VFAST Transactions on Software Engineering
https://vfast.org/journals/index.php/VTSE@ 2025, ISSN(e): 2309-3978, ISSN(p): 2411-6246
Volume 13, Number 2,April-June 2025 pp: 228-244

Eclipse Application Programming Interfaces: How Buggy Are They?

Simon Kawuma [1*], **David Sabiiti Bamutura** [2], **Aggrey Obbo** [1], **Vicent Mabirizi** [3], **Moreen Kabarungi** [4], **Evarist Nabaasa** [2]

[1]Department of Software and Informatics Engineering, Mbarara University of Science and Technology, Ugandan; [2]Department of Computer Science, Mbarara University of Science and Technology, Ugandan; [3]Department of Information Technology and Computer Science, Kabale University, Ugandan; [4]Department of Information Technology, Mbarara University of Science and Technology, Ugandan

Keywords: *Eclipse, public APIs, internal APIs, Bugs, Software Quality, Evolution.*

Journal Info:
Submitted:
April 29, 2025
Accepted:
June 25, 2025
Published:
June 30, 2025

Abstract

Eclipse Framework provides stable public APIs and unstable internal APIs. However, there is no guarantee that these interfaces are well tested because several bugs are reported by interface users on Bugzilla-based Eclipse project. Applications that use buggy APIs risk failing if bugs are not fixed. Bug fixation and resolution takes at least 3 years thus API users have to fix the bugs themselves or abandon that particular API. The study aimed at identifying bug free interfaces in the Eclipse Framework and recommend them to application developers. In this research study, we used both SonarQube and SpotBugs static analysis tools to carry out an empirical investigation on 28 major Eclipse releases to establish the existence of bug free interfaces. We provide a dataset of 218K and 303K bug-free public API and internal API respectively. There exist over $85.9%$ and $88.2%$ bug-free public APIs and internal APIs, respectively, in Eclipse releases. Furthermore, over 80.8% and 44.2% are major and Malicious code vulnerability bugs respectively and the average bug remediation effort is 105 days. 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 bugs and an online dataset of bug-free interface is available on Github for developer.

***Correspondence author email address:** simon.kawuma@must.ac.ug
DOI: 10.21015/vtse.v13i2.2134

1 Introduction

Application developers build their systems on top of frameworks and libraries. Building applications in this way fosters reuse of functionality and increases productivity [1, 2]. This is why large application frameworks such as Eclipse [3] MSDN [4], jBPM [5], JUnit [6] commonly provide public (stable) interfaces (APIs) to application developers. In addition to public APIs all these frameworks also provide internal APIs. One widely used and adopted application framework

This work is licensed under a Creative Commons Attribution 3.0 License.

is the Eclipse application framework. Eclipse is a large and complex open source software system used by thousands of application developers. Eclipse has been evolving for over two decades producing over 28 major and 55 minor releases. Eclipse, jBPM, jUnit, all adopt the convention of internal interfaces by using sub-string `internal` in their package names while JDK's internal APIs packages start with the substring `sun`.

Framework developers encourage the use of public APIs because they are considered to be stable, mature and supported however they discourage the use of internal APIs because they may be unstable, unsupported, immature and subject to change or removal without notice [3, 4, 6, 7]. Despite the internal APIs being discouraged, usage of internal APIs is not uncommon. Businge et al. [8] observed that about 44% of 512 Eclipse plug-ins use internal APIs. A similar study by Hora et al. [9] discovered 23.5% of 9,702 Eclipse client projects stored on GitHub depended on internal APIs. Experienced application developers revealed that using internal APIs is a better choice compared to writing their own APIs from scratch [10].

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 whereas internal APIs are unsupported, there is no guarantee that these interfaces are well tested because several bugs are reported by interface users on Bugzilla-based Eclipse project[1]. Applications that might use buggy interfaces risk failing if the bugs are not fixed. This implies that the application developer must be ready to fix the bugs themselves. Bug fixing can also be done by the framework developers on behalf of the user following a typical process of bug fixing which includes 1) a user files a bug report; 2) the bug is assigned to a developer; 3) the developer fixes the bug; 4) changed code is reviewed and verified; and 5) the bug is resolved. However, after bug assignment, some developers tend to take a long time to fix and resolve the bug [11] and thus the interface user might wait indefinitely for a solution from the developer.

The most influential factors in bug fixing and resolution time were bug location, bug reporting time, source code involved in the fix, code changes that are required to fix the bug and the severity of a bug [11, 12]. Zeinab et al. [13] found that not all bugs can be resolved and fixed before the Eclipse release deadline. He further discovered that the bug resolution rate is decreasing over time with most bugs lingering for more than 3 years before getting their final resolution. This implies that interface users are left with no choice but to fix the bugs themselves or abandon the interfaces. As a solution to avoid waiting indefinitely for solutions from interface developers or getting involved in bug fixing, users should use bug free interfaces but unfortunately these users may not be aware of the existence of bug free interfaces in the Eclipse framework.

In addition, interface users manually search for the functionality they require from the Eclipse Framework [10]. Eclipse being a large and complex software framework, it is possible that users may land on buggy interfaces first instead of the bug-free interfaces while searching for functionality to use in their application. In this study, using two static code analysis tools, namely SonarQube [14] and SpotBugs [15, 16] (formerly known as FindBugs [17]) we carried out an investigation on 28 major Eclipse Frameworks releases with a goal to establish the: 1) composition of bugs in 28 major eclipse releases and 2) existence bug-free interfaces during the evolution of the Eclipse framework. For the first goal both tools were used but only SonarQube produced results showing bug free interfaces. The study aims to recommend the identified bug-free interfaces to application developers. With this in mind, we formulated the following research questions to guide the study as follows:

RQ1: What is the composition of bugs in Eclipse Framework Releases? Rationale: to improve the quality of the Eclipse framework and its interfaces, it is vital to know the total bugs, bug distribution, bug severity because this will enable both the framework developer and interface users to know and estimate the amount effort needed to fix the bugs in a given Eclipse release.

RQ2: Can we find bug-free Eclipse Framework

[1] https://bugs.eclipse.org/bugs/

Interfaces? Rationale: Bug fixation and resolution takes a long period of time thus Eclipse interfaces users might wait indefinitely for a solution from Eclipse framework developers. In this research study, we carried out an empirical investigation to establish the existence of bug-free interfaces in the Eclipse framework. We aimed at creating awareness of the bug-free interfaces amongst developers because they can unknowingly use buggy interface.

To address these research questions, we used static code analysis tools: SonarQube[14] and SpotBugs[15] to extract information about bugs. A detailed description of the research protocol used for this study can be found in section 4.

In summary, contributions of this work are three-fold:

1. We provide a dataset of 217,876 and 302,690 bug-free public API and internal API classes respectively that can be used by both Eclipse interface providers and user. Providers can use this dataset to estimate the efforts need to remove bugs. Users can look up for bug-free interfaces they want to use when developing their application.
2. The Eclipse Interface providers claim that public APIs are good and stable interfaces [3]. Indeed this research study has empirically confirmed that public APIs are good since we have discovered that over 85.9% of public APIs are bug-free in all studied Eclipse releases.
3. Interfaces Providers discourage the use of internal APIs because they may be immature and unsupported [3, 4, 6]. However, this research study has empirically confirmed that not all internal APIs are bad since over 88.2% of the internal APIs were bug-free in all studied Eclipse releases and thus users can use them when developing their applications.

The remainder of this paper is organized as follows: Section 2 presents the background information on Eclipse interfaces, bugs, SonarQub and Spotbug tools used in this study. while Section 3 provides an overview of the related work. Section 4 discusses the Research Methodology. Section 5 discusses the results and findings of our study. Section 6 presents discusion of the study. Section 7 presents threats to the validity of our study, Finally, Section 8 concludes the paper and outlines some avenues for future work.

2 Background

This section introduces the necessary background related to types of interfaces in Eclipse framework, bugs and static analysis tools used to investigate bugs within the framework across its evolution.

2.1 Eclipse internal APIs

These are internal implementation artifacts that according to Eclipse naming convention [3] are found in packages with the substring `internal` in the fully qualified name. These internal implementation artifacts include public Java classes or interfaces, or public, protected methods or fields in such a class or interface. Usage of internal APIs is strongly discouraged since they may be unstable [18]. Eclipse clearly states that clients who think they must use these non-APIs do it at their own risk as internal APIs are subject to arbitrary change or removal without notice. Eclipse does not usually provide documentation and support to these internal APIs.

2.2 Eclipse Public APIs

These are public Java classes or interfaces that can be found in packages that do not contain the segment `internal` in the fully qualified package name, a public or protected method or field in such a class or interface. Eclipse states that, the public APIs are considered to be stable and therefore can be used by any application developer without any risk. Furthermore, Eclipse also provides documentation and support for these public APIs.

2.3 Bugs

Bugs can be termed as the errors, flaws, and faults present in the computer program that impact the performance as well as the functionality of the software that can cause it to deliver incorrect and unexpected results. These not only impact the performance of the software, but also cause it to behave in an unanticipated way [19]. Avoidance of bugs during

development and their fixing is a very important part of software development.

2.4 Static Analysis Tools Used

In this research study, we used two static analysis tools namely SonarQube and SpotBugs described briefly below.

2.4.1 SonarQube

We used SonarQube which is one of the most common Open Source static code analysis tools adopted both in academia [20] and in industry [21]. SonarQube is provided as a service platform[2] or it can be downloaded and executed on a private server. SonarQube calculates several metrics such as the number of lines of code and the code complexity, and verifies the code's compliance against a specific set of "coding rules" defined for most common development languages [22]. In case the analyzed source code violates a coding rule or if a metric is outside a predefined threshold, SonarQube generates an "issue". SonarQube includes Reliability, Maintainability and Security rules. SonarQube has separate sets of rules for the most common development languages such as Java, Python, C++, and JavaScript. In this research, we used SonarQube version 8.2 which has more than 500 rules for Java. The complete list of rules is available online[3]. Reliability rules, also named "bugs" create issues (code violations) that "represents something wrong in the code" and that will soon be reflected in a bug. The severity of the bugs can be categorized by their possible impact, either on the system or on the developer's productivity. SonarQube categorizes the severity of identified bugs into five types namely blocker, critical, major, minor and info [14] as explained below;

- **Blocker:** A bug of this kind might make the whole application unstable in production. For example, calling garbage collector, not closing a socket, memory leak, unclosed JDBC connection etc.
- **Critical:** A bug of this kind might lead to an unexpected behavior in production without impacting the integrity of the whole application. For example, NullPointerException, badly caught exceptions, division by zero as a denominator etc.
- **Major:** A bug of this kind might have a substantial impact on productivity. For example, Null pointers should not be dereferenced, too complex methods, package cycles.
- **Minor:** A bug of this kind might have a potential but minor impact on productivity. For example, finalizer does nothing but call superclass finalizer, lines should not be too long, switch statements should have at least 3 cases.
- **INFO:** This is neither a bug nor a quality flaw but just a finding.

Blocker and critical bugs might impact negatively the system, with blocker bugs having a higher probability compared to critical ones. SonarQube also recommends immediately reviewing blocker and critical issues. Major bugs can highly impact the productivity of a developer, while minor ones have little impact.

2.4.2 SpotBugs

SpotBugs [15, 16] is a static analysis tool that looks for bugs in Java source code. The tool is an open-source software, distributed under the GNU Lesser General Public License. SpotBugs inherits all of the features of its predecessor FindBugs [17] and adds new analyses to check for more than 400 bug patterns. SpotBugs checks for bug patterns such as null pointer dereferencing, infinite recursive loops, bad uses of the Java libraries and deadlocks. SpotBugs tool is available as a standalone program and can be downloaded from [15]. SpotBugs categorizes the detected bugs into nine types as explained below;

- **Bad Practice:** Violations of recommended and essential coding practice. Examples include hash code and equals problems, cloneable idiom, dropped exceptions, serializable problems, and misuse of finalize.
- **Correctness:** Probable bug which is an apparent coding mistake resulting in code that was probably not what the developer intended for example: an apparent infinite loop, super

[2] sonarcloud.io
[3] https://rules.sonarsource.com/java

method is annotated with @OverridingMethodsMustInvokeSuper, but the overriding method isn't calling the super method.

- **Malicious code vulnerability:** Code that is vulnerable to attacks from untrusted code for example: finalizer should be protected not public, method invoked that should only be invoked inside a doPrivileged block.
- **Dodgy code:** Code that is confusing, anomalous, or written in a way that leads itself to errors. Examples include dead local stores, switch fall through, unconfirmed casts, and redundant null check of value known to be null. In previous versions of SpotBugs, this category was known as style.
- **Security:** A use of untrusted input in a way that could create a remotely exploitable security vulnerability for example Servlet reflected cross site scripting vulnerability, HTTP Response splitting vulnerability.
- **Multithreaded correctness:** code flaw issues having to do with threads, locks, and volatiles.
- **Internationalization :** Code flaws having to do with internationalization and locale for example: consider using `Locale` parameterized version of invoked method, reliance on default encoding.
- **Performance:** Code that is not necessarily incorrect but may be inefficient for example: the equals and hashCode methods of URL are blocking, huge string constants are duplicated across multiple class files, maps and sets of URLs can be performance hogs etc.
- **Experimental:** Experimental and not fully vetted bug patterns for example Class too big for analysis i.e. This class is bigger than can be effectively handled, and was not fully analyzed for errors.

Detailed complete list of bug description and examples can be found in [15]

3 Related Work

In this section we discussed how the current work relates to the previous work. The previous studies by Businge et al. [8, 10, 23, 24] were based on empirical analysis of the co-evolution of the Eclipse SDK framework and its third-party plug-ins (ETPs). During the evolution of the framework, the authors studied how the changes in the Eclipse interfaces used by the ETPs, affect compatibility of the ETPs in forthcoming framework releases. The authors only used open-source ETPs in the study and the analysis was based on the source code. One of previous studies by Businge et al. [10] was based on analysis of a survey, where they complement other previous studies by including commercial ETPs and taking into account human aspects. One of the major findings of the previous studies was that interface users are continuously using unstable interfaces and the reason for using these unstable interfaces was because there exist no alternative stable interfaces offering the same functionality. Indeed, Kawuma et al. showed that less than 1% APIs offer the same or similar functionality as non-APIs [2]. The earlier studies by Businge et. al studied both public APIs and internal APIs interfaces but they did not look at finding bug-free interface as compared to our study.

From a recent study Businge et.al [1], using a clone detection tool, the authors looked at the stability of internal interface as the Eclipse framework evolves. We discovered that 327K stable internal Interfaces and we recommended them as possible candidate for promotion. Another study that is directly related to [1] is that of Hora et al. [9], in this study the authors investigated the transition from internal to public interfaces. They carried out their investigation on Eclipse (JDT), JUnit, and Hibernate. Their main aim was to study the transition from internal to public interfaces (i.e., internal interface promotion). They detect internal interface promotion when these two conditions are satisfied: 1) there is at least a file change that removes only one reference to Internal and adds only one reference to Public, and 2) the class names of the references remain the same or have a suffix/prefix added/removed. They discovered that 7% of 2,277 of internal interfaces are promoted to public interfaces. They also found that the promoted interfaces have more clients. They also predicted internal interface promotion with precision between 50%–80%, recall 26%–82%, and AUC 74%–85%. Finally,

by applying their predictor on the last version of the analyzed systems, they automatically detected 382 public interface candidates. Our study and this study both aim at identifying internal interfaces that are candidates of promotion. A similar study by Kawuma et. al[25] discovered that indeed the pace at which non-APIs are promoted to APIs is slow and promotion take long. Although studies in [1], [9] and [25] identified and recommended internal Interface for promotion, none of the above authors studied existence of bug-free interfaces in Eclipse Framework.

Businge et al. [26] studied the relationship between code authorship and fault-proneness of android applications, they investigated whether android applications with few major contributor have more or less faults compared to application with larger number of developer that do minor contributions. They discovered that android applications with higher level of code authorship among contributors experience fewer faults. This research studies focused on the relationship between code authorship and faults in small and large software ecosystems in general, but did not focused on a particular public or internal interfaces as compared to our study.

Latifa et al.[27] carried out a study on projects; ANT, ArgoUML and Hibernate to establish the relationship between lexical smell and software quality as well as their interaction with design smells. They discovered 29 smells out of which 13 were design smell and 16 were lexical smells. in addition, they found out that lexical smells can make clases with design smell more fault-prone and that classes containing design smell only are more fault-prone than classes with lexical smell only. Bavota et al. [28] conducted a study on 5,848 free android app to investigate how the apps user ratings correlate with the fault- and change proneness of the APIs such app relied on. From their study, they discovered that apps having high user ratings use APIs that are less fault- and change-prone than the apps used by low rated apps. Assaduzzaman et al.[29] mined changes and bug reports in Android to identify changes that introduced the bugs. The links between bugs and changes were identified by looking for keywords in commit messages, and by comparing the textual similarity between the reports and the commit messages. None of the above studies focused on internal or public interfaces.

There have been several studies that have presented the role of bug report evolution during reliable fixing estimates [30] or classify the software bugs from open source systems to improve bug localization [31] and characterization of software bugs in open-source cyber physical systems [32] . Many studies attempt to predict which bugs get fixed [33], re-assigned [34]. A few studies also have investigated how bugs are coordinated among software testers, users, developers[35] and how the misclassification of bugs affects the bug prediction task [36]. Fan et al.[37] analyse five different dimensions related to software bugs by looking at bug report texts, reporter's experience, developer-reporter collaborations and classify valid and invalid bug reports using machine learning. All these studies focus on bug reporting, prediction however our study focuses more on creating awareness of the existence of bug-free interfaces in Eclipse frameworks which interfaces users can use without worrying about bugs fixing and reporting.

4 Research Methodology

This section presents the experimental setup of how the data for the research questions was collected.

4.1 Eclipse Releases Collection

In this section, we explain the data sources of our study. Our study is based on 28 Eclipse SDK major releases from Eclipse project Archive website [38, 39]. Table 1 present the different Eclipse major releases we considered in this research. The first column shows the major releases starting from Eclipse–1.0 (E-1.0) until Eclipse–4.16 (E-4.16) 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 considered Eclipse as a subject of study because it is widely used and adopted open source framework and thus it will continue attracting more developers. The Eclipse framework is constantly evolving with new version

released every after 3 months. This creates an opportunity to study bug evolutionary trends as the framework evolves. This research focused on Eclipse major versions because as the framework evolves from one major version to another, new projects, sub-projects, packages, classes, interfaces, fields and methods are either added, changed or deleted from the framework.

4.2 Bug Collection and Extraction

In this section, we present how we extracted data for research questions **RQ1 and RQ2**. We used SonarQube tool (version-8.2) [14] and spotBugs tool[15] to extract information about bugs in the different Eclipse releases. We rely on these tools because they are broadly used by thousands of users in academic research settings [40],[20],[41] and in industry [21],[16]. We configured and ran both tools on a local computer. We used web interface and GUI to analyse results for sonarQube and spotBugs respectively. We considered 125 reliability rules of the total 500 rules for SonarQube because the remainder of the rules do not cover bug detection. For SpotBugs we utilized all 400 rules the tool provides. when any of the rules is violated, then that particular source code manifests as a bug. We investigated the total number of bugs, the bug remediation effort to fix the bugs and also collected information about the most dominant bug type i.e. the most violated rule for every Eclipse major release. In addition to bug detection, SonarQube estimates the bug remediation effort in days and an 8-hour day is assumed [14].

Both tools take source directories containing java files as input to detect possible bugs at specific points in the class. Each tool produces an output report for each Eclipse release. Sample output reports for Eclipse-4.16 obtained from sonarQube and spotBugs are shown in Tables 2 and 3 respectively. For SonarQube, each file in the report has a File Reliability Rating (FRR) assigned by SonarQube depending on the nature and number of bugs found in the class of source files under investigation. For example from Table 2, the last two rows have files with FRR of A i.e. they have no bugs. The tool counts the number of bugs reported in each class. Additionally, the tool further rates the file reliability rating of the class as follows: A–Zero bug, B–at least one minor bug, C–at least one major bug, D–at least one critical bug and E–at least one blocker bug. In this study, we considered the total number of bugs reported in each class for each Eclipse release. We then focused on the class that had zero bugs i.e. with rating A. To determine the percentage of bug-free classes, we express the number of classes with rating A as a fraction of the total number of classes in a given Eclipse release.

Table 2. SonarQube Sample Output Report

interfaces in Eclipse-4.6 Major Release	FRR
org/eclipse/core/resources/Synchronizer.java	E
org/eclipse/team/internal/ui/Utils.java	D
org/eclipse/ui/internal/ide/WelcomeItem.java	C
org/eclipse/team/internal/core/StringMatcher.java	B
org/eclipse/pde/internal/swt/tools/IconExe.java	E
org/eclipse/core/internal/dtree/DataDeltaNode.java	A
org/eclipse/compare/internal/TabFolderLayout.java	A

Table 3. SpotBugs Sample Output Report

Warning Type in Eclipse-4.6 Release	Number
Bad practice	4823
Correctness	808
Experimental	78
Internationalization	990
Malicious code vulnerability	16798
Multithreaded correctness	1369
Performance	3623
Security Warnings	19
Dodgy code	11874
Total	**40382**

5 Results

In this section we present the results and analysis of the extracted data in Section 4 to address RQ1 and RQ2.

5.1 Number of Bugs in Eclipse Releases

Figure 1 and 2 presents total number of bugs in different Eclipse releases detected using sonarQube and spotBugs respectively. This value is computed by adding the number of bugs identified by each tool in an entire Eclipse release as shown in sample

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
E-3.5	11-Jun-09	2.6M	19,169	E-4.11	Mar-19	3.5M	27,448
E-3.6	08-Jun-10	2.7M	20,922	E-4.12	Jun-19	3.51M	27,784
E-3.7	13-Jun-11	2.75M	21,104	E-4.13	Sept-19	3.52M	27,904
E-3.8	27-Jun-12	2.8M	22,477	E-4.14	Dec-19	3.53M	27,976
E-4.0	27-Jul-10	2.6M	20,498	E-4.15	Mar-20	3.55M	28,500
E-4.1	20-Jun-11	2.7M	21,234	E-4.16	Jun-20	3.6M	28,135

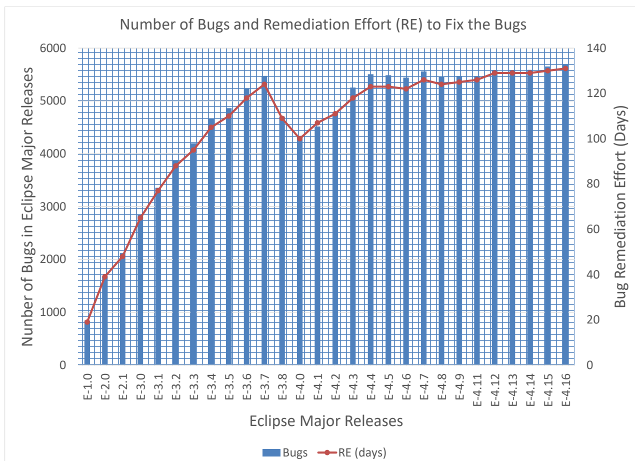

Figure 1. Number of Bugs Detected by Sonarqube.

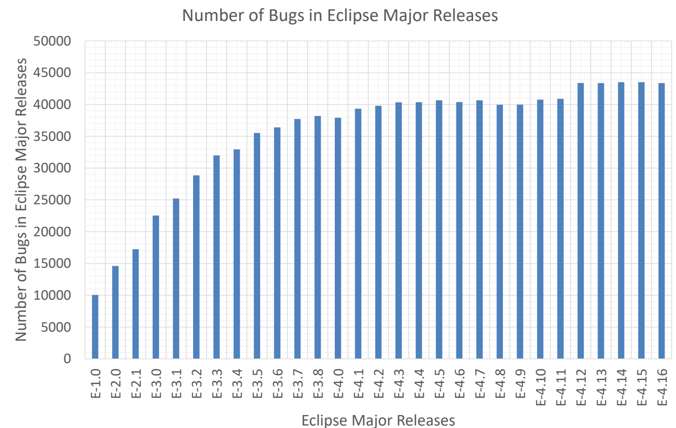

Figure 2. Number of Bugs Detected by SpotBug.

Table 1 and Table 2 for Elipse–4.6 release. In Figure 1 the bar graphs represent the total number of bugs while the line graph represent the remediation effort needed to fix the bugs in Eclipse releases. Focusing on both the bar graphs and line graph, we see a linear increase in the number of bugs and remediation effort in days to fix the bugs. Furthermore, there is a decline in bugs and remediation effort between Eclipse–3.8 and Eclipse–4.0 and there after a linear increased is observed after Eclipse–4.0. The slight change between Eclipse–3.8 and Eclipse–4.0 can be attributed to the fact that 1,979 classes were deleted from Eclipse–3.8 as observed from Table 1 thus more bugs were deleted which further lead to less efforts to fix the bugs. Furthermore, the reason for the observed change is that Eclipse developers assessed the architecture of Eclipse 3.x and discovered that in future, this architecture might be difficult to incorporate new technology, encourage growth of the community and attract new contributors. For this reason, Eclipse developers changed the architecture

to address the identified shortcomings from 4.0 on wards [2]. From figure 1 we observe that between 844 to 5,688 bugs were obtained using sonarQube. Still in the same figure we observe that it requires a developer between 19 to 131 days to fix bugs found in all analysed Eclipse releases. Furthermore, from figure 2, we observe that the minimum and maximum bugs obtained using spotBugs is between 10,054 to 43513 respectively. The difference in the total number of bugs observed in figures 1 and 2 is due to the fact that SpotBugs has over 400 bug detection rules compared to SonarQube with only 125 rules.

5.2 Bug Classification in Eclipse Releases

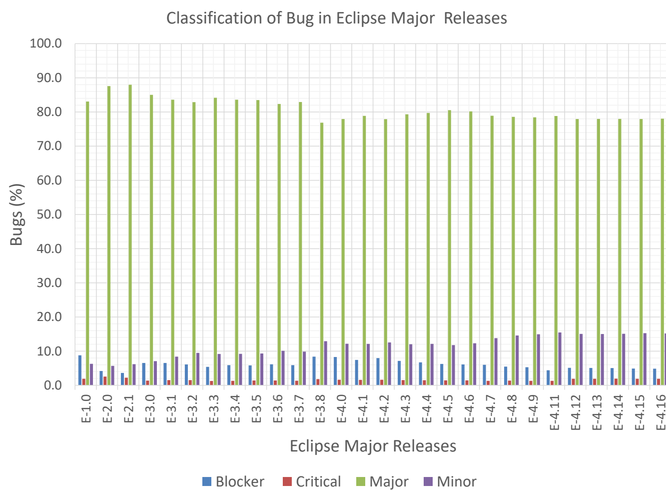

Figure 3. Classification of Bugs as Detected by Sonarqube.

Figures 3 and 4 present results corresponding to percentage of bugs in different categorizes as discussed in section 2.4. In figure 3, for each Eclipse releases there are four bars in 3 and each bar presents the percentage of bug whose bug type is Blocker, Critical, Major and Minor bugs respectively. Looking at the bars in figure 3 we observe that majority of bugs are major bugs, followed by minor bugs, then followed by Blocker bugs and lastly critical bugs. Furthermore, we have observed that on average 80.8%, 11.5%, 6.0% and 1.6% bugs are major, minor, blocker and critical bugs respectively in all the analysed Eclipse releases. In addition, figure 4 percentage of number of bugs detected by spotbugs. We observe that majority of bugs are Malicious code vulnerability(44.20 %),

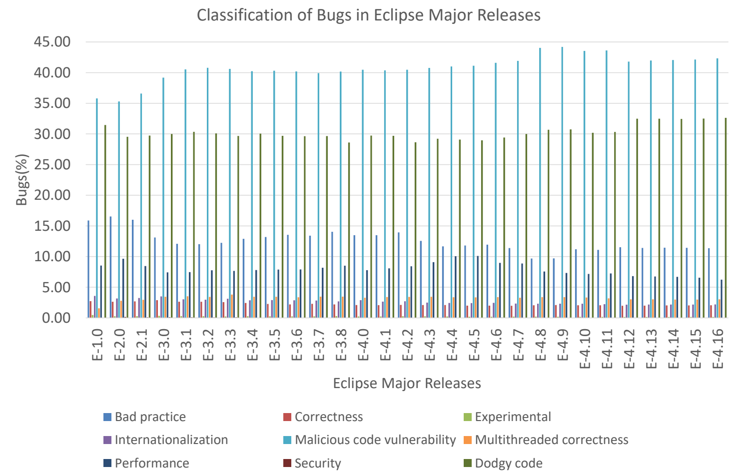

Figure 4. Classification of Bugs as Detected by Spotbug.

Dodgy code(32.62 %), Bad practice (16.54%), Performance(10.10%),Multithreaded correctness (3.81%), Internationalization(3.58%), Correctness(2.89%), Experimental (0.48%) and Security (0.15%)

5.3 Percentage of Bug-Free Interfaces in Eclipse Releases

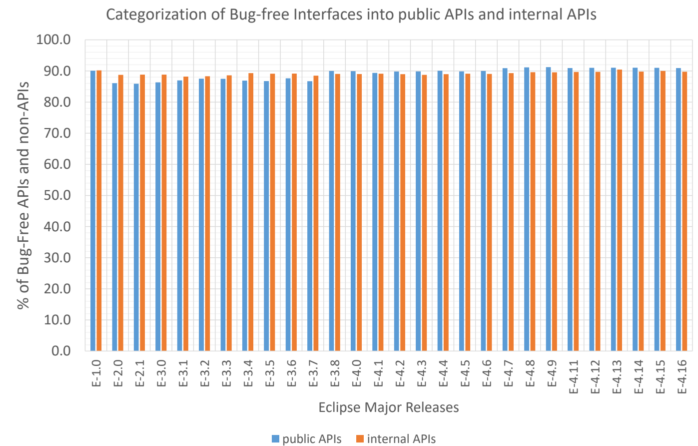

Figure 5. Bug-free Interfaces in public APIs and internal API categories.

Figure 5 and 6 present results corresponding to percentage of bug-free interfaces in different Eclipse major releases. Focusing on figure 5, for each eclipse release, the first and second bars present the percentage of bug-free classes found in public APIs and internal APIs. These percentages are computed by dividing the number of bug-free classes by the total number of

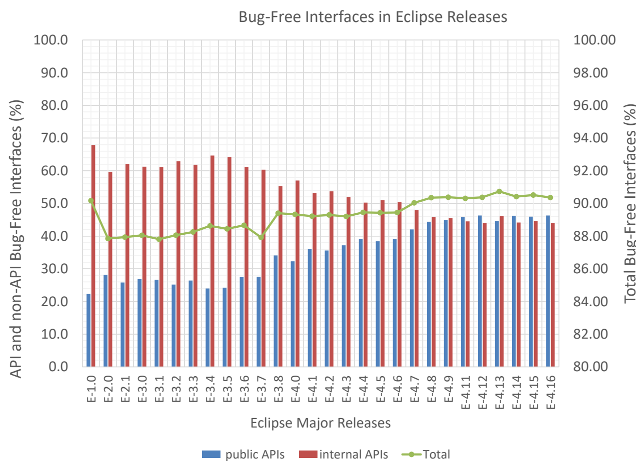

Figure 6. Bug-free Interfaces in Eclipse Releases.

classes in each category (public versus internal) multiplied by 100. In the same figure, we observe that there exist over 85.9% and 88.2% bug-free classes in both public API and internal API categories respectively.

In figure 6, the bar graph presents bug-free public APIs and internal API classes 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 bug-free interfaces (bug-free public API and internal API classes) with respect to the total number of public and internal APIs. From Figure 6 and specifically focusing on bar graph, we see that majority of bug-free classes are internal APIs compared to public APIs. This is because internal API are twice as much as the public API during the evolution of Eclipse[2]. The percentage of bug-free public API classes and internal API classes range from 22.3%-46.3% and 44.1%-67.9% of the total classes respectively in all the analysed Eclipse releases. Furthermore, we found out that on average, 35.1% and 54.2% of the total classes in a given Eclipse release are bug-free public API and internal API classes respectively. Focusing on the line graph in figure 6, we observe that over 87.5% of the total number of classes in a given Eclipse are bug-free. This higher percentage may imply that Eclipse interfaces are generally tested for bugs. Furthermore, Since internal API are considered to be immature, unsupported and subject to change and even can be deleted from the framework [3, 4, 6, 42], during framework evolution, one would expect to see almost all internal APIs classes with bugs. However from our investigation, we have discovered that on average 54.2% of total number of classes have zero bugs for all the studied Eclipse releases.

5.4 Common bugs in Eclipse Releases

In this section, we present results of the common bugs found in Eclipse releases. In this research, we considered 125 reliability rules provided in SonarQube tool (version 8.2) to detect bugs in Eclipse releases. The complete list of Reliability rules is available online[4]. Reliability rules create code violations that represents something wrong in the code which will be reflected as a bug. Tables 4, 5, 6 and 7 presents results of common bugs that arise as a result of violation of standard reliability rules in the SonarQube. The first column in the tables show the unique rule id whereas the second column show a brief description of the rule. The third column (bugs) shows the total number of bugs that are generated as a consequence of violating a given reliability rule in all the analysed Eclipse releases.

Tables 8 and 9 present bugs which were uniquely detected by either sonarqube or spotbugs respectively whereas Table 10 shows bugs which where detected by both tools. Because of limited space, 8,9, 10 only show the first 10 bugs but detailed list can be found on Github.[5]

6 Discussion

from Figure 1 and 2 , we generally observed a linear increase in the number of bugs 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 bugs. In addition, Eclipse has a large community of developers and committers who contribute to its large code base [2]. Furthermore, the total number of bugs discovered would give an insight on how much time and effort is needed by both the framework developer and interface users to remove

[4] https://rules.sonarsource.com/java
[5] https://github.com/simonkawuma/Bug-Free-APIs-in-Eclipse-Releases

Table 4. Common Blocker bugs in Eclipse releases

Rule	Rule Description	bugs
S2095	Resources should be closed e.g. Connections, streams, files must be closed	4,850
S2168	Double-checked locking should not be used	2,313
S2189	Loops should not be infinite	430
S2276	wait(...) should be used instead of Thread.sleep(...) when a lock is held	135
S3046	wait should not be called when multiple locks are held	32

Table 5. Common Critical bugs in Eclipse releases

Rule	Rule Description	bugs
S1143	Jump statements should not occur in finally blocks	1,503
S3518	Zero should not be a possible denominator	401
S2119	Random objects should be reused	64
S2222	Locks should be released	37
S4275	Getters and setters should access the expected fields	20
S2122	ScheduledThreadPoolExecutor should not have 0 core threads	8

Table 6. Top 10 common Minor bugs in Eclipse releases

Rule	Rule Description	bugs
S3077	Non-primitive fields should not be volatile	4,293
S1206	equals(Object obj) and hashCode() should be overridden in pairs	4,237
S2164	Math operands should be cast before assignment	2,207
S2674	The value returned from a stream read should be checked	982
S2183	Ints and longs should not be shifted by zero	957
S2153	Boxing and unboxing should not be immediately reversed	932
S2272	Iterator.next() methods should throw NoSuchElementException	876
S2236	Method parameters and foreach variables' initial values should not be ignored	508
S2097	equals(Object obj) should test argument type,	419
S2676	Neither Math.abs nor negation should be used on numbers	191

bugs. Furthermore, we observed that there exist over 85.9% and 88.2% bug-free public interfaces and internal interfaces respectively as shown in figures 5 and 6 in all the analyzed Eclipse Major releases. This finding implies that majority of the Eclipse interfaces are well tested by their developers before they commit them to be part of the Eclipse framework ecosystem. From figure 4, and Tables 4-7, we observe that most of bugs are *major* and *Malicious code vulnerability* as reported by sonarqube and spotbugs respectively in all the analyzed Eclipse releases. This finding is interesting because it provides information to both interfaces providers and users about the common bugs and thus they should adhere to good coding principles to avoid bugs in their applications.

In this study, we used Sonarqube and Spotbugs static analyze tools to study bug trends in Eclipse framework with a focus of establishing if there exist bug-free interfaces which we can recommend to developer. We choose static analyze tools because they are open-source thus available for use and can detect bugs early enough during development, when they are cheap to fix as revealed by [43]. These tools generates warnings based on well-defined program-

Table 7. Top 10 Common Major bugs in Eclipse releases

Rule	Rule Description	bugs
S2259	Null pointers should not be dereferenced	46,721
S2142	InterruptedException should not be ignored	18,425
S4973	Strings and Boxed types should be compared using equals()	7,072
S2583	Conditionally executed code should be reachable	6,316
S4143	Loops with at most one iteration should be refactored	5,418
S2159	Silly equality checks should not be made	4,089
S3064	Assignment of lazy-initialized members	2,111
S1764	Identical expressions should not be used on both sides of a binary operator	1,714
S5164	ThreadLocal variables should be cleaned up when no longer used	1,631
S3551	Overrides should match their parent class methods in synchronization	1,597

Table 8. Example of bugs Detected by Sonarqube tool only

#	Bug Description
1	InterruptedException should not be ignored
2	Conditionally executed code should be reachable
3	Non-thread-safe fields should not be static
4	ThreadLocal variables should be cleaned up when no longer used
5	BigDecimal(double) should not be used
6	Classes should not be compared by name
7	Non-thread-safe fields should not be static
8	null should not be used with "Optional
9	getClass should not be used for synchronization
10	Double.longBitsToDouble should not be used for int

Table 9. Example of bugs Detected by spotbugs tool only

#	Bug Description
1	Sleep with lock held
2	Questionable Boolean Assignment
3	Ambiguous invocation
4	Bad casts of object references
5	Bad implementation of cloneable idiom
6	Unused field
7	Bad use of return value from method
8	Confused Inheritance
9	Logger problem
10	Confusing method name

ming rules to find bugs [43]. For example Sonarqube and Spotbugs have 125 and 400 rules respectively and once any of these rules is violated, then a bugs will manifest which a developer can fix. However static analyze tools give a large volumes of warnings which becomes a burden to the users when they have to

Table 10. Example of bugs Detected by both Sonarqube and spotbugs tools

#	Bug Description
1	Zero should not be a possible denominator
2	Locks should be released
3	Resources should be closed
4	Loops should not be infinite
5	wait should not be called when multiple locks are held
6	=+ should not be used instead of +=
7	Null pointers should not be dereferenced
8	Strings and Boxed types should be compared using equals()
9	Loops with at most one iteration should be refactored
10	Silly equality checks should not be made
11	Jump statements should not occur in finally blocks
12	notifyAll should be used
13	Blocks should be synchronized on private final fields
14	volatile variables should not be used with compound operators
15	Getters and setters should be synchronized in pairs
16	compareTo should not be overloaded
18	Exception should not be created without being thrown
19	Synchronization should not be based on Strings or boxed primitives
20	iterator should not return this
21	Inappropriate regular expressions should not be used
22	Collection sizes and array length comparisons should make sense
23	Unary prefix operators should not be repeated
24	Non-primitive fields should not be volatile
25	equals(Object obj) and hashCode() should be overridden in pairs
26	Math operands should be cast before assignment
27	Ints and longs should not be shifted by zero or more than their number of bits-1
28	Boxing and unboxing should not be immediately reversed
29	"Iterator.next()" methods should throw "NoSuchElementException"
30	equals(Object obj) should test argument type

analyze the output generated by the these tools but Sonarqube and spotbugs tools have user-friendly and interactive interfaces which the developers can use to carry out analysis of the bug report.

7 Threats to Validity

As any other empirical study, our analysis may have been affected by validity threats. We categorize the possible threats into construct, internal and external validity.

Construct validity focuses on how accurately the metrics utilized measure the phenomena of interest. The methodology used to measure the the percentage of bug-free interfaces in Eclipse is subjected to construct validity. The reason being that we only use classes in our computations yet there are other objects we ignore, for example, methods, variable declarations etc.

There is an *internal validity* threat related to the tools used to extract the data used in our experiments. It is possible that results could differ if a different tool was used. Like any other static analysis tool,

the SonarQube and spotbugs tools we used don't not have a 100% precision. Furthermore, as earlier mentioned, sonarqube and spotbugs detect bugs depending on well-defined programming rules or well known patterns however this might not be the case for all bugs because some bugs can only manifest themselves at run-time or compilation. In addition, both tools can not detect Project-Specific Bug patterns (PSBPs) or sibling bugs and bugs whose detection is based on bug signatures by analysing bug fixes [44].

External Validity is related to the possibility to generalize our results. We focused on the analysis of widely adopted and large-scale framework. Therefore Eclipse SDK Framework is a credible and representative case study. The framework is open source and thus its source code is easily accessible. Despite these observations, our findings as usual in any empirical software engineering study cannot be directly generalized to other systems, specifically to systems implemented in other programming languages other than java.

8 Conclusion and Future Work

In this study we have carried out an investigation on the 28 Eclipse major release to establish the percentage of bug-free interfaces. We have observed that there exist over 85.9% and 88.2% bug-free APIs and non-APIs in all the studied Eclipse release. we observe that over 87.5% of the total number of classes in a given Eclipse are bug-free. This higher percentage may imply that Eclipse interfaces are generally tested for bugs. Furthermore, we provide a public dataset of 217,876 and 302,690 bug-free API and non-API classes respectively that can be used by both interface providers and users. We recommend that *Interface providers and users* can use our dataset findings as starting point to choose bug-free interfaces to use in their application instead of randomly using interfaces which might have bugs.

In a follow up study, we are planning to study the popularity of the identified bug-free interfaces by looking at both their internal and external usage. Internal interface usage can be determined by looking at how many packages and libraries in Eclipse framework use the identified bug-free interfaces. External usage can be determined by looking at how many applications on Github use bug-free interfaces. Similarly, external usage can be measured by looking at the number of developers who have used or touched a particular bug-free interfaces. Although majority of the interfaces have no bugs, this does not qualify them to meet all the other software quality metrics thus we plan to carry out an investigation and ascertain the other software quality of the identified bug-free interfaces by looking at more parameters like Technical debt, complexity, documentation, maintainability etc.

Furthermore, there exist a need to carry out a survey to get feedback from programmers or users of the interfaces about the bugs reported by both tools. There is also a need to cross-link bugs reported in Bugzilla-based Eclipse project platform with those found in this study by comparing the distribution, consistency, severity and new bugs detected by thetools but not yet reported on Bugzilla-based Eclipse project.

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.

Funding sources

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sector

Data availability

To allow independent replication and verification of the study, we provide a full replication package on Github with *the detailed lists of both bug-free public and internal APIs in different Eclipse releases*.[6]

Completing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

[6] https://github.com/simonkawuma/Bug-Free-APIs-in-Eclipse-Releases

Authors Contribution:

Kawuma Simon: Conceptualization, Methodology, Formal analysis, Resources, Data curation, conducting experiment, Writing - original draft, Writing - review & editing, Visualization. **David Sabiiti Bamutura:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing, Visualization. **Aggrey Obbo:** performing experiments, or data/evidence collection, Writing - original draft, Writing - review & editing. **Vicent Mabirizi:**Writing - original draft, Writing - review & editing, performing the experiments, or data/evidence collection. **Moreen Kabarungi:** Writing - original draft, Writing - review & editing. **Evarist Nabaasa:**Conceptualization, writing review & editing.

References

[1] 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)*. IEEE, 2019, pp. 117–127.

[2] S. Kawuma, J. Businge, and E. Bainomugisha, "Can we find stable alternatives for unstable eclipse interfaces?" in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[3] J. des Rivières, "How to use the Eclipse API," http://www.eclipse.org/articles/article.php?file=Article-API-Use/index.html, consulted January, 2020.

[4] Oracle, "Why developers should not write programs that call 'sun' packages," https://www.oracle.com/technetwork/java/faq-sun-packages-142232.html, consulted October, 2020.

[5] T. jBPM Team, "The jbpm api," http://docs.jboss.org/jbpm/v5.0/userguide/ch05.html#d0e2099, consulted October, 2020.

[6] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp et al., "Junit 5 user guide," https://junit.org/junit5/docs/current/user-guide/#api-evolution, consulted October, 2020.

[7] E. Foundation, "Evolving Java-based APIs," https://wiki.eclipse.org/Provisional_API_Guidelines, consulted July, 2020.

[8] J. Businge, A. Serebrenik, and M.G. Van Den Brand, "Eclipse api usage: the good and the bad," *Software Quality Journal*, Vol. 23, No. 1, 2015, pp. 107–141.

[9] 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*. ACM, 2016, pp. 278–289.

[10] J. Businge, A. Serebrenik, and M.G.J.v. Brand, "Analyzing the Eclipse API usage: Putting the developer in the loop," in *17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, 2013, pp. 37–46.

[11] A. Yan, H. Zhong, D. Song, and L. Jia, "How do programmers fix bugs as workarounds? an empirical study on apache projects," *Empirical Software Engineering*, Vol. 28, No. 4, 2023, p. 96.

[12] Z. Abou Khalil, E. Constantinou, T. Mens, and L. Duchien, "On the impact of release policies on bug handling activity: A case study of eclipse," *Journal of Systems and Software*, Vol. 173, 2021, p. 110882.

[13] Z. Abou Khalil, E. Constantinou, T. Mens, L. Duchien, and C. Quinton, "A longitudinal analysis of bug handling across eclipse releases," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 1–12.

[14] A. Campbell, "Metric definitions - sonarqube documentation," https://docs.sonarqube.org/display/SONAR/Metric+Definitions, consulted July, 2021.

[15] Spotbugs Community, "Spotbugs – find bugs in java programs," https://spotbugs.github.io/, 2022.

[16] L. Lavazza, D. Tosi, and S. Morasca, "An empirical study on the persistence of spotbugs issues in open-source software evolution," in *Quality of Information and Communications Technology*, M. Shepperd, F. Brito e Abreu, A. Rodrigues da Silva, and R. Pérez-Castillo, Eds. Cham: Springer International Publishing, 2020, pp. 144–151.

[17] The University of Maryland, "Findbugs – find bugs in java programs," ://findbugs.sourceforge.net/, 2022.

[18] J. des Rivières, "Evolving Java-based APIs," http://wiki.eclipse.org/Evolving_Java-based_APIs, consulted January, 2020.

[19] ProfessionalQA.com, “Difference between error, mistake, fault, bug, failure, defect.” https://www.professionalqa.com/difference-between-error-bug-mistake-fault-bug-failure-defect, consulted December, 2020.

[20] V. Lenarduzzi, A. Sillitti, and D. Taibi, “A survey on code analysis tools for software maintenance prediction,” in *International Conference in Software Engineering for Defence Applications*. Springer, 2018, pp. 165–175.

[21] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H.C. Gall et al., “How developers engage with static analysis tools in different contexts,” *Empirical Software Engineering*, Vol. 25, No. 2, 2020, pp. 1419–1457.

[22] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, “Are sonarqube rules inducing bugs?” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 501–511.

[23] J. Businge, A. Serebrenik, and M.G.J. van den Brand, “Eclipse API usage: the good and the bad,” in *SQM*, 2012, pp. 54–62.

[24] J. Businge, A. Serebrenik, and M.G.J.v. Brand, “Compatibility prediction of Eclipse third-party plug-ins in new Eclipse releases,” in *12th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 164–173.

[25] S. Kawuma and E. Nabaasa, “Identification of promoted eclipse unstable interfaces using clone detection technique,” *International Journal of Software Engineering and Application*, 2018.

[26] J. Businge, S. Kawuma, E. Bainomugisha, F. Khomh, and E. Nabaasa, “Code authorship and fault-proneness of open-source android applications: An empirical study,” in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2017, pp. 33–42.

[27] L. Guerrouj, Z. Kermansaravi, V. Arnaoudova, B.C. Fung, F. Khomh et al., “Investigating the relation between lexical smells and change-and fault-proneness: an empirical study,” *Software Quality Journal*, Vol. 25, No. 3, 2017, pp. 641–670.

[28] G. Bavota, M. Linares-Vasquez, C.E. Bernal-Cardenas, M. Di Penta, R. Oliveto et al., “The impact of api change- and fault-proneness on the user ratings of android apps,” *IEEE Transactions on Software Engineering*, Vol. 41, No. 4, 2015, pp. 384–407.

[29] M. Asaduzzaman, M.C. Bullock, C.K. Roy, and K.A. Schneider, “Bug introducing changes: A case study with android,” in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 116–119.

[30] R.G. Vieira, C.L.C. Mattos, L.S. Rocha, J.P.P. Gomes, and M. Paixão, “The role of bug report evolution in reliable fixing estimation,” *Empirical Software Engineering*, Vol. 27, No. 7, 2022, p. 164.

[31] F. Fang, J. Wu, Y. Li, X. Ye, W. Aljedaani et al., “On the classification of bug reports to improve bug localization,” *Soft Computing*, Vol. 25, 2021, pp. 7307–7323.

[32] F. Zampetti, R. Kapur, M. Di Penta, and S. Panichella, “An empirical characterization of software bugs in open-source cyber–physical systems,” *Journal of Systems and Software*, Vol. 192, 2022, p. 111425.

[33] A. Goyal and N. Sardana, “Nrfixer: Sentiment based model for predicting the fixability of non-reproducible bugs,” *e-Informatica Software Engineering Journal*, Vol. 11, No. 1, 2017.

[34] X. Xia, D. Lo, E. Shihab, and X. Wang, “Automated bug report field reassignment and refinement prediction,” *IEEE Transactions on Reliability*, Vol. 65, No. 3, 2015, pp. 1094–1113.

[35] D. Vyas, T. Fritz, and D. Shepherd, “Bug reproduction: A collaborative practice within software maintenance activities,” in *COOP 2014-Proceedings of the 11th International Conference on the Design of Cooperative Systems, 27-30 May 2014, Nice (France)*. Springer, 2014, pp. 189–207.

[36] W. Maalej and H. Nabil, “Bug report, feature request, or simply praise? on automatically classifying app reviews,” in *2015 IEEE 23rd international requirements engineering conference (RE)*. IEEE, 2015, pp. 116–125.

[37] Y. Fan, X. Xia, D. Lo, and A.E. Hassan, “Chaff from the wheat: Characterizing and determining valid bug reports,” *IEEE transactions on software engineering*, Vol. 46, No. 5, 2018, pp. 495–525.

[38] E. Foundation, “Eclipse project archived download,” http://archive.eclipse.org/eclipse/downloads/index.php, consulted January, 2020.

[39] E. Project, "Eclipse ide for java developers," https://www.eclipse.org/downloads/packages/release/2020-03/r, consulted July, 2020.

[40] 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)*. IEEE, 2017, pp. 146–148.

[41] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz et al., "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)*. IEEE, 2019, pp. 209–219.

[42] ggraham412, "Using internal interfaces while preserving encapsulation," https://www.codeproject.com/Articles/43156/Using-Internal-Interfaces-While-Preserving-Encapsu, consulted April, 2020.

[43] L.N.Q. Do, J.R. Wright, and K. Ali, "Why do software developers use static analysis tools? a user-centered study of developer needs and motivations," *IEEE Transactions on Software Engineering*, Vol. 48, No. 3, 2020, pp. 835–847.

[44] H. Zhong, X. Wang, and H. Mei, "Inferring bug signatures to detect real bugs," *IEEE Transactions on Software Engineering*, Vol. 48, No. 2, 2020, pp. 571–584.