ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/305026988

Can we find stable alternatives for unstable Eclipse interfaces?

Conference Paper · May 2016

DOI: 10.1109/ICPC.2016.7503716

CITATIONS
11

READS
16

3 authors, including:

Simon Kawuma
Mbarara University of Science & Technology (MUST)

11 PUBLICATIONS **59** CITATIONS

SEE PROFILE

John Businge
Mbarara University of Science & Technology (MUST)

4 PUBLICATIONS **72** CITATIONS

SEE PROFILE

All content following this page was uploaded by Simon Kawuma on 20 March 2023.

The user has requested enhancement of the downloaded file.

Can We Find Stable Alternatives For Unstable Eclipse Interfaces?

Simon Kawuma, John Businge
Mbarara University of Science and Technology
Mbarara, Uganda
{*simon.kawuma,jbusinge*}*@must.ac.ug*

Engineer Bainomugisha
Makerere University
Kampala, Uganda
ebainomugisha@cis.mak.ac.ug

***Abstract*—The Eclipse framework is a popular and widely adopted framework that has been evolving for over a decade. Like many other evolving software systems, the Eclipse framework provides both stable and supported interfaces (APIs) and unstable, discouraged, and unsupported interfaces (non-APIs). However, despite being discouraged by Eclipse, the usage of bad interfaces is not uncommon. Our previous research has confirmed that as Eclipse states, indeed APIs are stable while non-APIs are unstable. Applications using non-APIs face compatibility challenges in new Eclipse releases. Furthermore, our previous studies further revealed that the reason why application developers use the unstable interfaces is because they cannot find stable interfaces with the functionality that they require. Moreover, in a study we conducted, Eclipse application developers stated that they manually find the functionality from Eclipse. Eclipse being a very large complex software system with a large number of committers, we hypothesize that as developers manually search for the functionality they they require, it is possible that they miss stable interfaces offering the same functionality. To this end, using code clone detection techniques, we analyzed 18 major releases of Eclipse for possible clones. Our findings are threefold: i) we discover that indeed there exist clones in Eclipse, ii) we also discovered that some of the identified clones originate from different Eclipse projects, iii) our findings reveal that there is no significant number of APIs (less than 1%) offering the same or similar functionality as the non-APIs in all Eclipse releases we studied. This reveals that there are very few syntactic clones between API and non-API, thus developers were forced to either use non-API or find an API that exists, that is similar in functionality, but not in syntax.**

***Keywords*-Eclipse; Stable Interfaces; Unstable interfaces; code clone; Evolution**

I. Introduction and Motivation

Applications developed on top of frameworks are continuously growing. This is because the approach of developing applications on top of frameworks has many advantages, namely: Improved development speed [1], higher quality software [1] and reuse of the functionality provided by the framework application programming interfaces (APIs) i.e., there is no need to develop software from scratch. One widely used and adopted application framework is Eclipse. The Eclipse framework has been evolving since 2001 and a new version is released every year based on its an annual Eclipse release train policy. The current major release, *Eclipse Mars*, represents the work of 79 different open-source projects, 65 million lines of code, and involved 380 Eclipse committers[1]. The Eclipse Marketplace[2] reports over 1,685 Eclipse applications. Like any other evolving software system, the Eclipse framework provides two types of interfaces to application developers: the stable and supported interfaces (APIs) and unstable, undocumented, discouraged and unsupported interfaces (non-APIS) [2]. Eclipse framework developers (interface providers) discourage the use of non-APIs because they are immature and subject to change or could be removed from the framework without notice [2].

In our previous studies related to the Eclipse interfaces usage, we made a number of observations: First, while Eclipse providers discourage the use of non-APIs, we discovered that their use is not uncommon. We found that about 44% of the Eclipse third-party plug-ins (ETPs) on `sourceforge.net` use non-APIs and that these ETPs continue to use the non-APIs in new versions of the ETPs [3], [4]. Second, as stated by Eclipse providers [2], we confirmed that indeed APIs are stable since they do not cause compatibility failures to applications that solely depend on them [5]. Third, we also observed that indeed non-APIs cause compatibility failures to applications that use them in new framework releases [5], [6]. Fourth, we discovered that the major reason why application developers use non-APIs is that they claim they cannot find APIs with the functionality that they require. Moreover, experienced ETP developers revealed that using non-APIs is a better choice compared to writing their own APIs from scratch or re-implementing the non-APIs in their own code as they hope that the non-APIs will graduate to APIs in future. Fifth, in the survey we conducted about the use of Eclipse interfaces by developers, we learned that developers manually search for the functionality they require from the Eclipse interfaces [7].

The Eclipse framework constantly evolving releasing new versions with the aim of introducing new functionality and improving the quality of existing functionality. If application developers are to tap from the benefits in the new Eclipse release, then they must take care of the incompatibilities that come along. Lastly, as modifications are made on these Eclipse interfaces, the incompatibilities that new releases

[1]https://www.eclipse.org/org/press-release/20150624_mars_release.php
[2]http://marketplace.eclipse.org

cause about not only affects the Eclipse interface users (Application developers) but also the Eclipse interface providers (Eclipse core contributors) [8].

The Eclipse framework being a large complex open-source software system, with a large number of committers as well as many projects, it is possible that stable interfaces located in a different project or subproject could be offering the same functionality as the unstable interfaces. A study by Chanchal, Cordy and Koschke [9], [10] shows that in large code bases, typically 7% to 23% of the code is considered as code clones while in a similar study by Kamiya, Kusumoto and Inoue [11] report about 10% to 30%. To this end, we hypothesized that, because application developers manually search for functionality they require from Eclipse, it is possible that they could first land on an unstable interface (non-APIs) and miss a stable interface (APIs) offering the similar or same functionality.

In this paper, we analyze to what extent cloned Eclipse interfaces (method APIs) can be useful to both Eclipse interface providers and users in addressing the incompatibility challenges when changing to a new Eclipse release. Our study is guided by three major research questions. First, because Eclipse is a large and complex software system, we seek to understand indeed there exists clones.

RQ1. What is the percentage of cloned Eclipse interfaces in the different Eclipse releases?

Second, having identified that indeed clones exist in Eclipse, we proceed to identify if some of these identified clones originate from different Eclipse projects or originate from the same project but different sub-project. This is interesting because it means that clones are being introduces by different teams of Eclipse core contributors. If the introduced clones are non-API clones it is possible that are afraid of the change or disappearance of these non-APIs and therefore they decide to re-implement the code.

RQ2. What extent do the different cloned Eclipse interfaces originate from different Eclipse projects or same Eclipse project but different sub-projects?

Lastly, having addressed research RQ1 and RQ2, we proceed to determine if some of these identified clones can be useful to both the Eclipse interface providers and the interface users. In other words, amongst the identified clones, can we find clone pairs having one of the interfaces originating from APIs and the other from non-APIs. Positive results of this analysis can be an input to a API recommendation tool to the non-API users because they search for functionality they require manually.

RQ3. Using clone detection analysis, can we find stable Eclipse interfaces as an alternative to the unstable interfaces?

To answer the above research questions, we extract clone data from 18 major releases of Eclipse. The reason for this choice is two fold: 1) we want to study the evolutionary trend of Eclipse clones, and 2) the results are intended for all Eclipse users because some users may prefer to work with older Eclipse releases.

The remainder of the paper is organized as follows: In Section II we define the terms and concepts used in this paper. In Section III we explain how we collected the data used in this experiment. In Section IV we discuss the analysis of our results and the findings. In Section V we discuss the threats to validity affecting our findings. In Section VI we discuss the related work and finally, in Section VII we present the conclusions and future work.

II. Background

In this section we shall explain the different terms concepts we use in this paper.

A. Eclipse Plug-in Architecture

Eclipse SDK is an extensible framework comprising a set of tools working together to support programming tasks. Eclipse SDK is a collection of different plug-ins. The plug-ins can be classified into three different categories:

- *Eclipse Core Plug-ins (ECP):* These provide core functionality of the framework. When the users download the Eclipse SDK, they download all ECPs. These ECPs are developed by Eclipse core team contributors. These core teams contribute to the framework by wrapping their tools in pluggable components. The study in this paper is based on ECPs [12].
- *Eclipse Extension Plug-ins (EEP):* These are built with the main goal of extending the Eclipse platform. Like the ECPs, fully qualified names of EEP packages start with org.eclipse. but, as opposed to the ECPs, the EEPs are not considered to be a part of the Eclipse SDK. Examples of an EEPs include J2EE Standard Tools (JST) and Eclipse Modeling Framework (EMF) [12].
- *Eclipse Third-Party Plug-ins (ETP):* These are the remaining plug-ins. They use at least some functionality provided by ECPs but may also use functionality provided by EEPs. Many ETPs can be found in open-source repositories, e.g., SourceForge and GitHub.

B. Eclipse Interfaces

API offers an interface through which developers can access programmatic functionality. Eclipse has two different types of interfaces:

1) Eclipse non-APIs: These are the Eclipse unstable interfaces. The non-APIs are internal implementation artifacts that are found in a package with the substring `internal` in the fully qualified package name according to Eclipse naming convention [2]. The internal implementations include public Java classes or interfaces or public or protected methods, or fields in such a class or interface. Users are strongly discouraged from using any of the non-APIs because they may be unstable [13]. Eclipse clearly states that clients who

think they must use these non-APIs do it at their own risk as non-APIs are subject to arbitrary change or removal without notice. Eclipse does not usually provide documentation and support to these non-APIs.

2) Eclipse APIs: These are Eclipse stable interfaces. The APIs are the public Java classes or interfaces that can be found in packages that do not contain the segment internal in the fully qualified package name or a public or protected method, or field in such a class or interface. Eclipse states that, the 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 APIs.

In this study, we only look at Eclipse method interfaces because the tool we use in our data collection only outputs cloned methods. The other reason why we look at method interfaces is that users of the interfaces will mainly be calling methods inside their applications.

C. Clone Terminology

In this section, we define the different clone terminologies we used in the paper.

1) Code Fragment: Chanchal and Cordy [10] define code fragment (CF) as any sequence of code lines with or without comments. It can be of any granularity, for example, function definition, begin-end block, or sequence of statements. A code fragment is identified by its file name and begin-end line numbers in the original code base.

2) Code Clone: A code fragment CF2 is a relation of another code fragment CF1 if they are similar by some given definition of similarity, that is $f(CF1) = f(CF2)$ where f is the similarity function of two fragments that are similar to each other form a clone pair (CF1; CF2), and when many fragments are similar, they form a clone class or clone group [10].

3) Code pair: A pair of code portions/fragments is called a clone pair if there exists a clone relation between them, i.e., a clone pair is a pair of code portions/fragments which are identical or similar to each other [10].

4) Code Clone Types: Code fragments can be similar into two kinds namely; textual and functionality similarities. Also two code fragments do not have to be identical to be considered clones, they can have little differences. Code clones can be classified into four categories [10] namely:

- *Type-I* clones are identical code fragments except for variations in white space, layout, and comments.
- *Type-II* clones are structurally and syntactically identical except for variations in identifiers, literals, types, layout, and comments.
- *Type III* code clones are copies with further modifications,statements can be changed, added, or removed in addition to variations in identifiers, literals, types, layout, and comments.
- *Type-IV* clones are code fragments that have the functionalities but are implemented through different syntactic variants.

In this study we only look at clone Type-I, Type-II and Type-III because the tool, NiCad [14], we use to extract clones does not handle Type-IV clones and besides, finding cloned functionalities is more difficult.

D. Classification of clone pairs

In this study we identify three clone pair classifications:

- *non-API clone pairs*: If the interfaces the clone pair both originate from non-API files.
- *API clone pairs*: If the interfaces in the clone pair both originate from API files.
- *API-non-API clone pairs*: If one of the CF in the clone pair originates from an API file and the other from a non-API file.

III. Experimental Design and Data Collection

Our study is based on 18 Eclipse SDK major release source projects from Eclipse Project Archive [12]. We subjected the 18 Eclipse SDK releases to the tool NiCad [14] so as to extract clone reports. The clone report contains of clone pairs where each clone pair has got a *percentage of similarity* and the number of *lines of code* of the clone. Each clone method in the clone pair has got a method signature and a *source file* where the clone originates. In Fig 1, we show a representation of the XML clone report generated by subjecting Eclipse release `Rx` to the NiCad tool.

A. Percentage of Clones in Eclipse Releases

We now illustrate the methodology used to collect data for RQ1 with a example. Fig 2 show an example of a software system, `SS`, having three files, `f1`, `f2` and `f3`. `SS` comprises of four fully qualified methods, i.e., `f1.m1`, `f2.m1`, `f3.m1`, and `f3.m2`. Let us assume that method `m1` has been cloned in all the files. This means `SS` will have three fully qualified method clone pairs (cp), i.e., $cp1(f1.m1, f2.m1)$, $cp2(f1.m1, f3.m1)$, and $cp3(f2.m1, f3.m1)$. Hence we have three *fully qualified clone pairs* and three *fully qualified cloned methods*. We compute the percentage of clones in the software system `SS` as follows:

$$\frac{\text{Number of fully qualified methods clones}}{\text{Total number of fully qualified methods in } \texttt{SS}}\%$$

In the above example, software system `SS` would have a clone percentage of $3/4\% = 75\%$. In this paper, we shall frequently use the phrases *fully qualified clone pairs* and *fully qualified cloned methods*.

```
<clones>
    <clone nlines="74" similarity="93">
        public void md1(Cl c1) {<source="Rx/org/eclipse/p1/sp1/pk1/f1.java">
        public void md2(Cl c2) {<source="Rx/org/eclipse/p2/sp2/pk2/f2.java">
    </clone>
    <clone nlines="79" similarity="100">
        void m3(){<source="Rx/org/eclipse/p3/internal/sp3/pk3/f3.java">
        void m4(){<source="Rx/org/eclipse/p4/internal/sp4/pk4/f4.java">
    </clone>
    <clone nlines="90" similarity="95">
        public C1 m5(){<source="Rx/org/eclipse/p5/internal/sp3/pk5/f5.java">
        public C2 m6(){<source="Rx/org/eclipse/p6/sp3/pck6/f6.java">
    </clone>
    <clone nlines="90" similarity="100">
        public int m7(){<source="Rx/org/eclipse/p7/internal/sp3/pk7/f7.java">
        public int m8(){<source="Rx/org/eclipse/p8/sp8/pk8/f8.java">
    </clone>
</clones>
```

Fig. 1 – Sample Nicad Clone Report

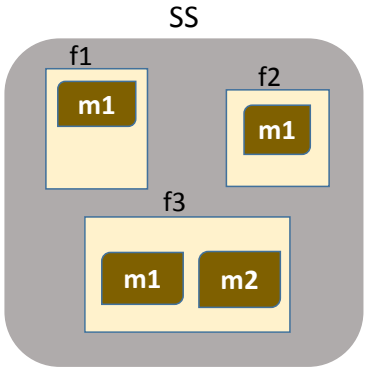

Fig. 2 – An example of a software system

B. Clone types data

To detect Type-I, Type-II, and Type-III code clones, with `NiCad` tool on the command line, we run the source files and any of the following `NiCad` configuration files: `type1.cfg`, `type2.cfg` and `type3-2.cfg`. The output of the execution is an `XML` file like example in Fig 1. Using the configuration file `type1.cfg` would produce an `XML` with only Type-I clones, `type2.cfg` would produces a combined file of both Type-I and Type-II and `type3-2.cfg` produces a combined file of all the three clone types. To isolate Type-II and Type-III, for simplicity, we use direct subtraction clone pair count, i.e., Type-II alone would be the number of clones pairs in produced by `type2.cfg` less those produced by `type1.cfg` and Type-III alone would be the number of clones pairs in produced by `type3-2.cfg` less those produced by `type2.cfg`. Of course direct subtraction would yield errors in case the source files contain nested methods, thus nested clones would be missed. An alternative method to avoid errors is to carry a step-wise subtraction by using the unix command `comm` to separate the two files. The step-wise subtraction produced a difference of less that 0.4% on Eclipse 3.4 source release.

C. Different project/sub-project data

Eclipse files are structured in a systematic way. All Eclipse project files start with a substring `org.eclipse`. For example, looking at Fig 1, the project name always follows after the substring `org.eclipse` and the sub-project name follows after the project name. From this Eclipse file naming convention, one can easily determine the different categories of clones mentioned in this section. Furthermore, we collected data about Eclipse committers for six Eclipse projects namely: `JDT`, `SWT`, `UI`, `TEXT`, `PDE`, and `DEBUG`.

D. Clone Pair Classifications Data

In this section, we explain how we collected data for RQ3. To extract data for the different clone pair classifications, i.e., API, non-API and API-non-API: from Fig 1, one can tell that the a method in a clone pair is an API or non-API or an API-non-API by looking for the substring `internal` in the source file where the methods originate. For example, the first clone pair in Fig 1 is an API classification, the second in a non-API classification and the third is an API-non-API classification. To determine the percentage of non-APIs that have alternative APIs offering the same functionality, we count the number of fully qualified non-API methods in classification API-non-API divided by the total non-API methods.

IV. Results and Findings

We now discuss the results and findings of our study.

Table I – Summary statistics for number of fully-qualified cloned methods and the total fully-qualified methods for all the Eclipse releases

	Cloned methods	Total methods
Mean	15,640	171,665
Median	17,050	192,672
Min	3,482	40085
Max	22,646	238,919

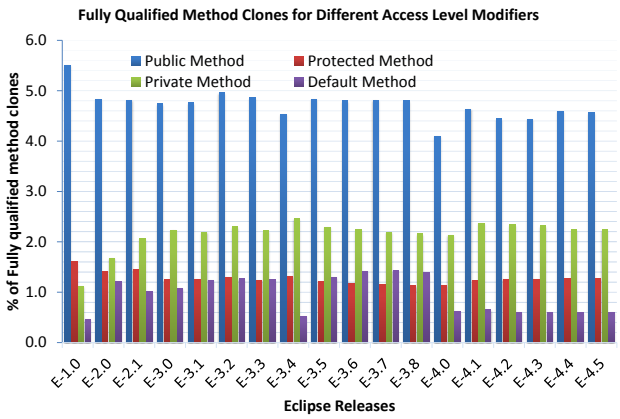

Fig. 3 – A graph showing the percentage of fully-qualified cloned methods for the different method access level modifiers in each Eclipse release, respectively.

A. Percentage of Clones in Eclipse

RQ1. What is the percentage of cloned Eclipse interfaces in the different Eclipse releases?

1) Results: Based on the methodology presented in Section III-A, we discovered that the percentage of clones in the different Eclipse releases range from 8.0% to 9.8%. In Table I we show the summary statistics corresponding to the percentages of clones in all the Eclipse releases. Column–*Cloned methods* in Table I shows the summary statistics of the fully-qualified cloned methods while column–*Total methods* shows the summary statistics total number of fully qualified methods in all the Eclipse releases. In Figs 3 and 4, we present additional results corresponding to percentage of clones in Eclipse. In these figures, we present graphs of clone percentages corresponding to fully qualified cloned methods for the different access level modifiers and the different clone types, respectively, for each of the Eclipse SDK releases. In Fig 3, for each Eclipse release there are four bars. The first bar indicates the percentage of fully-qualified cloned methods that are have an access level modifier of `public`, followed by `protected`, `private`, and `default`, respectively. In Fig 4, for each Eclipse releases there are three bars. The first bar indicates the percentage of fully-qualified cloned methods corresponding to Type-I clones, followed by Type-II clones and Type-III clones, respectively.

2) Findings: First, as can be seen, our results of the percentage of cloned methods ranging from 8.0% and 9.8% for the different Eclipse releases. Indeed these results correspond to what has been reported in literature by Chanchal, Cordy and Koschke [9], [10]. Second, from Fig 3, we can observe that the `public` level access modifier has a very high percentage relative to the other modifiers. This is not surprising because Eclipse is a framework with interfaces that is designed to be used by other applications. These findings are interesting for our major research–RQ3 because interface users reuse framework functionality by calling the calling *methods interfaces* having the access level modifier `public` as these methods are outside the framework. Third, in Fig 4 we observe that Type-III clones have a much higher percentage than Type-I and Type-II clones. The reason for this observation is that unlike Type-III, Type-I and Type-II clones do not allow any differences at all, for example, ff a line has changed in a clone, that clone will not be reported. Lastly, in the clone evolution perspective, looking at Figs 3 and 4, we observe a similar trends in the percentage of clones across all the Eclipse releases studied. There is only a slight change in Eclipse release 4.0, a trend that we also observe in the results of research question RQ2 and RQ3. The reason 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 onwards.

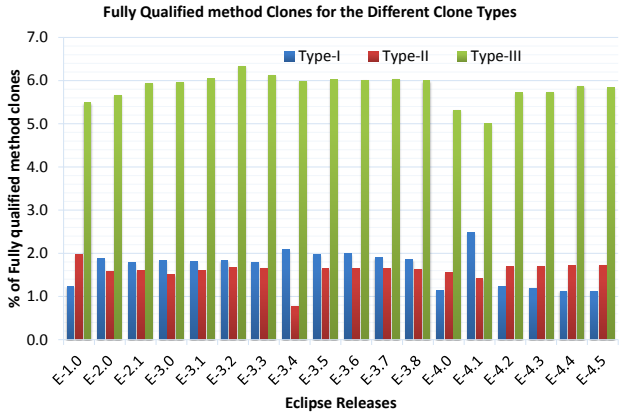

Fig. 4 – A graph showing the percentage of fully-qualified cloned methods for the different clone types in each Eclipse release, respectively.

B. Percentage of Clone pairs in different projects/sub-projects

RQ2. What extent do the different cloned Eclipse interfaces originate from different Eclipse projects or originate from same Eclipse project but different sub-projects?

1) Results: In Tables III to IV and Figs 5 and 6, we present the results of RQ2. Table III shows the different summary statistics corresponding to fully-qualified method clone pairs with methods originating the different categories

Table II – The summary statistics of the fully-qualified cloned methods in clone pairs originating from the following categories: i) same project and sub-project–*column Same proj/sub-proj*, ii) different projects–*column Diff proj*, and iii) same project but different sub-projects–*column Same proj/diff sub-proj*, for all the Eclipse releases.

	Same proj/sub-proj	Diff proj	Same proj/diff sub-proj
Mean	165,271	2,594	3,800
Median	185,817	2,904	4,254
Min	38,721	419	945
Max	230,024	3,685	5,210

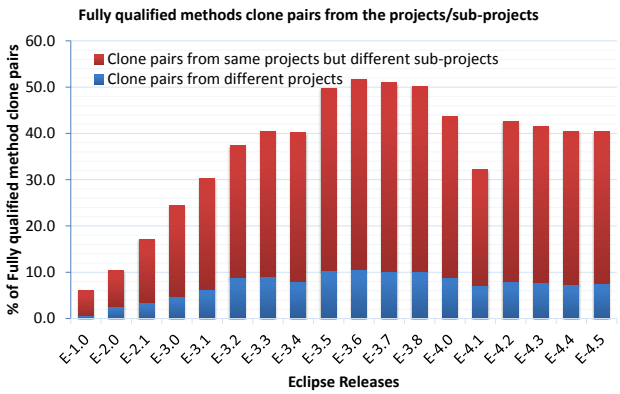

Fig. 5 – A graph showing the percentage of fully-qualified method clone pairs originating from different Eclipse projects and those originating from same project but different sub-projects.

in RQ2. In Table II, we present the summary statistics of the fully-qualified cloned methods in clone pairs originating from the different categories. In Table IV, we present statistics of Eclipse core developers committing in one or more Eclipse projects. A total of 151 committers were involved in at least one the six Eclipse projects presented in Section III, were collected. Column–*Project Involvement* shows the number of projects that committers are involved in one of more projects and column–*# of Committers* shows the corresponding number of committers involved. For example, each of 111 committers were involved in only one of the six Eclipse projects, 22 of the committers were involved in two of the six Eclipse projects and one committer was involved in all the six Eclipse projects.

In Fig 5, we present the percentages of fully-qualified method clone pairs in each Eclipse release corresponding to summary statistics of Table III. Each of the Eclipse releases in Fig 5 has got two bars (bottom and top) representing the percentage of the two clone pair categories. The percentages of each of the two clone pair categories in a given each Eclipse release, say `E-Rx`, is computed as a fraction of the total number of clone pairs in `E-Rx`. In Fig 6 we present the percentages of fully-qualified cloned methods in clone pairs, for each Eclipse release, corresponding to the summary statistics in Table II. Each Eclipse release in Fig 6 has got two bars (bottom and top) representing the percentage of the two clone pair categories. The percentages of each of the two cloned method categories in a given each

Table III – The summary statistics of the fully-qualified method clone pairs with methods originating from the following categories: i) same project and sub-project–*column Same proj/sub-proj*, ii) different projects–*column Diff proj*, and iii) same project but different sub-projects–*column Same proj/diff sub-proj*, for all the Eclipse releases.

	Same proj/sub-proj	Diff proj	Same proj/diff sub-proj
Mean	20,586	2,713	10,671
Median	20,001	3,121	12,151
Min	13,652	105	799
Max	29,966	4,345	16,896

Table IV – Frequency of committers involved in different Eclipse Projects.

Project Involvement	# of Committers
1 Project	111
2 Projects	22
3 Projects	11
4 Projects	3
5 Projects	3
6 Projects	1

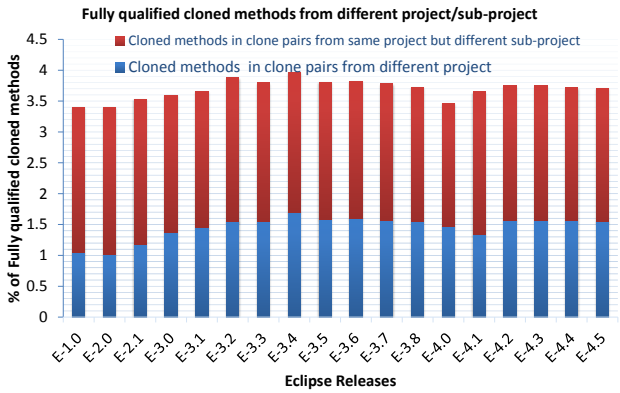

Fig. 6 – A graph showing the percentage of fully-qualified cloned methods in clone pairs originating from different Eclipse projects and those originating from same project but different sub-projects.

Eclipse release, say `E-Rx`, is computed as a fraction of the total number of fully-qualified methods in `E-Rx`.

2) Findings: In this section we discuss the findings of research question RQ2. As can be seen from Fig 5, we observe that we have up to $40-50\%$ clone pairs of the total clone pairs identified in Eclipse with methods in clone pairs originating from different projects/sub-projects. In Fig 6 we have an average of 3.7% cloned methods as a fraction of the total methods in Eclipse with methods in clone pairs originating from different projects/sub-projects. On further investigation to the possible cause of this behaviour, we discovered that some committers are involved in more than one Eclipse project as presented in Table IV. It is possible that these committers copy code from one Eclipse project they are involved in to other Eclipse projects they are involved in.

C. Clone pair Classifications

RQ3. To what extent can Eclipse interface providers/users benefit from the cloned Eclipse interfaces to address the

interface incompatibilities in new Eclipse releases?

1) Hypothesis Testing: As we stated in Section I, in the survey that we carried out to discover why developers use non-APIs [7], one of our findings was that Eclipse interface users find the functionality that they require manually. A a result of the manual search, we thought that because Eclipse is a very large complex software systems with many committers, it is possible that there could be alternative APIs offering the same functionality as the non-APIs. Based on the claims of the Eclipse developers, we come up with the following hypothesis.

- H_0: *There is no significant number of APIs offering the same or similar functionality as the non-APIs in all Eclipse releases*;
- H_a: *There is a significant number of APIs offering the same or similar functionality as the non-APIs in all Eclipse releases*.

2) Results: Tables V and VI and Figs 7 to 10 we present our results. In Table V, we present summary statistics corresponding to the evolution of the number of methods in Eclipse releases. In Tables VII and VI, we present summary statistics for the fully-qualified method clone pairs and fully-qualified cloned methods for the different clone pair classifications, respectively, for all the Eclipse releases. In Fig 7 we present the percentages of methods originating from the two Eclipse interface categories, APIs and non-APIs, corresponding to the summary statistics in Table V. In Fig 8, we present percentages of the fully-qualified method clone pairs (as a fraction of the total clone pairs) in each classification for the different Eclipse releases, corresponding to the summary statistics in Table VII. Each Eclipse release in Fig 8 has got three bars. The first bar corresponds to percentage of fully-qualified clone pairs in classification–*non-APIs*, the next two bars show the percentages for *APIs* and *API-non-API* classification, respectively. In Fig 9, we present percentages of the fully-qualified cloned methods (as a fraction of the total methods in Eclipse), in each classification for the different Eclipse releases, corresponding to the summary statistics in Table VI. Each Eclipse release in Fig 9 has three bars. The first bar corresponds to percentage of fully-qualified cloned methods in classification–*non-APIs*, the next two bars show the percentages for *APIs* and *API-non-API* classification, respectively. In Fig 10, we present the results of our main research question RQ3. In the figure, we show percentages of fully-qualified non-API methods for clone Type-I, II and III in the different Eclipse releases.

3) Findings: We observe a number of findings: First, looking at Fig 7, we observe that, on average, the percentages of non-API methods are twice as much those API methods along the evolution of Eclipse. Comparing the actual numbers and the percentages, we observe that non-API and API methods are growing. This means that the rate at which new non-APIs are introduced in new Eclipse releases is much higher compared to the rate at which the

Table V – The summary statistics of the fully-qualified methods originating from non-APIs and APIs in each of the Eclipse releases.

	non-API	API
Mean	122,665	54,353
Median	144,401	55,054
Min	30,766	25,961
Max	160,680	78,239

Table VI – The summary statistics of the `public` fully-qualified cloned methods for the different clone pair classifications in each of the Eclipse releases, respectively.

	non-API	API	API-non-API
Mean	5,082	2,438	1,398
Median	5,831	2,375	1,509
Min	1,788	277	175
Max	6,502	5,572	2,059

Table VII – The summary statistics of the `public` fully-qualified method clone pairs for the different clone pair classifications in each of the Eclipse releases, respectively.

	non-API	API	API-non-API
Mean	11,517	7,447	2,017
Median	11,646	7,456	2,255
Min	9,458	250	150
Max	13,622	13,730	3,184

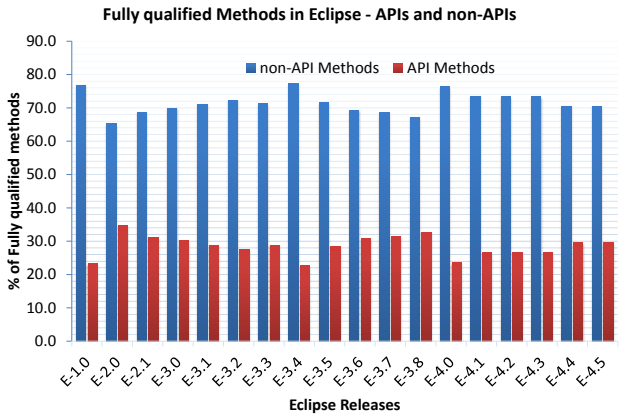

Fig. 7 – A graph showing the percentages of the fully-qualified methods originating from non-APIs and APIs for the different Eclipse releases.

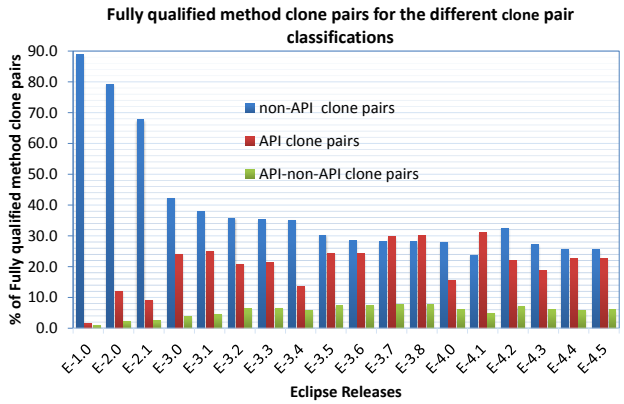

Fig. 8 – A graph showing the percentages of the `public` fully-qualified method clone pairs for the different clone pair classifications in the different Eclipse releases, respectively.

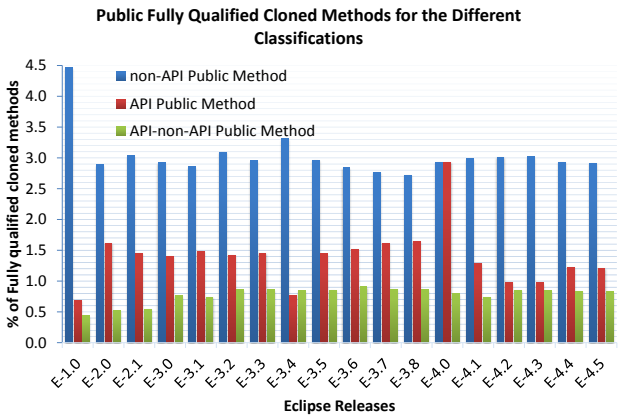

Fig. 9 – A graph showing the percentages of the `public` fully-qualified cloned methods for the different clone pair classifications in the different Eclipse releases, respectively.

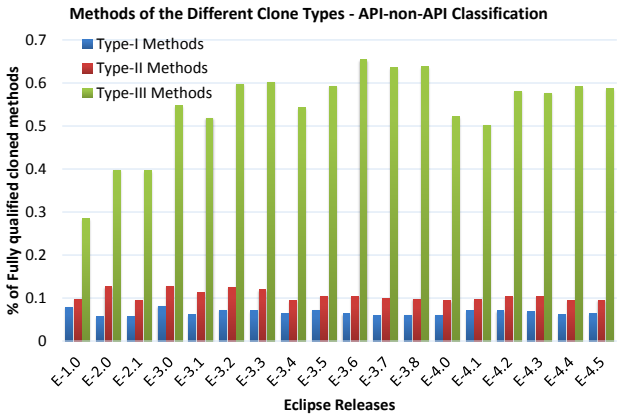

Fig. 10 – A graph showing the percentages of the `public` fully-qualified non-API methods having similar functionality as the APIs in the different Eclipse releases, respectively.

APIs are introduced. This tells us that there is an introduction of more new non-APIs methods compared to the graduation of non-APIs methods to APIs files. It is also possible that there is a very low graduation of non-APIs to APIs. Second, looking at Figs 8 and 9, in general, we observe that the highest percentages are observed in classification non-API, followed by API and finally API-non-API. Comparing the percentages of API and non-API in Fig 9 we observe that we constantly have higher percentages of non-API cloned methods in comparison to those of APIs in all the Eclipse releases. An explanation for this observation could be as follows; in RQ2 we observed that some clone pairs have methods originating from different projects and these projects are being developed by different groups committers. It is possible that these committers decide to re-implement the functionality of non-APIs from other projects in their own code for fear that these non-APIs may disappear or change in the next Eclipse release. Third, looking at Fig 10 we observe that the percentage of non-API methods that we can find alternative same or similar API methods is very low, i.e., less than 1% combined for clone Type-I,II, and III for all the Eclipse releases studied. Based on these results, in this study, we confirm our hypothesis, H_0, that indeed *there are no significant numbers of APIs offering the similar functionality as the non-APIs in all Eclipse releases*.

Based on the results of the hypothesis and the fact that Eclipse offers twice as much functionality in the non-APIs than in the APIs, Eclipse interface users could be right to state that the reason that they use non-APIs is because the functionality they require can only be found in non-APIs. Our study also reveals that there are very few syntactic clones between API and non-API, thus developers were forced to either use non-API or find an API that exists, that is similar in functionality, but not in syntax. To reduce on the challenges faced by the Eclipse interface users, our recommendation to Eclipse interface providers is that they should expedite the graduation of non-APIs to APIs. Although the results that we have discovered in this study about alternative APIs for the non-APIs are insignificant, Eclipse providers can use them as a starting point to provided APIs from non-APIs for the Eclipse releases that they still support. For each of the higher releases of Eclipse, the following are the average numbers of API methods alternatives for non-API methods we have discovered: 100 for Type-I, 150 for Type-II, and $1,000$ for Type-III. It would cost less effort to convert non-APIs we have found to APIs than doing it from scratch.

V. 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 clones in Eclipse is subjected to construct validity. The reason being that we only use methods in our computations yet there are other objects we ignore, for example, variable declarations.

Internal validity is related to validity of the conclusion within the experimental context of the data collection considered above. Our conclusions are threaten by internal validity because in our clone detection analysis we do not consider Type-IV clones. The reason is that it is difficult for a tool to identify Type-IV clones. It is possible that our results could have been different if we could have also detected Type-IV clones. In Section III we mentioned that, for simplicity, we used direct subtraction as opposed to stepwise subtraction to isolate Type-II and Type-III clones. If the code we are analyzing contains a lot of nested methods, direct subtraction would result into subtraction errors. On testing Eclipse version 3.4 we discovered that the difference in the two computation methods is less that 0.4%. This

means that there are very few nested methods in Eclipse source code.

External validity is the validity of generalizations based on this study. Our study is generalizable to some extent since frameworks, open-source or commercial, are designed to provide interfaces to users. However, to confirm this claim, in our followup studies carry out the same study on a different plug-in framework.

VI. Related Work

In the previous sections, we implicitly discussed how the current work relates to our previous work [3]–[7], [15]. In general, our previous work is 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, we studied how the changes in the Eclipse interfaces used by the ETPs, affect compatibility of the ETPs in forthcoming framework releases. We only used open-source ETPs in the study and the analysis was based on the source code. In one of previous studies [7] was based on analysis of a survey, where we complements our other previous studies by including commercial ETPs and taking into account human aspects. In all your previous studies our major findings were that interface users are continuously using unstable interfaces and the reason for using these unstable interfaces was because there no alternative stable interfaces offering the same functionality. This study is based on verifying if the claim by the interface users is true. We use code clone detection analysis to verify the claim.

There are many studies that have been carried out on clone detection analysis in large software projects. Most of these studies try to understand how clones are introduced, changed or removed across different versions of a software. Lague et al. [16] examined six versions of a telecommunication software system and found that a significant number of clones were removed due to refactoring, but the overall number of clones increased due to the faster rate of clone introduction. Kim et al. [17] describe a study of clone genealogies and find that: (1) many code clones are short-lived, so performing aggressive refactoring may not be worthwhile; and (2) long-lived clones pose great challenges to refactoring because they evolve independently and can deviate significantly from the original copy. [18] carried out an empirical study based on the clone genealogies from a significant number of releases of six software systems, to characterize the patterns of clone change and removal in evolving software systems. Their findings reveal insights into the removal of individual clone fragments and provide empirical evidence in support of conventional clone evolution wisdom. Chen et al. [19] investigate the use of a clone detector to identify known Android malware. They extract the Java source code from the binary code of the Android applications and use NiCad, a near-miss clone detector, to find the classes of clones in a small subset of the malicious applications. They then use these clone classes as a signature to find similar source files in the rest of the malicious applications. Their results show that using a small portion of malicious applications as a training set can detect 95% of previously known malware with very low false positives and high accuracy at 96.88%. Casazza et al. [20] used metrics based clone detection to detect cloned functions within the Linux kernel. They discovered that in general the addition of similar subsystems was done through code reuse rather than code cloning, and that more recently introduced subsystems tend to have more cloning activity. Antoniol et al. [21] h did a similar study as that of Casazza et al. [20] by evaluating the evolution of code cloning in the Linux, and they found out that the structure of the Linux kernel did not appear to be degrading due to code cloning activities. In comparison to our work, we use clone detection analysis to identify if there exists Eclipse stable interfaces offering the same or similar functionality as the as the unstable interfaces in the different Eclipse releases.

VII. Conclusion and Future Work

In this study we have investigated the existence of cloned interfaces in 18 major releases of the Eclipse SDK. First, we studied the code cloning of Eclipse method interfaces in general. We discovered that in all the Eclipse releases, cloned method range from 8.0% to 9.8%. Our findings are related to what has been reported by earlier researchers in literature. Secondly, we studied code cloning for the identified clone pairs in the different Eclipse releases based on two categories: i) methods in clone pairs originating from different Eclipse projects, ii) methods in clone pairs originating from same project but different sub-projects. Our findings in all the Eclipse releases indicate that in category–(i) an average of about 1.5% and in category–(ii) an average of about 2.2% of the fully-qualified cloned methods as a fraction of the total number of methods in Eclipse. On further investigations on the possible causes of observed behaviour, we discovered that some committers are involved in more than one Eclipse project. It is possible that these committers copy code from one Eclipse project they are involved in to other Eclipse projects they are also involved in. Third, using clone detection analysis, we wanted to discover if there exists stable Eclipse interfaces (APIs) as an alternative for the unstable ones (non-APIs). Our findings reveal that Eclipse offers twice as much functionality in the non-APIs compared to the APIs it provides to the interface users. Furthermore, we did not find significant numbers (less than 1%) of APIs offering the same or similar functionality as the non-APIs in all Eclipse releases. This reveals that there are very few syntactic clones between API and non-API, thus developers were forced to either use non-API or find an API that exists, that is similar in functionality, but not in syntax.

We have observed that in all the 18 Eclipse releases, there exists more than two times non-APIs compared to the

APIs. To reduce on the challenges faced by the Eclipse non-API users, our recommendations to the Eclipse interface providers are that they should expedite the graduation of non-APIs to APIs. Although the results we have discovered in this study about alternative APIs for the non-APIs are insignificant, Eclipse providers can use them as a starting point to provided APIs from non-APIs for the Eclipse releases they still support. It would cost less effort to convert non-APIs we have found to APIs than doing it from scratch. In our followup study, we would like to investigate the evolution of non-APIs and assess their graduation rate to APIs.

ACKNOWLEDGMENT

We would like to thanks Dr. James R. Cordy for providing the latest version of NiCad tool and also answering our inquiries regarding NiCad. We would also like to extend our appreciation for the funding from the Sida-BRIGHT project.

REFERENCES

[1] J. M. Daughtry, U. Farooq, B. A. Myers, J. Stylos, Api usability: Report on special interest group at chi, SIGSOFT Softw. Eng. Notes 34 (4) (2009) 27–29.

[2] J. des Rivières, How to use the Eclipse API, http://www.eclipse.org/articles/article.php?file=Article-API-Use/index.html, consulted on January 01, 2011 (2001).

[3] J. Businge, M. van den, A. Serebrenik, Eclipse API usage: The good and the bad, Software Quality Journal 23 (2013) 107–141.

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

[5] J. Businge, A. Serebrenik, M. G. J. van den Brand, Survival of Eclipse third-party plug-ins, in: ICSM, 2012, pp. 368–377.

[6] J. Businge, A. Serebrenik, 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.

[7] J. Businge, A. Serebrenik, 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.

[8] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of api changes and usages based on apache and eclipse ecosystems, Empirical Software Engineering (2015) 1–47doi:10.1007/s10664-015-9411-7.
URL http://dx.doi.org/10.1007/s10664-015-9411-7

[9] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming 74 (2009) 470495.

[10] C. K. Roy, J. R. Cordy, A survey on software clone detection research, Queen's Technical Report: 541, p. 115, 2007.

[11] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: A multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28 (7) (2002) 654–670.

[12] Eclipse project archived download, http://archive.eclipse.org/eclipse/downloads/index.php, consulted November, 2013.

[13] J. des Rivières, Evolving Java-based APIs, http://wiki.eclipse.org/Evolving_Java-based_APIs, consulted on January 01, 2011 (2007).

[14] J. R. Cordy, C. K. Roy, The nicad clone detector, in: ICPC '11 Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension Pages 219-220, 2011, pp. 219–220.

[15] J. Businge, A. Serebrenik, M. G. J. van den Brand, An empirical study of the evolution of Eclipse third-party plug-ins, in: EVOL-IWPSE'10, ACM, 2010, pp. 63–72.

[16] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, J. Hudepohl, Assessing the benefits of incorporating function clone detection in a development process, in: Proceedings of the International Conference on Software Maintenance, ICSM '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 314–.
URL http://dl.acm.org/citation.cfm?id=645545.853273

[17] M. Kim, V. Sazawal, D. Notkin, G. Murphy, An empirical study of code clone genealogies, SIGSOFT Softw. Eng. Notes 30 (5) (2005) 187–196. doi:10.1145/1095430.1081737.
URL http://doi.acm.org/10.1145/1095430.1081737

[18] M. F. Zibran, R. K. Saha, C. K. Roy, K. A. Schneider, Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, ACM, New York, NY, USA, 2013, pp. 1123–1130. doi:10.1145/2480362.2480573.
URL http://doi.acm.org/10.1145/2480362.2480573

[19] J. Chen, M. H. Alalfi, T. R. Dean, Y. Zou, Detecting android malware using clone detection, J. of Comp. Sci. and Tech. 30 (5) (2015) 942–956.

[20] G. Antoniol, U. Villano, M. D. Penta, G. Casazza, E. Merlo, Identifying clones in the linux kernel, in: 1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), 10 November 2001, Florence, Italy, 2001, pp. 92–99. doi:10.1109/SCAM.2001.10003.

[21] G. Antoniol, U. Villano, E. Merlo, M. D. Penta, Analyzing cloning evolution in the linux kernel, Information & Software Technology 44 (13) (2002) 755–765. doi:10.1016/S0950-5849(02)00123-4.

View publication stats