, i.e., the time needed to fix them, was extracted from the SonarQube reports. The total number of code smells and Technical Debt is provided in the reports, as illustrated in Figure 1. For example, from Figure 1, 1K code smells were detected, and 11 days of technical debt are needed to fix the identified code smells.

3.4 Data Extraction for Code Smell-Free Interfaces in Eclipse Releases

We used the SonarQube tool to extract information about code smell-free interfaces in each Eclipse release to address RQ3: What is the percentage of code smell-free internal interfaces in Eclipse Framework? and RQ4: What is the percentage of code smell-free public interfaces in Eclipse Framework? We considered interfaces that have a maintainability rating of A. To determine the percentage of code smell-free public interfaces and internal interfaces in each Eclipse release, we counted both the number of classes with and without the substring internal in their file path for internal interfaces and public interfaces, respectively. We used Eq. (1) and Eq. (2) below to calculate the percentage of internal and public code smell-free interfaces, respectively, by looking at public APIs and internal APIs individually in each Eclipse release, as shown below;

$$Code\ smell\ free\ internal\ interface = \frac{Number\ of\ internal\ \ Interface\ with\ Maintainabilty\ rating\ A}{Total\ number\ of\ internal\ interface\ in\ the\ Eclipse\ releases} * 100\% \quad (1)$$

$$Code\ smell\ free\ public\ interface = \frac{Number\ of\ public\ Interface\ with\ Maintainabilty\ rating\ A}{Total\ number\ of\ public\ interface\ in\ the\ Eclipse\ releases} * 100\% \quad (2)$$

To determine the percentage of code smell-free public APIs and internal APIs as a combination in a given Eclipse release, code smell-free public APIs and internal APIs were each computed separately in each release, and then the total number of both public and internal APIs in a given release was calculated. To get the percentage comparison of code smell-free public and internal APIs, we used Eq. (3) and Eq. (4) respectively, as shown below;

$$Code\ smell\ free\ internal\ interface = \frac{Number\ of\ internal\ \ Interface\ with\ Maintainabilty\ rating\ A}{Total\ \ number\ of\ interfaces\ in\ the\ Eclipse\ releases} * 100\% \quad (3)$$

$$Code\ smell\ free\ public\ interface = \frac{Number\ of\ public\ Interface\ with\ Maintainabilty\ rating\ A}{Total\ \ number\ of\ interfaces\ in\ the\ Eclipse\ releases} * 100\% \quad (4)$$

3.5 Data Extraction for the Commonest Code Smell in Eclipse Releases

To address research question RQ5: What is the commonest code smell in Eclipse Frameworks? In addition to the detection of code smells, SonarQube also mentions the violated maintainability rule together with the code smell that violated it. To establish the most common code smell, we compute the frequency of violation of each rule.

4. Results

4.1 Number of code smells and technical debt in Eclipse releases

Figure 2 below presents the total number of code smells together with the Technical Debt needed to fix them in all the different Eclipse releases. In this figure, the bar graphs represent the total number of code smells while the line graph represents the Technical Debt, i.e., the remediation effort needed to fix the code smells in Eclipse releases. Focusing on the bar graph in this figure, we see a proportionate increase in the number of code smells and Technical Debt. Furthermore, there is a decline in the number of code smells and Technical Debt between Eclipse-3.8 and Eclipse-4.0, and thereafter a proportionate increase is observed after Eclipse-4.0. The slight change between Eclipse-3.8 and Eclipse-4.0 can be attributed to the fact that some classes were deleted from Eclipse-3.8, as observed from Table 1; thus, more code smells were deleted, which further led to less Technical Debt to fix the code smells. From Figure 2, we observe that the minimum and maximum number of code smells detected are between 31,199 and 186,590 across the studied Eclipse releases. Whereas the minimum and maximum efforts needed to fix the identified code smells range from 545 to 3,391 days in all analysed Eclipse releases. The average number of code smells and technical Debt is 147,277 and 2,744 days in all the studied Eclipse releases.

4.2 Percentage of code smell-free interfaces in Eclipse releases

Figures 3 and 4 present results corresponding to the percentage of code smell-free interfaces in different Eclipse major releases. Focusing on Figure 3, for each Eclipse release, the first and second bars present the percentage of code smell-free classes for internal APIs and public APIs, respectively. In the same figure, we observe that there are over 91.5% and 87.3% code smell-free internal APIs and public APIs, respectively. In Figure 4, the bar graph presents code smell-free public APIs and internal APIs as a percentage of the total number of interfaces in each Eclipse release, while the line graph presents the percentage of the total number of code smell-free interfaces (i.e., both public APIs and internal APIs) with respect to the total number of interfaces in a given release.

From Figure 4, and specifically focusing on the bar graph, we see that the majority of code smell-free classes are internal APIs compared to public APIs. This is because internal APIs are twice as many as public APIs during the evolution of Eclipse [25]. The percentage of code smell-free public APIs ranges from 24.9%-46.9% whereas that of internal APIs is between 47.4%-68.2% of the total interfaces, respectively, in all the analysed Eclipse releases. On average, 36.1% and 57.2% of the total interfaces in each Eclipse release are code smell-free public APIs and internal APIs, respectively. Focusing on the line graph in Figure 4, we observe that over 89.6% of the total number of classes in each Eclipse release are code smell-free.

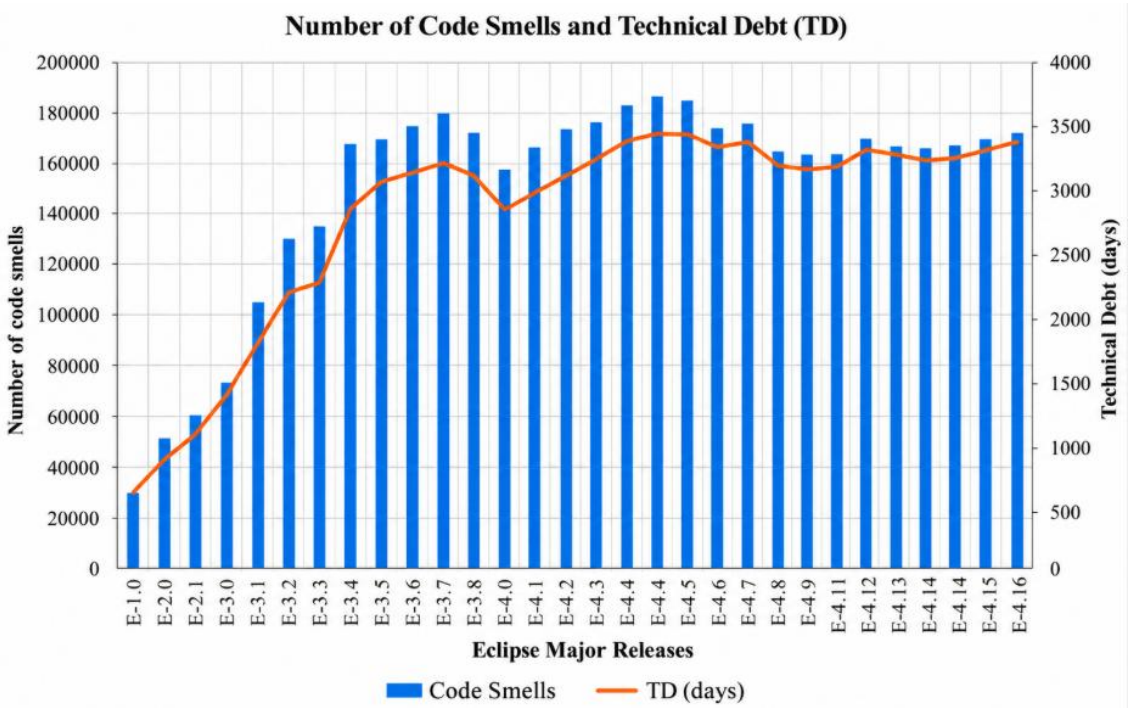


Figure 2. Number of code smells and technical debt in Eclipse releases

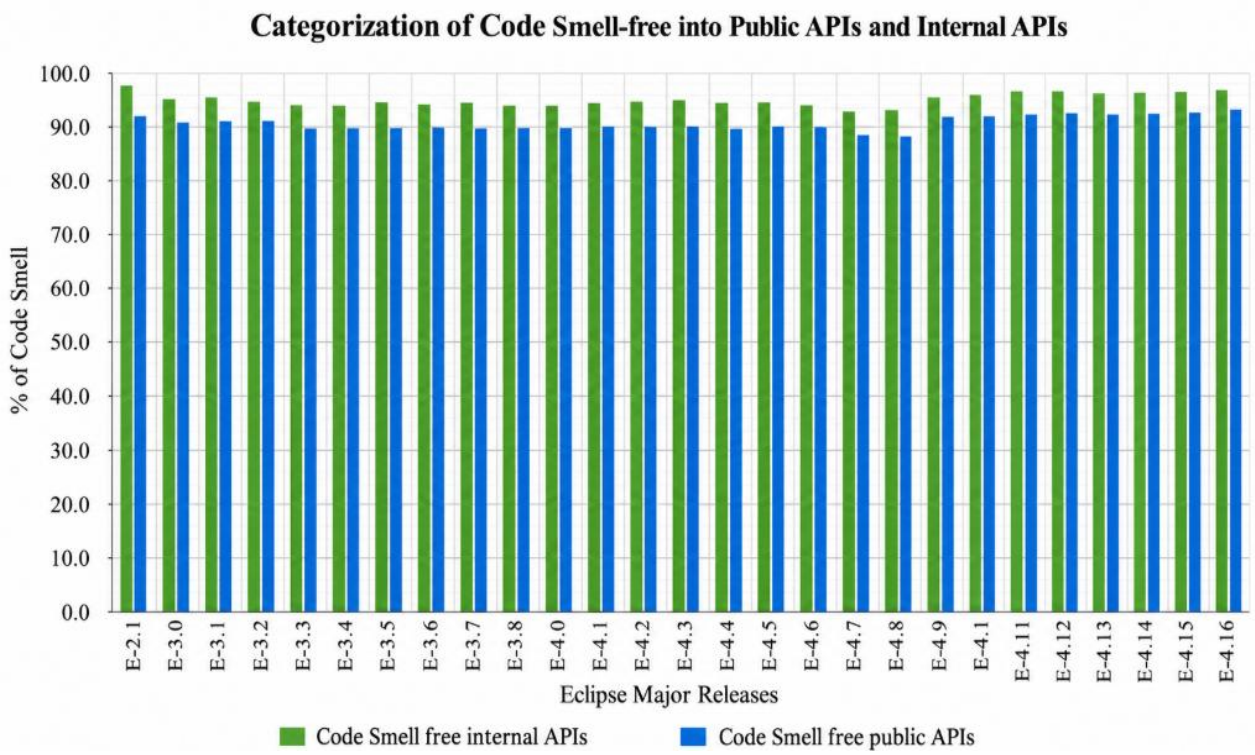


Figure 3. Distribution of code-smell-free public APIs vs. internal APIs across different Eclipse releases

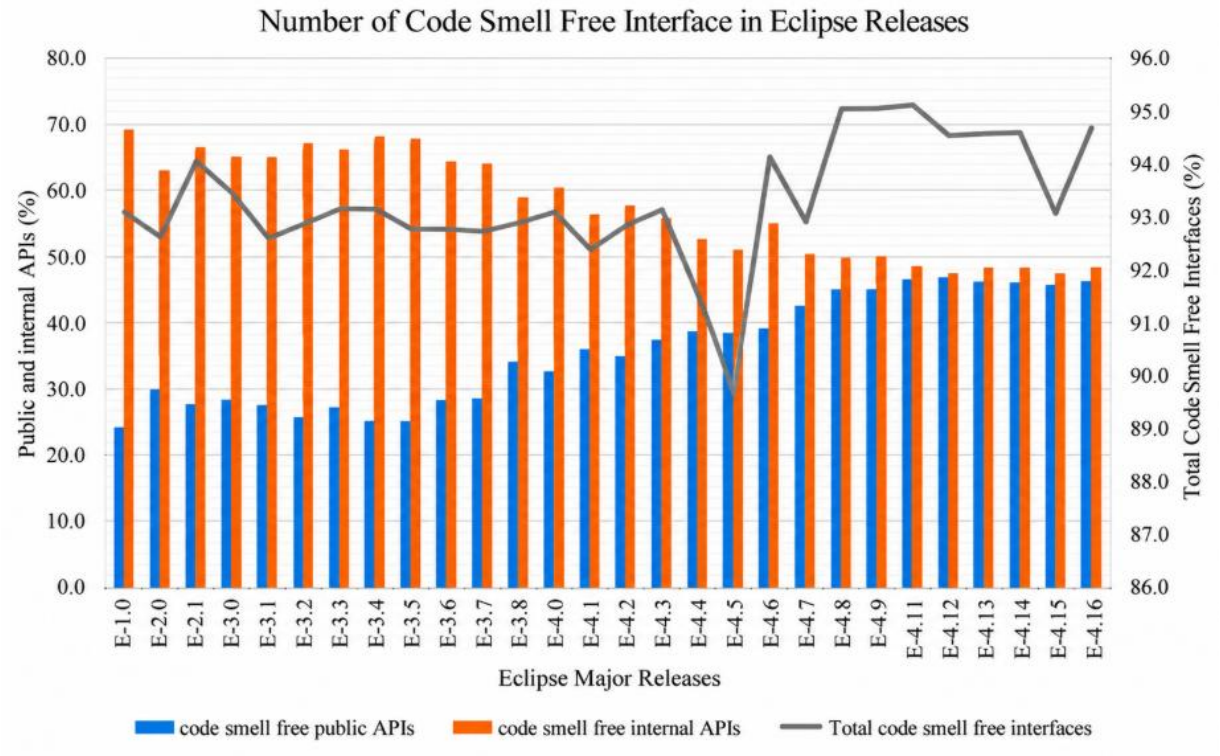


Figure 4. Distribution of code smell-free interfaces in Eclipse releases

4.3 Common code smells in Eclipse releases

In this section, we present the results of the common code smells found in Eclipse releases. Maintainability rules create code violations that represent something wrong in the code, which will be reflected as a code smell. Table 2 presents the 25 most common maintainability rules along with the number of code smells that arose because of them being violated. The first column in the table shows the unique rule ID, whereas the second column shows a brief description of the rule.

The third column shows the total number of code smells that are generated because of violating that rule in all the analysed Eclipse releases.

Table 2. Top 25 common code smells in Eclipse releases

Rule	Rule Description	# Code smell
S3008	Static non-final field names should comply with a naming convention	575,905
S3776	Cognitive Complexity of methods should not be too high	233,011
S1172	Unused method parameters should be removed	211,329
S100	Method names should comply with a naming convention	189,485
S115	Constant names should comply with a naming convention	157,628
S125	Sections of code should not be commented out	148,795
S1124	Modifiers should be declared in the correct order	133,779
S1149	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	126,485
S1659	Multiple variables should not be declared on the same line	106,115
S1066	Collapsible "if" statements should be merged	96,240
S116	Factory method injection should be used in "@Configuration" classes	93,196
S117	Local variable and method parameter names should comply with a naming convention	88,008
S106	Standard outputs should not be used directly to log anything	86,475
S2293	The diamond operator ("<>") should be used	86,027
S1133	Deprecated code should be removed	81,984
S1874	"@Deprecated" code should not be used	76,665
S1186	Methods should not be empty	76,199
S131	"switch" statements should have "default" clauses	75,989
S2129	Constructors should not be used to instantiate "String", "BigInteger", "BigDecimal"	69,402
S1135	Track uses of "TODO" tags	68,138
S1121	Assignments should not be made from within sub-expressions	60,532
S108	Nested blocks of code should not be left empty	54,563
S1199	Nested code blocks should not be used	53,219
S1168	Empty arrays and collections should be returned instead of null	46,789
S1192	String literals should not be duplicated	44,281

5. Discussion

In this study, we used the SonarQube static analysis tool to study code trends in the Eclipse Framework with a focus on establishing whether there exist code smell-free interfaces that we can recommend to developers. We chose SonarQube because it is open-source and therefore available for use and can detect code smells early enough during development, when they are cheap to fix, as revealed by [18], [29]. This tool generates warnings based on well-defined programming rules for finding code smells [18]. For example, SonarQube has 432 rules for detecting code smells, and once any of these rules is violated, a code smell will manifest that a developer can fix. However, static analysis tools give a large volume of warnings, which becomes a burden to users when they must analyse the output generated by these tools [29], but the SonarQube tool has user-friendly and interactive interfaces that developers can use to carry out analysis of the code smell report.

From Figure 2, we generally observed a proportionate increase in the number of code smells across all the analysed Eclipse major releases. This trend can be attributed to the fact that as the Eclipse Framework evolves, new functionality is added to it, for example, more projects, classes, and methods, and hence the lines of code (LOC) increase. Therefore, the added functionalities come with new code smells. In addition, Eclipse has a large community of developers and committers who contribute to its large code base [26]. Furthermore, the total number of code smells discovered would give insight into how much time and effort are needed by both the framework developer and interface users to remove code smells in each Eclipse release. From Table 2, we observe that most code smells are “static non-final field names should comply with a naming convention”, “Cognitive Complexity of methods should not be too high”, “Unused method parameters should be removed”, and “method names should comply with a naming convention”, as reported by SonarQube in all the analysed Eclipse releases. This finding is interesting because it provides information to both interfaces’ providers and users about the most common code smell; thus, API developers should adhere to good coding principles to avoid code smell in their applications. Furthermore, we observed that there are over 87.3% and 91.5% code smell-free public APIs and internal APIs, respectively, as shown in Figure 3 in all the analysed Eclipse Major releases. This finding implies that the majority of the Eclipse interfaces are well tested by their developers before they commit them to be part of the Eclipse Framework ecosystem. Furthermore, since internal APIs are immature and unsupported [3]-[4], [8] during framework evolution, one would expect to see almost all internal API classes with code smells.

However, from our investigation, we have discovered that, on average, 57.2% of the total number of classes have zero code smells for all the studied Eclipse releases. A similar study [62] used SonarQube to analyse the existence of bugs in stable internal APIs that had remained stable during the evolution of the Eclipse Framework. In that study, they discovered that over 79.8% of classes containing stable internal APIs had zero bugs. A similar study by Kawuma et al. [63] looked at the existence of bug-free APIs in Eclipse, and they discovered that there exist 217K and 302K bug-free public and internal APIs, respectively, in all the studied Eclipse releases. Although both studies used SonarQube, our study looked at code smells and considered both public APIs and internal APIs in the entire frameworks.

6. Conclusions

In conclusion, the Eclipse Framework offers both public and internal Application Programming Interfaces (APIs), with public APIs widely recommended for use, while internal APIs remain immature and frequently subject to modifications. Nonetheless, the quality of these APIs varies, with numerous reports of code smells potentially causing application failures if unresolved. Recognizing the difficulties users face in either resolving these issues or abandoning problematic APIs, this study undertook an empirical analysis using the SonarQube static code analysis tool across 28 major Eclipse releases to identify code-smell-free APIs.

The study generated a comprehensive dataset comprising 218,000 code-smell-free public APIs and 321,000 internal APIs. It was observed that 87.3% of public APIs and 91.5% of internal APIs in the analysed releases were free from code smells, underscoring the significance of adopting these cleaner APIs for enhanced application stability and long-term usability. Additionally, a proportional increase in the number of code smells was noted as the Eclipse Framework evolved, with an average of 147,000 code smells and a technical debt accumulation of approximately 2,744 days per release.

These findings provide valuable insights for both interface providers and users, assisting them in identifying and prioritizing the use of code smell-free interfaces and in estimating the effort required to mitigate code smells in various Eclipse releases. Moreover, by recognizing the patterns and trends of code smells, developers and maintainers can implement more targeted strategies for quality assurance and software maintenance. Future research could focus on examining the factors contributing to code smell proliferation and exploring automated mechanisms to proactively detect and rectify these issues. Additionally, further studies could evaluate the effectiveness of employing code smell-free interfaces in improving application performance, stability, and maintainability. Ultimately, understanding and addressing code smells can significantly enhance software quality and the overall user experience within the Eclipse ecosystem.

Acknowledgements

The authors would like to express their sincere appreciation to the staff in the Computing Services Department at Mbarara University of Science and Technology for providing space on their servers where the study experiments were conducted.

Funding

This study was not supported by any grants from funding bodies in the public, private, or not-for-profit sectors.

Authors Contribution

Simon Kawuma: Conceptualization, Methodology, Formal analysis, Resources, Data curation, Investigation, Writing – original draft, Writing – review & editing, Visualization.
David Sabiiti Bamutura: Conceptualization, Methodology, Writing – original draft, Writing – review & editing.
Enock Mabberi: Investigation, Data curation, Writing – original draft.
Moreen Kabarungi: Writing – original draft, Writing – review & editing.
Dickson Kalungi: Writing – original draft, Writing – review & editing.
Aggrey Obbo: Writing – review & editing.
Evarist Nabaasa: Conceptualization, Writing – review & editing.

Conflict Of Interest

The authors declare no conflicts of interest.

References

[1] T. Tourwé and T. Mens, "Automated support for framework-based software," in *Proc. Int. Conf. Software Maintenance (ICSM)*, 2003, pp. 148-157.
[2] D. Konstantopoulos, J. Marien, M. Pinkerton, and E. Braude, "Best principles in the design of shared software," in *Proc. 33rd Annu. IEEE Int. Computer Software and Applications Conf. (COMPSAC)*, vol. 2, 2009, pp. 287-292.
[3] J. Businge, S. Kawuma, M. Openja, E. Bainomugisha, and A. Serebrenik, "How stable are Eclipse application framework internal interfaces?" in *Proc. 26th IEEE Int. Conf. Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 117–127, https://doi.org/10.1109/SANER.2019.8668018.
[4] J. des Rivières, "How to use the Eclipse API," Eclipse Foundation. [Online]. Available: https://www.eclipse.org/articles/Article-API-Use/index.html. [Accessed: Jan. 2024].
[5] jBPM Team, "The jBPM APIs," Red Hat. [Online]. Available: https://docs.jboss.org/jbpm/v5.0/userguide/ch05.html#d0e2099. [Accessed: Jan. 2024].
[6] Oracle, "Why developers should not write programs that call 'sun' packages." [Online]. Available: https://www.oracle.com/java/technologies/faq-sun-packages.html. [Accessed: Jan. 2024].

[7] JUnit Team, "JUnit 5 user guide." [Online]. Available: https://junit.org/junit5/docs/current/user-guide/. [Accessed: Jan. 2024].
[8] Eclipse Foundation, "Provisional API guidelines," *Eclipsepedia*. [Online]. Available: https://wiki.eclipse.org/Provisional_API_Guidelines. [Accessed: Jan. 2024].
[9] J. Businge, A. Serebrenik, and M. van den Brand, "Eclipse API usage: The good and the bad," *Software Quality Journal*, vol. 23, pp. 107-141, 2015, https://doi.org/10.1007/s11219-013-9221-3.
[10] A. Hora, M. T. Valente, R. Robbes, and N. Anquetil, "When should internal interfaces be promoted to public?" in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, 2016, pp. 278-289, https://doi.org/10.1145/2950290.2950306.
[11] J. Businge, A. Serebrenik, and M. van den Brand, "Analyzing the Eclipse API usage: Putting the developer in the loop," in *Proc. 17th European Conf. Software Maintenance and Reengineering*, 2013, pp. 37-46, https://doi.org/10.1109/CSMR.2013.14.
[12] D. Johannes, F. Khomh, and G. Antoniol, "A large-scale empirical study of code smells in JavaScript projects," *Software Quality Journal*, vol. 27, pp. 1271-1314, 2019.
[13] A. Gupta, B. Suri, and S. Misra, "Code bad smells in Java source code: A systematic literature review," in *Proc. Int. Conf. Computational Science and Its Applications (ICCSA)*, 2017, pp. 665-682.
[14] A. Walker, D. Das, and T. Cerny, "Automated code-smell detection in microservices through static analysis: A case study," *Applied Sciences*, vol. 10, no. 21, Art. no. 7800, 2020, https://doi.org/10.3390/app10217800.
[15] A. D. F. Martins, C. S. Melo, J. M. Monteiro, and J. C. Machado, "Class change proneness prediction using software metrics and code smells," in *Proc. Int. Conf. Enterprise Information Systems (ICEIS)*, 2020, pp. 140-147.
[16] M. Alfadel, K. Aljasser, and M. Alshayeb, "Relationship between design patterns and code smells: An empirical study," *PLoS ONE*, vol. 15, no. 4, Art. no. e0231731, 2020, https://doi.org/10.1371/journal.pone.0231731.
[17] F. Li, M. Zhang, S. Wang, J. Zhang, and Q. Gu, "On the relative value of imbalanced learning for code smell detection," *Journal of Systems and Software*, vol. 198, Art. no. 111566, 2023, https://doi.org/10.1016/j.jss.2023.111566.
[18] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, Art. no. 7, 2017, https://doi.org/10.1186/s40411-017-0041-1.
[19] M. Gradisnik and M. Hericko, "Impact of code smells on defect rate: A literature review," *CEUR Workshop Proceedings*, vol. 2217, pp. 27-30, 2018.
[20] R. S. Menshawy, A. H. Yousef, and A. Salem, "Code smells and detection techniques: A survey," in *Proc. MIUCC*, 2021, pp. 78-83.
[21] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, pp. 529–552, 2017, https://doi.org/10.1007/s11219-016-9309-7
[22] E. Doğan and E. Tüzün, "Towards a taxonomy of code review smells," *Information and Software Technology*, vol. 142, Art. no. 106737, 2022, https://doi.org/10.1016/j.infsof.2021.106737.
[23] A. Tahir, A. Yamashita, S. Licorish, J. Dietrich, and S. Counsell, "How developers discuss code smells on Stack Overflow," in *Proc. Evaluation and Assessment in Software Engineering (EASE)*, 2018, pp. 68-78.
[24] A. Campbell, "SonarQube documentation," Jan. 2022. [Online]. Available: https://scm.thm.de/sonar/documentation/user-guide/metric-definitions/. [Accessed: Jan. 2024].
[25] S. Kawuma, J. Businge, and E. Bainomugisha, "Stable alternatives for unstable Eclipse interfaces," in *Proc. IEEE Int. Conf. Program Comprehension (ICPC)*, 2016, pp. 1–10.
[26] S. Kawuma and E. Nabaasa, "Identification of promoted Eclipse unstable interfaces using clone detection technique," *International Journal of Software Engineering & Applications*, vol. 9, no. 5, pp. 15–25, 2018, https://doi.org/10.5121/ijsea.2018.9502.
[27] L. Guerrouj, A. Azad, G. Antoniol, and Y.-G. Guéhéneuc, "Investigating the relation between lexical smells and change- and fault-proneness," *Software Quality Journal*, vol. 25, pp. 641-670, 2017.
[28] B. Seref and O. Tanriover, "Software code maintainability: A literature review," *International Journal of Software Engineering and Applications*, vol. 7, no. 3, pp. 1-16, 2016.
[29] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in JavaScript projects," *IEEE Transactions on Software Engineering*, vol. 48, no 10, pp. 3790-3807, 2022, https://doi.org/10.1109/TSE.2021.3106247.
[30] G. Lacerda, F. Petrillo, M. Pimenta, and Y.-G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review," *Journal of Systems and Software*, vol. 167, Art. no. 110610, 2020, https://doi.org/10.1016/j.jss.2020.110610.
[31] M. Agnihotri and A. Chug, "Software metrics, code smells and refactoring techniques: A systematic survey," *International Journal of Information Processing Systems*, vol. 16, no. 4, pp. 915-934, 2020.
[32] H. M. Dos Santos, V. H. Durelli, M. Souza, E. Figueiredo, L. T. Silva, and R. S. Durelli, "CleanGame: Gamifying the identification of code smells," in *Proc. Brazilian Symp. Software Engineering*, 2019, pp. 437-446.
[33] P. Meananeatra, "Identifying refactoring sequences for improving software maintainability," in *Proc. IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, 2012, pp. 406-409.

[34] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code," *Information and Software Technology*, vol. 92, pp. 223–235, 2017, https://doi.org/10.1016/j.infsof.2017.08.008.
[35] A. Yamashita and L. Moonen, "Do developers care about code smells?" in *Proc. Working Conf. Reverse Engineering (WCRE)*, 2013, pp. 242-251.
[36] D. J. Kim, "An empirical study on the evolution of test smells," in *Proc. IEEE/ACM Int. Conf. Software Engineering Companion (ICSE-Companion)*, 2020, pp. 149-151.
[37] S. Jain and A. Saha, "An empirical study on research and developmental opportunities in refactoring practices," in *Proc. Int. Conf. Software Engineering and Knowledge Engineering (SEKE)*, 2019, pp. 313-318.
[38] B. F. Békefi, K. Szabados, and A. Kovács, "A case study on the effects and limitations of refactoring," in *Proc. Informatics*, 2019, pp. 213-218.
[39] S. Vidal, I. Berra, S. Zulliani, C. Marcos, and J. A. D. Pace, "Assessing the refactoring of brain methods," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 1, pp. 1-43, 2018.
[40] A. AbuHassan, M. Alshayeb, and L. Ghouti, "Software smell detection techniques: A systematic literature review," *Journal of Software: Evolution and Process*, vol. 33, no. 3, Art. no. e2320, 2021, https://doi.org/10.1002/smr.2320.
[41] A. Kaur and G. Dhiman, "A review on search-based tools and techniques to identify bad code smells in object-oriented systems," *International Journal of Applied Engineering Research*, vol. 14, pp. 909-921, 2019.
[42] M. Tufano, F. Palomba, and G. Bavota, "When and why your code starts to smell bad," in *Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE)*, 2015, pp. 403-414.
[43] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *Journal of Systems and Software*, vol. 176, Art. no. 110936, 2021, https://doi.org/10.1016/j.jss.2021.110936.
[44] N. A. A. Khleel and K. Nehéz, "Deep convolutional neural network model for bad code smells detection based on oversampling method," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 26, no. 3, pp. 1725–1735, Jun. 2022, https://doi.org/10.11591/ijeecs.v26.i3.pp1725-1735.
[45] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "Code smell detection using ensemble machine learning algorithms," *Applied Sciences*, vol. 12, no. 20, Art. no. 10321, 2022, https://doi.org/10.3390/app122010321.
[46] A. K. Das, S. Yadav, and S. Dhal, "Detecting code smells using deep learning," in *Proc. IEEE Region 10 Conf. (TENCON)*, 2019, pp. 2081–2086.
[47] Y. Zhang and C. Dong, "MARS: Detecting brain class and brain method code smells," *Journal of Software: Evolution and Process*, vol. 36, no. 1, Art. no. e2403, 2024.
[48] M. Y. Mhawish and M. Gupta, "Predicting code smells and analysis of predictions using machine learning techniques," *Journal of Computer Science and Technology*, vol. 35, no. 6, pp. 1428–1445, 2020.
[49] M. Fawad, G. Rasool, and F. Palma, "Android source code smells: A systematic literature review," *Software: Practice and Experience*, vol. 54, no. 2, pp. 345–372, 2024.
[50] Z. Wu, X. Chen, and S. J. Lee, "A systematic literature review on Android-specific smells," *Journal of Systems and Software*, vol. 201, Art. no. 111677, 2023, https://doi.org/10.1016/j.jss.2023.111677.
[51] J. A. Hurtado Alegría, M. C. Bastarrica, and A. Bergel, "Avispa: A tool for analyzing software process models," *Journal of Software: Evolution and Process*, vol. 26, no. 4, pp. 434–450, 2014.
[52] Eclipse Foundation, "Eclipse project archived downloads." [Online]. Available: https://archive.eclipse.org/eclipse/downloads/. [Accessed: Jan. 2024].
[53] Eclipse Foundation, "Eclipse IDE for Java Developers." [Online]. Available: https://www.eclipse.org/downloads/packages/. [Accessed: Jan. 2024].
[54] V. Lenarduzzi, A. Sillitti, and D. Taibi, "Analyzing forty years of software maintenance models," in *Proc. IEEE/ACM Int. Conf. Software Engineering Companion (ICSE-C)*, 2017, pp. 146-148.
[55] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction," in *Proc. SEDA*, 2020, pp. 165-175.
[56] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed?" in *Proc. IEEE Int. Conf. Program Comprehension (ICPC)*, 2019, pp. 209-219.
[57] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, pp. 1419-1457, 2020.
[58] L. Lavazza, D. Tosi, and S. Morasca, "An empirical study on the persistence of SpotBugs issues in open-source software evolution," in *Proc. Int. Conf. Quality of Information and Communications Technology (QUATIC)*, 2020, pp. 144-151.
[59] J. Businge, S. Kawuma, E. Bainomugisha, F. Khomh, and E. Nabaasa, "Code authorship and fault-proneness of open-source Android applications," in *Proc. Int. Conf. Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2017, pp. 33-42.
[60] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are SonarQube rules inducing bugs?" in *Proc. 27th IEEE Int. Conf. Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 501-511, https://doi.org/10.1109/SANER48275.2020.9054821.
[61] SonarSource, "Java rules." [Online]. Available: https://rules.sonarsource.com/java/. [Accessed: Jan. 2024].
[62] S. Kawuma, E. Nabaasa, D. B. Sabiiti, and V. Mabirizi, "An empirical study of bugs in Eclipse stable internal interfaces," *Indian Journal of Software Engineering and Project Management*, vol. 2, no. 2, pp. 4–10, 2022, https://doi.org/10.54105/ijsepm.D9015.072222.

[63] S. Kawuma, D. S. Bamutura, A. Obbo, V. Mabirizi, M. Kabarungi, and E. Nabaasa, "Eclipse application programming interfaces: How buggy are they?" *VFAST Transactions on Software Engineering*, vol. 13, no. 2, pp. 228–244, 2025, https://doi.org/10.21015/vtse.v13i2.2134