Journal of Software Engineering, Information and Communication Technology (SEICT)

Journal homepage: https://ejournal.upi.edu/index.php/SEICT

An Empirical Analysis of Code Smell in Eclipse Framework Ecosystem

Simon Kawuma[1], Enock Mabberi[2], David Sabiiti Bamutura[3], Moreen Kabarungi[4],*

Dickson Kalungi[5], Evarist Nabaasa[6]

[1-2]Software Engineering Department, Mbarara University Science and of Technology
[3,6]Computer Science Department, Mbarara University Science and of Technology
[4,5]Information Technology Department, Mbarara University Science and of Technology
Correspondence E-mail: simon.kawuma@must.ac.ug

ABSTRACT

Eclipse Framework developers claim that public APIs are supported whereas internal APIs are unsupported. However, there is no guarantee that these interfaces are well-tested because several code smells are reported by interface users. Applications that use code-smelly interfaces risk failing if the code-smell are not fixed. Previous research revealed that not all code smells can be resolved and fixed within a short period. Thus, interface users have to fix the code smells themselves or abandon that particular interface. To avoid waiting indefinitely for solutions from interface developers or getting involved in code smell fixing, users should use code-smell-free interfaces. However, interface users may not be aware of the existence of code smell-free interfaces in the Eclipse framework. In this research study, we used SonarQube tool to carry out an empirical investigation on 28 major Eclipse releases to establish the existence of code-smell- free interfaces. We provide a data set of 218K and 321K code-smell-free public APIs and internal APIs classes respectively. Also, we discovered that on average, 36.1% and 57.2% of the total interfaces in a given Eclipse release are code smell-free public APIs and internal APIs respectively in all the studied Eclipse releases. Furthermore, we have discovered that the number of code smells linearly increases as the Eclipse framework evolves. The average number of code smell and technical Debt is 147K and 2,744 days in all the studied Eclipse releases. Results from this study can be used by both interface providers and users as a starting point to know tested interfaces and also estimate efforts needed to fix code smells in Eclipse Frameworks.

ARTICLE INFO

Article History:
Submitted/Received
20 Januari 2025
First Revised 14 February 2025
Accepted 07 March 2025
First Available online
01 May 2021
Publication Date 01 June 2025

Keyword:
Code smells;
Evolution;
Internal APIs;
Public APIs;
Software quality.

1. INTRODUCTION

Frameworks and libraries serve as the foundation for application development (T. Tourwé and T. Mens, 2003). This approach in application development encourages functionality reuse (D. Konstantopoulos et al., 2009) and boosts productivity (J. Businge, 2019). This is why major application frameworks, including Eclipse, jBPM, JDK and JUnit, frequently provide public (stable) interfaces (APIs) to developers. In addition to public APIs, all of 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 main 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.

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 (J. Businge, 2019). Despite the fact that internal APIs are discouraged, they are widely used. Businge et al. discovered that around 44% of the 512 Eclipse plug-ins employ internal APIs (J. Businge et al., 2015). (A. Hora, 2016) found that 23.5% of 9702 Eclipse client projects stored on GitHub relied on internal APIs. Experienced application developers stated that leveraging internal APIs is a better option than creating their own APIs from scratch (J. Businge, A et al., 2013).

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 (D. Johannes et al, 2019). Code smells reveal something wrong with the underlying code of the application which can lead to the eventual failure of an application or slows its performance. They include duplicate codes, long methods, comments, long parameter lists, unnecessary primitive variables dead code and data clumps etc. (A. Gupta et al, 2017). These can affect the speed of activity and may also become a detriment to an application program.

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 (A. Gupta et al, 2017), (R. S. Menshawy, 2021). Prior to refactoring, the initial process includes identifying code fragments that violate the semantic properties or structure for instance the complexity or coupling (U. Mansoor, 2017), 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. According to Mansoor, Usman, et al, refactoring is really challenging. According to reports, software maintainers spend at least 60% of their time comprehending the code, and maintenance accounts for around 80% of software costs (U. Mansoor, 2017), (E. Doğan et al., 2022). Therefore, fixing and resolving code smells is characterized by a long period hence the interface user might wait indefinitely for a solution from the developer.

Furthermore, software with code smells actually work and therefore can give an output, but it is characterized by slowed processing (A. Tahir et al., 2018). This increases the risk of failure as well as making the program vulnerable to bugs which in turn contributes to poor code quality and hence increases the technical debt. This implies that interface users are left with no choice but to fix the code smells themselves or abandon the interfaces. Reusing source code with code smells could lead to high maintenance time and costs. Literature provides a long list of code smell, and developers are advised to avoid code smell when developing new and reusing existing code, as it increases the effort and cost of identifying and refactoring code after system development (M. M. Rahman et al., 2022). However, remembering this long list is difficult especially for new developers. 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.

In addition, interface users manually search for the functionality they require in the Eclipse Framework (J. Businge., 2013). Because Eclipse is a vast and complex software framework, it is possible that interface users will first encounter code smelly interfaces rather than code smell-free interfaces when looking for functionality to use in their applications. In this study, we used SonarQube static code analyser tool to investigate 28 Eclipse Framework releases, with the purpose of determining the presence of code smell-free interfaces as the Eclipse framework evolves. The study's goal is to recommend the code smell-free interfaces to application developers. We developed five research questions namely:

1) RQ1: what is the number code smells in Eclipse Frameworks?

2) RQ2: What is the Technical Debt needed to fix code smell in Eclipse Frameworks?

3) RQ3: What is the percentage of code smell-free internal interfaces in Eclipse Framework?

4) RQ4: What is the percentage of code smell-free public interfaces in Eclipse Framework?

5) RQ5: What is the commonest code smell in Eclipse Frameworks?

To address these research questions, we used static code analysis tools called SonarQube (S. Kawuma and Nabaasa, 2018) to extract information about code smell. In summary, the contributions of this work are threefold:

1) We provide a dataset of 218K and 321K code smell-free public APIs and internal APIs classes respectively to Eclipse interface providers and users. Providers can use this dataset to estimate the efforts 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 rest of the paper is organized as follows: Section 2 presents related work, whereas Section 3 presents Research Methodology and Section 4 discusses experimental results. Finally, Section 5 concludes the paper and suggests some areas for future research.

2. METHODS

One of the conclusions of prior studies by Businge et al. was that interface users are always utilizing unstable interfaces, and the reason for this is because there are no alternative stable interfaces that provide the same functionality (J. Businge et al., 2013). Indeed, Kawuma et al. demonstrated that less than 1% of APIs provide the same or similar functionality as non-APIs (S. Kawuma et al., 2016). In a recent study, Businge et al (J. Businge et al., 2013) 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., 2016) revealed that 7% of 2,277 of internal interfaces were promoted to public interfaces and the promotion rate was too low. (S. Kawuma and Nabaasa, 2018) confirmed that the rate at which internal APIs are promoted to public APIs is slow.

(L. Guerrouj et al., 2017) 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.

(B. Seref and Tanriover., 2016) 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 it is one of the most important attributes. (Jafari et al., 2021) 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, quantify and understand dependency smells and conducted a series of surveys with practitioners who identified and quantified seven dependency smells with varying degree of popularity. They also discovered that two or more distinct smells appear in 80 percent of JavaScript project and that dependency smell cause security threats, runtime error and dependency breakage.

Previous survey studies have examined the issues of code smells, including definitions, detection methodologies, and refactoring tools (G. Lacerda et al., 2020),(M. Agnihotri et al., 2020), and (H. M. dos Santos et al., 2019). Several research (P. Meananeatra, 2012), (D. Taibi et al., 2017), (A. Yamashita and L. Moonen, 2013), and (D. J. Kim, 2020) examined developers' awareness of code smells, perceptions, and motivations for deleting them. While other studies (S. Jain and A. Saha, 2019), (B. F. Békefi et al., 2019) and (S. Vidal et al., 2019) focused on refactoring activities and challenges. Several research (Yousef, and A. Salem, 2021), (A. AbuHassan et al., 2021) and (A. Kaur et al., 2019) examined the methods and algorithms utilized in detection and refactoring tools. Furthermore, (M. Tufano et al., 2015) studied the evolution of code smells in 200 open-source Java systems namely; Android, Apache, and Eclipse ecosystems and discovered that code smells are introduced in the code by both new and experienced engineers at the start of the project. Whereas (D. Johannes et al., 2019) examined the effect of code smells on the fault-proneness of JavaScript server-side projects. Although some of the above studies look at public APIs and internal interface by identifying and recommending internal Interface for promotion, whereas other studies look at code smell definition, detection tools, evolution, fault-proneness, management, none of the above authors studied code smell composition, distribution, remediation efforts (Technical Debt) to fix them and also existence of code smell-free interfaces in Eclipse Framework as compared to our study. This section describes the experimental setup used to collect data for the study questions.

3.1 Eclipse Releases Collection

Table 1. Eclipse major releases and their corresponding release dates

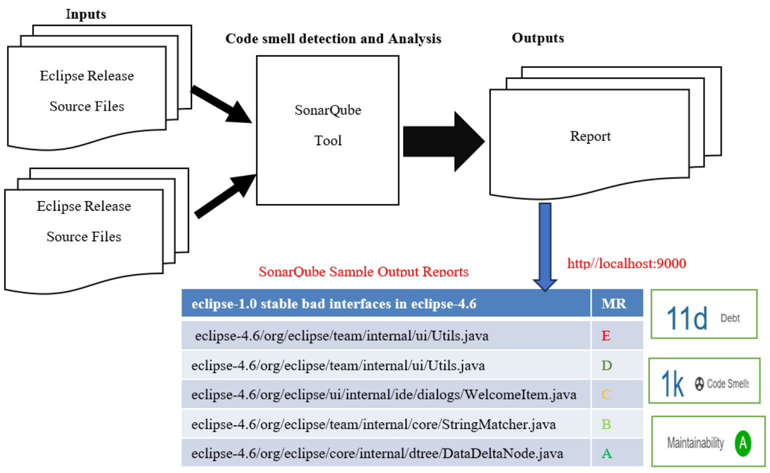
Major Releases	Release Date	Java LOC	Java Classes	Major Release	Release Date	Java LOC	Java. Classes
E-1.0	07-Nov-01	449K	4,608	E-4.2	27-Jun-12	2.8M	22,443
E-2.0	27-Jun-02	769K	6,751	E-4.3	05-Jun-13	2.9M	22,798
E-2.1	27-Mar-03	959K	7,911	E-4.4	06-Jun-14	3.1M	23,880
E-3.0	25-Jun-04	1.3M	10,634	E-4.5	03-Jun-15	3.14M	23,920
E-3.1	27-Jun-05	1.6M	12,299	E-4.6	06-Jun-16	3.2M	23,936
E-3.2	29-Jun-06	2M	14,941	E-4.7	28-Jun-17	3.3M	25900
E-3.3	25-Jun-07	2.1M	16,036	E-4.8	27-Jun-18	3.39M	26,180
E-3.4	17-Jun-08	2.5M	18,800	E-4.9	19-Sept-18	3.4M	26,363

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. **Table 1** below present the different Eclipse major releases we considered in this research. The first column shows the major releases while the second column shows their corresponding release date. The third column shows the java Lines of code (LOC) in each major Eclipse release while the fourth column shows the total number of java classes in a given 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.

3.2 Code smell Collection and Extraction Using SonarQube Tool.

In this section, we present how we extracted data for research questions RQ1-RQ5. We used the SonarQube tool (version 8.2) 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 (A. Sillitti, and D. Taibi, 2017), (V. Lenarduzzi et al., 2017) and in industry (C. Vassallo et al., 2020), (L. Lavazza et al., 2020). We configured and ran SonarQube on a local computer. We used the 432 maintainability rules that cover code smell detection in SonarQube. When any of the rules are violated, then that particular source code manifests as a code smells. We investigated the total number of code smells, the code smell remediation effort to fix all code smells and also collected information about the most dominant code smell type i.e. the most violated rule for every Eclipse major release. In addition to code smell detection, SonarQube estimates the code smell remediation effort in days and an 8-hour day is assumed.

Figure 1 illustrates the procedure we followed to detect and extract code smell information in all the analyzed 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 on the SonarQube server via the following URL **http://localhost:9000**. A Sample output reports for Eclipse-4.16 obtained from SonarQube is shown in figure 1 below. For SonarQube, 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, from figure 1, the last rows has file with MR of A i.e. it has no code smell. The tool counts the number of code smells reported in each release and the Technical Debt as shown in the report.


Inputs
Eclipse Release Source Files
Eclipse Release Source Files
Code smell detection and Analysis
SonarQube Tool
Outputs
Report
http//localhost:9000
SonarQube Sample Output Reports
11d Debt
1k Code Smells
Maintainability A

eclipse-1.0 stable bad interfaces in eclipse-4.6	MR
eclipse-4.6/org/eclipse/team/internal/ui/Utils.java	E
eclipse-4.6/org/eclipse/team/internal/ui/Utils.java	D
eclipse-4.6/org/eclipse/ui/internal/ide/dialogs/WelcomeItem.java	C
eclipse-4.6/org/eclipse/team/internal/core/StringMatcher.java	B
eclipse-4.6/org/eclipse/core/internal/dtree/DataDeltaNode.java	A

Figure 1. SonarQube Code Smell Detection Tool

3.3 Data extraction for Number of code smell 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 code smells in Eclipse Frameworks***? and ***RQ2: What is the Technical Debt needed to fix code smell in Eclipse Frameworks?*** Information about the code smells and technical debt i.e. time needs to fixed them was extracted from the SonarQube reports. The total number of smell code and Technical Debt is provided in reports as illustrated in **figure 1**. For example, from figure 1,1K code smells were detected and 11 days of technical debt were needed to fix the identified code smells.

3.4 Data extraction for code smell- Free Interfaces and in Eclipse Releases

We used SonarQube tool to extract information about code smell-free 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** as illustrated in the output report **figure 1**. To determine the percentage of code smell-free public interfaces and internal interface in a given Eclipse release, we counted both the number of classes with and without a substring ***internal*** in their file path for internal interface and public interfaces respectfully. We used equation 1 and 2 to calculate percentage of internal and public code smell-free interfaces respectfully by looking at public APIs and internal APIs individually in a given Eclipse release as shown below;

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

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

To determine the percentage of code smell free public APIs and interfaces APIs as a combination in a given Eclipse release, code smell free public APIs and internal APIs were each counted 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 equations 3 and 4 respectively as shown below;

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

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

3.5 Data extraction for commonest code smell in Eclipse Releases

To address research question ***RQ5: What is the commonest code smell in Eclipse Frameworks?*** We used the SonarQube tool to extract information about code smell maintainability rules in the different Eclipse releases. SonarQube has 432 maintainability rules that are uses to detect code smells and if any of the rules is violated, that particular source code manifests as a code smell. The tool list the violated rule together with the code smell which violated it. To establish the commonest code smell, we count the frequency the maintainability rule is violated.

3. RESULTS AND DISCUSSION

3.1 Number of code smells and Technical Debt in Eclipse Releases

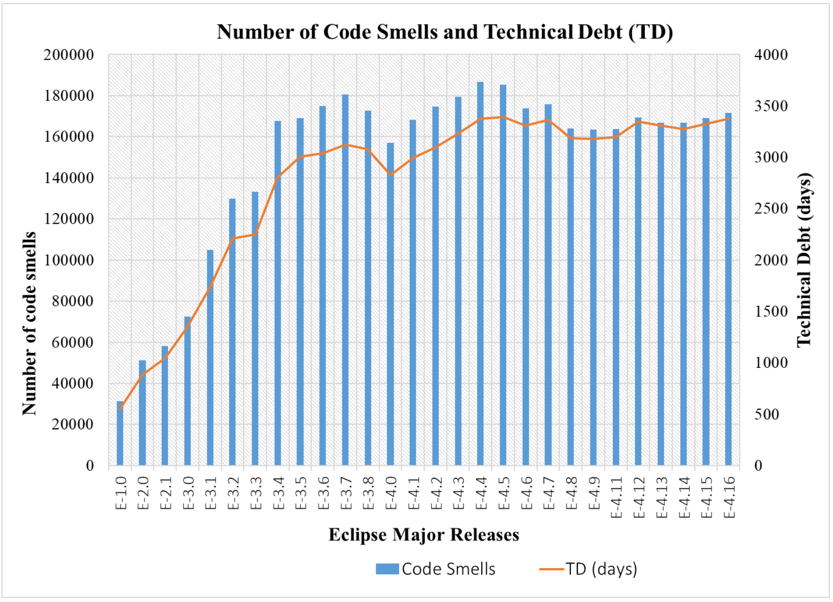


Figure 2. Number of Code smells and Technical Debt in Eclipse Releases

Figure 2 presents the total number of code smells together with the Technical Debt need to fix them in all the different Eclipse releases. In Figure 2 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 both the bar graphs and line graphs, we see a linear increase in the number of code smells and Technical Debt. Furthermore, there is a decline in the code smells and Technical Debt between Eclipse-3.8 and Eclipse-4.0 and thereafter a linear 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 smell detected is between 31,199 to 186,590 across the studied Eclipse releases. Whereas the minimum and maximum efforts need to fix the identified code smells ranges between 545 to 3,391 days in all analyzed Eclipse releases. The average number of code smell 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

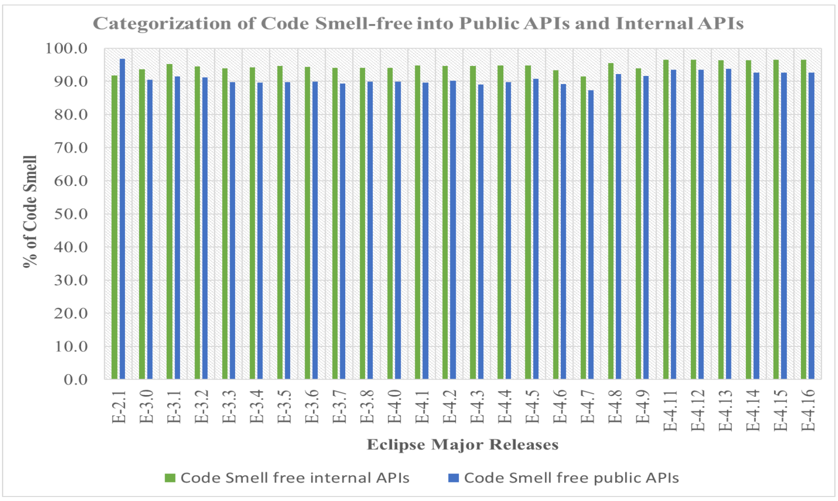


Figure 3. Code Smell-free Public APIs and Internal APIs Categories.

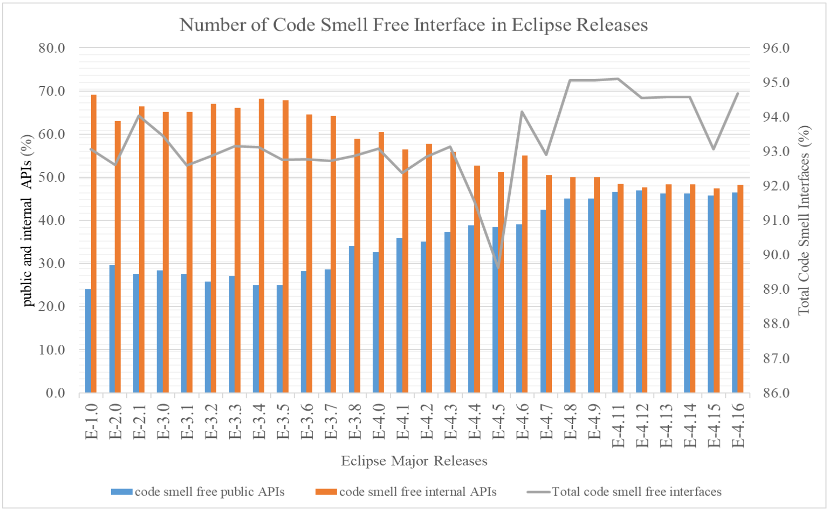


Figure 4. Code Smell-free Interfaces in Eclipse Releases.

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 which are public APIs and internal APIs respectively. In the same figure, we observe that there exist over 87.3% and 91.5% code smell-free public APIs and internal 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 bar graph, we see that majority of code smell-free classes are internal APIs compared to public APIs. This is because internal APIs are twice as much as the public APIS during the evolution of Eclipse (S. Kawuma et al., 2016). The percentage of code smell-free public APIs ranges from 24.9%-46.9% whereas that of internal interfaces is between 47.4%-68.2% of the total interfaces respectively in all the analyzed Eclipse releases. On average, 36.1% and 57.2% of the total interfaces in a given 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 a given Eclipse release are code smell-free. This higher percentage from both may imply that Eclipse interfaces are generally tested for code smell. Furthermore, since internal APIs are considered to be immature, and unsupported (J. Businge et al., 2019) during framework evolution, one would expect to see almost all internal APIs classes with code smells. However, from our investigation, we have discovered that on average 57.2% of total number of classes have zero code smells for all the studied 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. This research considered 432 maintainability rules provided in the SonarQube tool to detect code smells in Eclipse releases. The complete list of Maintainability rules is available online. Maintainability rules create code violations that represent something wrong in the code which will be reflected as a code smell. Tables 2, present result of the 25 commonest code smells that arise as a result of the violation of the maintainability rules. 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 as a consequence of violating a given maintainability rule in all the analyzed Eclipse releases. A detailed list of all code smell found in Eclipse releases can be found on GitHub using the URL provide in the data availability section.

4. CONCLUSION

In this research paper, we used SonarQube to study code smell trends in the Eclipse framework. We focused on establishing if there exist code smell-free interfaces which can be recommend to the developers. We chose SonarQube because it is an open-source tool thus available for use and can detect code smells early enough during development. We have discovered that on average, there exist over 36.1% and 57.2% of the total interfaces in a given Eclipse release are code smell-free public APIs and internal APIs respectively. 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. The average number of code smell and technical Debt is 147,277 and 2,744 days in all the studied Eclipse releases. Furthermore, we observed a linear increase of code smells across all the analyzed 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 line of code (LOC) increases. 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. Furthermore, the total number of code smells discovered would give an insight on how much time and effort is needed by both the framework developer and interface users to remove code smells. In Table 2, we listed the commonest code smells reported by SonarQube in all the analyzed Eclipse releases. This finding is interesting because it provides information to interfaces providers and users about the common code smell and thus, they should adhere to good coding principles to avoid code smells in their applications.

In a follow-up study, we intend to investigate the popularity of the identified code smell-free interfaces by looking at both internal and external usage. Internal interface utilization can be determined by examining how many packages and libraries in the Eclipse framework use the discovered code smell-free interfaces. The number of applications on GitHub that have code smell-free interfaces can be used to determine external usage. Similarly, external utilization can be calculated by counting the number of developers who have used or touched a specific code smell-free interface.

5. ACKNOWLEDGMENT

The authors would like to thank and acknowledge staff in computing services department at Mbarara University of Science and Technology for providing space on their serve where the study experiment was carried out.

6. REFERENCES

T. Tourwé and T. Mens, "Automated support for framework-based software," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 2003, pp. 148-157: IEEE.

D. Konstantopoulos, J. Marien, M. Pinkerton, and E. Braude, "Best principles in the design of shared software," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, 2009, vol. 2, pp. 287-292: IEEE.

J. Businge, S. Kawuma, M. Openja, E. Bainomugisha, and A. Serebrenik, "How stable are eclipse application framework internal interfaces?," in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*, 2019, pp. 117-127: IEEE.

J. Businge, A. Serebrenik, and M. G. J. S. Q. J. Van Den Brand, "Eclipse API usage: the good and the bad," vol. 23, pp. 107-141, 2015.

A. Hora, M. T. Valente, R. Robbes, and N. Anquetil, "When should internal interfaces be promoted to public?," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 278-289.

J. Businge, A. Serebrenik, and M. van den Brand, "Analyzing the Eclipse API usage: Putting the developer in the loop," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 37-46: IEEE.

D. Johannes, F. Khomh, and G. J. S. Q. J. Antoniol, "A large-scale empirical study of code smells in JavaScript projects," vol. 27, pp. 1271-1314, 2019.

A. Gupta, B. Suri, and S. Misra, "A systematic literature review: code bad smells in java source code," in *Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part V 17*, 2017, pp. 665-682: Springer.

R. S. Menshawy, A. H. Yousef, and A. Salem, "Code smells and detection techniques: a survey," in *2021 international mobile, intelligent, and ubiquitous computing conference (MIUCC)*, 2021, pp. 78-83: IEEE.

U. Mansoor, M. Kessentini, B. R. Maxim, and K. J. S. Q. J. Deb, "Multi-objective code-smells detection using good and bad design examples," vol. 25, pp. 529-552, 2017.

E. Doğan, E. J. I. Tüzün, and S. Technology, "Towards a taxonomy of code review smells," vol. 142, p. 106737, 2022.

A. Tahir, A. Yamashita, S. Licorish, J. Dietrich, and S. Counsell, "Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, 2018, pp. 68-78.

M. M. Rahman, A. Satter, M. M. A. Joarder, and K. Sakib, "An Empirical Study on the Occurrences of Code Smells in Open Source and Industrial Projects," in *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2022, pp. 289-294.

S. Kawuma, J. Businge, and E. Bainomugisha, "Can we find stable alternatives for unstable eclipse interfaces?," in *2016 IEEE 24th international conference on program comprehension (ICPC)*, 2016, pp. 1-10: IEEE.

S. Kawuma and E. J. a. p. a. Nabaasa, "Identification of promoted eclipse unstable interfaces using clone detection technique," 2018.

L. Guerrouj *et al.*, "Investigating the relation between lexical smells and change-and fault-proneness: an empirical study," vol. 25, pp. 641-670, 2017.

B. Seref and O. J. I. J. S. E. A. Tanriover, "Software code maintainability: a literature review," 2016.

A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. J. I. T. o. S. E. Tsantalis, "Dependency smells in javascript projects," vol. 48, no. 10, pp. 3790-3807, 2021.

G. Lacerda, F. Petrillo, M. Pimenta, Y. G. J. J. o. S. Guéhéneuc, and Software, "Code smells and refactoring: A tertiary systematic review of challenges and observations," vol. 167, p. 110610, 2020.

M. Agnihotri and A. J. J. o. I. P. S. Chug, "A systematic literature survey of software metrics, code smells and refactoring techniques," vol. 16, no. 4, pp. 915-934, 2020.

H. M. dos Santos, V. H. Durelli, M. Souza, E. Figueiredo, L. T. da Silva, and R. S. Durelli, "Cleangame: Gamifying the identification of code smells," in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, 2019, pp. 437-446.

P. Meananeatra, "Identifying refactoring sequences for improving software maintainability," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 406-409.

D. Taibi, A. Janes, V. J. I. Lenarduzzi, and S. Technology, "How developers perceive smells in source code: A replicated study," vol. 92, pp. 223-235, 2017.

A. Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," in *2013 20th working conference on reverse engineering (WCRE)*, 2013, pp. 242-251: IEEE.

D. J. Kim, "An empirical study on the evolution of test smell," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 149-151.

S. Jain and A. Saha, "An Empirical Study on Research and Developmental Opportunities in Refactoring Practices," in *SEKE*, 2019, pp. 313-418.

B. F. Békefi, K. Szabados, and A. Kovács, "A case study on the effects and limitations of refactoring," in *2019 IEEE 15th International Scientific Conference on Informatics*, 2019, pp. 000213-000218: IEEE.

S. Vidal, I. Berra, S. Zulliani, C. Marcos, J. A. D. J. A. T. o. S. E. Pace, and Methodology, "Assessing the refactoring of brain methods," vol. 27, no. 1, pp. 1-43, 2018.

A. AbuHassan, M. Alshayeb, L. J. J. o. S. E. Ghouti, and Process, "Software smell detection techniques: A systematic literature review," vol. 33, no. 3, p. e2320, 2021.

A. Kaur, G. J. H. S. Dhiman, N. I. O. A. Theory, and I. Applications, "A review on search-based tools and techniques to identify bad code smells in object-oriented systems," pp. 909-921, 2019.

M. Tufano *et al.*, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, vol. 1, pp. 403-414: IEEE.

V. Lenarduzzi, A. Sillitti, and D. Taibi, "Analyzing forty years of software maintenance models," in *2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C)*, 2017, pp. 146-148: IEEE.

V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction," in *Proceedings of 6th International Conference in Software Engineering for Defence Applications: SEDA 2018 6*, 2020, pp. 165-175: Springer.

D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 209-219: IEEE.

C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. J. E. S. E. Zaidman, "How developers engage with static analysis tools in different contexts," vol. 25, pp. 1419-1457, 2020.

L. Lavazza, D. Tosi, and S. Morasca, "An empirical study on the persistence of spotbugs issues in open-source software evolution," in *International Conference on the Quality of Information and Communications Technology*, 2020, pp. 144-151: Springer.