How Stable are Eclipse Application Framework Internal Interfaces?

John Businge*, Simon Kawuma*, Moses Openja*, Engineer Bainomugisha† and Alexander Serebrenik‡
*Mbarara University of Science and Technology, Mbarara, Uganda
†Makerere University, Kampala, Uganda
‡Eindhoven University of Technology, Eindhoven, The Netherlands

***Abstract*—Eclipse framework provides two interfaces: stable interfaces (APIs) and unstable interfaces (non-APIs). Despite the non-APIs being discouraged and unsupported, their usage is not uncommon. Previous studies showed that applications using relatively old non-APIs are more likely to be compatible with new releases compared to the ones that used newly introduced non-APIs; that the growth rate of non-APIs is nearly twice as much as that of APIs; and that the promotion of non-API to APIs happens at a slow pace since API providers have no assistance to identify public interface candidates.**

Motivated by these findings, our main aim was to empirically investigate the entire population (2,380K) of non-APIs to find the non-APIs that remain stable for a long period of time. We employ cross-project clone detection to identify whether non-APIs introduced in a given Eclipse release remain stable over successive releases. We provide a dataset of 327K stable non-API methods that can be used by both Eclipse interface providers as possible candidates of promotion. Instead of promoting non-APIs which are too fine-grained, we summarized the non-API methods groups in given classes that are stable together and present class-level non-APIs that possible candidates promotion. We have shown that it is possible to predict the stability of a non-API in subsequent Eclipse releases with a precision of $\geq$ 56%, a recall of $\geq$ 96% and an AUC of $\geq$ 92% and an F-measure of $\geq$ 81%. We have also shown that the metrics of length of a method and number of method parameters in a non-API method are very good predictors for the stability of the non-API in successive Eclipse releases. The results provided can help the API providers to estimate a priori how much work could be involved in performing the promotion.

***Index Terms*—Eclipse; Framework; Internal Interfaces; Internal Interfaces Promotion; Internal Interfaces Stability; Software Evolution**

I. Introduction

Application developers build their systems on top of frameworks and libraries [1]. Building applications this way fosters reuse of functionality [2] and increases productivity [3]. This is why large application frameworks such as Eclipse [4] MSDN [5], jBPM [6], JUnit [7] commonly provide public (stable) interfaces (APIs) to application developers.

In addition to public (stable) interfaces all these frameworks also provide internal (possibly unstable) interfaces (non-APIs). Eclipse, jBPM, jUnit, all adopt the convention of internal interfaces by using sub-string `internal` in their package names while JDK non-APIs packages start with the substring `sun`. Framework developers discourage the use of non-APIs because they may be immature, unsupported, and subject to change or removal without notice [4], [5], [7], [8]. Supporting these recommendations previous empirical studies have shown that when the Eclipse application framework evolves, APIs do not cause compatibility failures in applications that solely depend on them [9], while non-APIs cause compatibility failures in applications that depend on them [9], [10].

Despite the non-API being discouraged and causing compatibility failures, usage of non-APIs is not uncommon. Businge et al. have observed that about 44% of 512 Eclipse plug-ins use non-APIs [11], [12]: application developers claim that they cannot find APIs with the functionality they require among APIs and therefore feel compelled to use non-APIs [13].

Much as the developers discourage the use of non-APIs they do know that application developers use them. One example of non-API client usage known to the API providers is the non-API class `org.eclipse.jdt.internal.corext.dom.NodeFinder`.

> *On `Bugzilla` an Eclipse bug tracking forum a client requested that: `NodeFinder` class is part of the package `org.eclipse.jdt.internal.corext.dom` and provides very useful logic. Would be nice if the node finder (or a similar one) becomes part of the AST API[1] (reported after the release of Eclipse 2.1 (25-20-2004)). After a series discussions and adaptations on `NodeFinder` over years, the interface providers promoted the `NodeFinder` to API package `org.eclipse.jdt.core.dom` during Eclipse release 3.6 (05-10-2009 six years later).*

In a preliminary study of the non-APIs, Kawuma et al. [14] observed twice as many fully qualified non-API methods compared to APIs methods. As a solution to mitigate discussed risks and help client developers, API producers may promote some internal interfaces to public ones. However, Hora et al. [15] discovered that promotion occurs slowly causing a delay to client developers to benefit from stable and supported interfaces. The authors further state that slow promotion results from API producers having no assistance in identifying public interface candidates (i.e., internal interfaces that should be public). In this paper, using *clone detection techniques*, we study the stability of Eclipse internal interfaces over subsequent Eclipse releases. Our main aim is *to detect internal interfaces that could be recommended to the API producers for promotion to public interfaces*.

[1] Commit: http://goo.gl/MlrRzx. Request Issue: http://goo.gl/DLRUKS.

The remainder of this paper is organized as follows: Section II presents the background information on Eclipse Framework and its interfaces. Section III discusses the experimental setup of our study. We present our answers for the research questions in Sections IV and V. Section VI presents threats to the validity of our study, while Section VII provides an overview of the related work. Finally, Section VIII concludes the paper and outlines some avenues for future work.

II. Background

Similarly to most of the previous studies of non-APIs we focus on Eclipse [9], [10], [11], [12], [13], [14], [15]. This section introduces the background related to Eclipse interfaces and to the context of the technique we use, software clones.

A. Eclipse Interfaces

Eclipse application framework provides two different types of interfaces.

Eclipse non-APIs: Non-APIs are internal implementation artifacts that according to Eclipse naming convention [4] 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 or protected methods, or fields in such a class or interface. Usage of non-APIs is strongly discouraged since they may be unstable [16]. 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.

Eclipse APIs: These are public Java classes or interfaces 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 APIs are considered to be stable and can be used by any application developer without any risk. Furthermore, Eclipse also provides documentation and support for these APIs.

B. Clone Terminology

We use clone detection techniques to determine the stability of a non-API method between subsequent Eclipse releases. Software clone detection is a well-established research area [17], [18]; in the remainder of the section we briefly introduce notions related to clone detection.

Code Fragment [17] is a sequence of code lines with or without comments. A code fragment is identified by its file name and begin-end line numbers in the original code base.

A code fragment CF2 is a *clone* of another code fragment CF1 if they are similar by some given definition of similarity. The pair (CF1; CF2) form then a *clone pair*. If multiple fragments are similar, they form a clone class or clone group [17].

Code Clone Types Depending on definition of similarity different clone types can be distinguished. In this study we consider code clones of Types I, II, & III [17]. *Type-1* clones are identical code fragments except for variations in white space, layout, and comments. *Type-2* clones are structurally and syntactically identical except for variations in identifiers, literals, types, layout, and comments. *Type-3* 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.

C. Stability of non-APIs

In this section we formalize the notion of non-API stability.

Def. 1: Let E_{old} and E_{new} be two Eclipse releases. We say a non-API method m is *stable* between E_{old} and E_{new}, if m in E_{old} and m in E_{new} are a pair of Type-1 or Type-2 clones.

Since the inception with Eclipse 1.0, the Eclipse framework has been introducing a new release every year. Therefore, the non-APIs that have remained unchanged since Eclipse release 1.0 and still present in Eclipse 4.6 are now 15 years old.

Using the notion of stability we pose the following research questions.

RQ1 **Are there stable non-API methods that could be candidates for promotion to APIs?**
We hypothesize that the non-APIs that have not changed over subsequent Eclipse releases could be mature and therefore possible candidates of promotion to APIs. Using our methodology identify stability of the non-APIs at method-level as a way of identifying candidates for promotion. We understand that method-level non-APIs that are candidates of promotion may be too fine grained for recommendation. However, if we identify groups of methods in a class that are stable, then they could be together candidates for promotion at class-level.

RQ2 **Can we predict whether a non-API will remain stable in subsequent Eclipse releases?**
The results to this question can help the API providers to estimate a priori how much work could be involved in performing the promotion.

Contributions of our work are twofold:

1) A dataset of 327K stable non-API methods that can be used by Eclipse interface providers as possible candidates for promotion. Instead of promoting individual non-API methods which might be too fine-grained, we identify groups of non-API methods in given classes that are stable together. These groups are presented as possible candidates promotion.
2) We have shown that it is possible to predict the stability of a non-API in subsequent Eclipse releases with precision 56%, recall 96%, AUC 92% and the F-measure 81%. We have also shown that the length of a method and the number of method parameters are very good predictors for the stability of the non-API method in successive Eclipse releases.

III. Experimental Setup

Our study is based on all the 19 major releases of Eclipse SDK downloaded from the Archive website [19]. Details of the releases are summarized in Table I.

To answer the RQs we extract the data using the `NiCad` clone detection tool [20]. We opt for `NiCad` as it has also been extensively validated in the past [21], [22], [23]. When `NiCad`

Listing 1: Nicad Cross Clone Report

```
<clones>
    <clone nlines="74"  similarity ="100">
        public void md1(Cl c1) {<source File="E_1.0/org/ eclipse /p1/sp1/pk1/F1.java">
        public void md1(Cl c1) {<source File="E_4.6/org/ eclipse /p2/sp2/pk2/F2.java">
    </clone>
    <clone nlines="79"  similarity ="100">
        void m2(){<source File="E_1.0/org/ eclipse /p3/ internal /sp3/pk3/F3.java">
        void m2(){<source File="E_4.6/org/ eclipse /p4/ internal /sp4/pk4/F4.java">
    </clone>
    <clone nlines="90"  similarity ="100">
        public C1 m3(){<source File="E_1.0/org/eclipse /p5/ internal /sp5/pk5/F5.java">
        public C1 m3(){<source File="E_4.6/org/eclipse /p6/sp6/pk6/F6.java">
    </clone>
</clones>
```

TABLE I: Eclipse major releases and their corresponding release dates

Major Releases	Release Date	Major Release	Release Date
E-1.0	07-Nov-01	E-3.7	13-Jun-11
E-2.0	27-Jun-02	E-3.8	08-Jun-12
E-2.1	27-Mar-03	E-4.0	27-Jul-10
E-3.0	25-Jun-04	E-4.1	20-Jun-11
E-3.1	27-Jun-05	E-4.2	08-Jun-12
E-3.2	29-Jun-06	E-4.3	05-Jun-13
E-3.3	25-Jun-07	E-4.4	06-Jun-14
E-3.4	17-Jun-08	E-4.5	03-Jun-15
E-3.5	11-Jun-09	E-4.6	06-Jun-16
E-3.6	08-Jun-10		

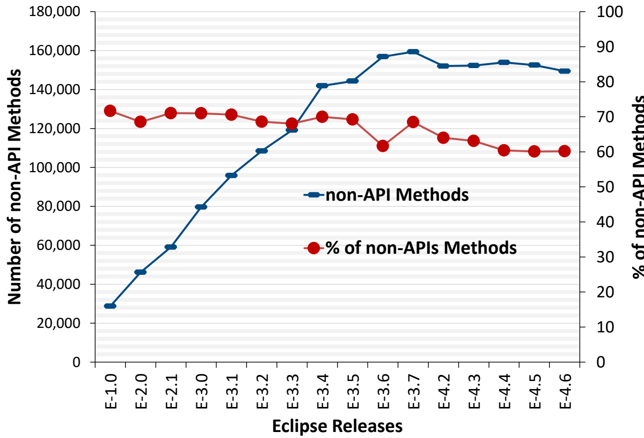

Fig. 1: Evolution of the number of non-API methods (left y-axis) and their percentage (right y-axis).

is applied to the collection of the 19 releases of Eclipse, in addition to detecting clones it generates an XML report with a list of all methods in any given Eclipse release. Since non-API methods can be distinguished from API methods by having a substring `internal` in their fully qualified name, we counted the number of methods with substring `internal` in the XML report. We used this report to obtain the total number of non-API methods in each Eclipse release.

Figure 1 superimposes two charts–evolution of the number of non-API methods and of the percentage of non-API methods. The percentage of non-API methods shows a slow decrease. We determine the percentage of the non-API methods in a given Eclipse release as a ratio of the non-API methods to the total interfaces (i.e., APIs plus non-API methods). The number of non-API methods per releases shows a non-linear increasing trend in the different subsequent Eclipse releases.

IV. RQ1: Stability of Internal Interfaces

With RQ1, we want to investigate if non-API methods released in earlier Eclipse releases remain stable in later Eclipse releases.

A. Data extraction for RQ1

Often later releases of Eclipse contain interfaces that were introduced in the earlier Eclipse releases. Therefore, before carrying out the `NiCad` clone detection, we first eliminate the old non-API methods in the new Eclipse release by locating the unchanged non-API in E_{new} that were previously introduced E_{old}. This leaves only non-API methods newly introduced in E_{new}.

We illustrate the data collection for RQ1 using Figure 2 and Listing 1. For a given Eclipse release—R4, in Step 1 of Figure 2 we identify files from R4 absent from earlier releases R1–R3, i.e., the shaded area R4'. Next, in Step 2 we subject R4' and a later Eclipse release, say R5, to `NiCad` cross-project clone detection. `NiCad` reports clones similarly to the report in Listing 1. Using different `NiCad` configurations, the tool is able to produce Type-1 and Type-2 clone reports.

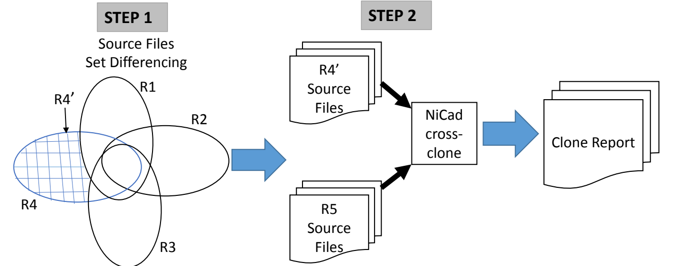

Fig. 2: Illustration diagram of data collection of RQ1.

Listing 1 is an example of fragment of the XML clone report generated by `NiCad` cross-project clone detection for Eclipse

TABLE II: The number of stable non-APIs methods in subsequent Eclipse releases identified through clone detection–Type-1 and Type-2

	E-1.0		E-2.0		E-2.1		E-3.0		E-3.1		E-3.2		E-3.3		E-3.4		E-3.5		E-3.6	
Total	28,732		46,156		59,075		79,630		95,868		108,455		119,153		141,915		144,353		156,893	
New	28,732		30,993		19,011		42,281		30,094		27,344		19,992		28,852		19,603		14,785	
	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2
E-2.0	2504	2744																		
E-2.1	2007	2264	3523	3865																
E-3.0	1052	1308	1529	1829	1229	1420														
E-3.1	731	914	968	1164	658	790	4269	4562												
E-3.2	544	673	770	933	511	631	3082	3391	3086	3256										
E-3.3	487	606	673	834	437	548	2739	3044	2533	2752	3659	3830								
E-3.4	475	590	636	783	419	523	2523	2848	2294	2509	3142	3331	2603	2708						
E-3.5	436	541	618	755	404	498	2334	2649	2088	2287	2865	3053	2214	2378	2421	2640				
E-3.6	381	491	601	740	388	482	2254	2577	2038	2271	2660	2852	2075	2247	2049	2219	2677	2757		
E-3.7	366	473	559	689	350	429	2020	2320	1768	1986	2404	2590	1966	2135	1765	1917	2431	2515	2047	2067
E-4.2	331	420	471	579	297	368	1774	2060	1656	1875	2253	2439	1805	1971	1682	1835	2274	2382	1398	1422
E-4.3	327	414	454	557	290	360	1736	1997	1599	1811	2193	2366	1659	1813	1554	1713	2173	2270	1327	1348
E-4.4	324	405	447	542	271	326	1583	1817	1417	1623	2034	2197	1529	1677	1391	1522	1935	2016	1142	1171
E-4.5	317	395	428	519	258	311	1477	1691	1369	1557	1906	2056	1439	1584	1287	1408	1861	1940	1064	1088
E-4.6	295	357	411	492	233	283	1320	1490	1261	1414	1655	1778	1268	1409	1166	1264	1725	1801	908	922
Mean	705	840	863	1,020	442	536	2,259	2,537	1,919	2,122	2,477	2,649	1,840	1,991	1,664	1,815	2,154	2,240	1,314	1,336
Min	295	357	411	492	233	283	1,320	1,490	1,261	1,414	1,655	1,778	1,268	1,409	1,166	1,264	1,725	1,801	908	922
Median	436	541	610	748	388	482	2,137	2,449	1,768	1,986	2,329	2,515	1,805	1,971	1,618	1,774	2,173	2,270	1,235	1,260
Max	2,504	2,744	3,523	3,865	1,229	1,420	4,269	4,562	3,086	3,256	3,659	3,830	2,603	2,708	2,421	2,640	2,677	2,757	2,047	2,067

TABLE III: non-APIs that have remained stable over subsequent Eclipse releases–candidates of promotion summarized at class level.

	E-1.0		E-2.0		E-2.1		E-3.0		E-3.1		E-3.2		E-3.3		E-3.4	
	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2
E-2.0	130//704	146//765														
E-2.1	97//593	109//668	208//1,229	233//1,308												
E-3.0	50//384	58//473	72//661	90//774	76//413	94//459										
E-3.1	34//291	40//368	34//482	40//573	35//277	45//325	244//1,380	266//1,449								
E-3.2	26//216	32//271	25//408	29//481	23//236	34//284	161//1,126	181//1,215	212//891	224//913						
E-3.3	21//202	28//256	20//360	24//435	19//213	25//258	146//1,043	167//1,134	177//782	195//813	241//1,085	256//1,112				
E-3.4	21//196	28//250	19//344	22//414	17//206	21//250	126//982	144//1,085	149//740	164//778	195//988	213//1,023	153//791	163//801		
E-3.5	18//182	26//233	19//333	22//399	17//199	21//240	111//922	131//1,020	136//692	149//739	170//927	187//965	126//717	141//739	150//726	162//768
E-3.6	15//166	24//217	18//324	21//392	17//191	21//234	108//911	130//1012	127//683	142//734	156//892	174//932	115//683	131//710	121//636	130//677
E-3.7	14//157	23//206	17//315	19//375	12//182	17//218	87//855	111//957	104//645	117//695	140//855	156//893	103//655	120//682	97//590	107//630
E-4.2	12//139	22//180	12//276	13//331	9//158	12//194	72//780	95//881	97//611	111//663	132//809	147//849	91//615	107//644	89//568	101//607
E-4.3	12//139	22//179	11//267	12//321	9//156	12//191	71//766	92//861	95//596	106//649	126//795	140//831	83//588	97//619	78//556	91//595
E-4.4	12//136	22//174	10//263	11//314	7//149	9//178	63//716	80//806	83//550	92//605	115//757	125//794	72//558	86//589	65//522	75//556
E-4.5	12//132	22//166	10//253	11//300	7//142	9//171	55//676	71//758	80//533	89//584	103//726	113//759	67//532	80//561	62//486	68//521
E-4.6	11//125	19//156	10//243	11//288	6//134	7//162	45//635	55//708	71//512	75//555	78//675	86//708	54//486	66//520	56//458	61//489

// – Symbol is used as a separator of two numbers. We explain how it is used in the text.

releases 1.0 and 4.6. In the first clone pair in Listing 1, both methods are from APIs (no segment `internal` in the source file paths). In the second clone pair both methods are from non-API methods. This clone pair indicates that the method `p4.internal. sp4.pk4.F4.m2()` that was introduced in Eclipse 1.0 and is still unchanged in the Eclipse release 4.6.

B. Results

Tables II and III present the results of RQ1, i.e., the number of stable non-API methods in a given Eclipse release observed in subsequent Eclipse releases. The first row of Table II shows the different Eclipse old releases (E_{old}) where we carried `NiCad` cross-project clone detection with the corresponding successive Eclipse new releases (E_{new}), in the first column. Due to space limitations, in the first row, we present the results from Eclipse 1.0 to 3.6. However, the remaining Eclipse releases show similar figures. The second row–*Total* shows the total number of non-API methods in the different Eclipse releases: e.g., E-2.0 has a total of $46,156$ non-API methods. The third row–*New* shows the number of newly introduced non-API methods in the different Eclipse releases: e.g., E-2.0 has a total of $30,993$ newly introduced non-API methods during its release and the rest $(46,156 \quad 30,993 = 15,163)$ evolved with Eclipse from earlier releases. The rest of the values in the matrix report the number of non-API methods of the newly introduced non-API methods in the given Eclipse releases that remained unchanged in successive Eclipse releases. For example, the value (E-2.1, E-2.0–T1) = $3,523$ indicates that among $30,993$ non-API methods introduced in E-2.0, $3,523$ stayed unchanged (Type-1) in E-2.1. The last four rows of Table II present the descriptive statistics of stability of the non-API methods. From the results presented in Table II, we observe that a few of the non-API methods that were newly introduced in a given Eclipse release still exist and have remained unchanged in subsequent Eclipse releases. In Table II, in the summary statistics in the last four rows, we can observe high values on unchanged newly introduced non-API methods of given current Eclipse releases in successive Eclipse releases. Take for example the results of Eclipse 1.0, after 15 years of the evolution of Eclipse framework producing a major release every year in Table II in cell (E-1.0, E-4.6) we still find 1.02%–Type 1 and 1.24%–Type 2 of the $28,732$ have remained stable. In Eclipse-E-3.5, we observe that in the last Eclipse release–E-4.6 there still exists over 8.8%–Type 1 and 9.19%–Type 2 of the $19,603$ non-API methods newly introduced in E-3.5 that have not been changed. Looking at row-E4.6 in Table II we can see the numbers of old non-API methods still present in Eclipse 4.6. For example Type 1 E-4.6–295, 411, 233, $\ldots$, shows the number of old non-API methods from Eclipse release–1.0, 2.0, 2.1, $\ldots$, respectively. The 295 non-API methods in E-1.0 present in E-4.6 indicate

that they have never been changed throughout the evolution of Eclipse.

In Table II we have presented stable non-API methods that are candidates of promotion to APIs. However, we understand that non-API methods are too fine grained to be considered as candidates of promotion. To this end, we present a summarized version of Table II in Table III. For example, in Table III cell (E-2.1, E-2.0-T2) = $233//1,308$ is a class-level summary of the non-API methods in cell (E-2.1, E-2.0-T2) = $3,865$ of Table II. The value $1,308$ in cell (E-2.1, E-2.0-T2) = $233//1,308$ indicates that the $3,865$ non API methods are contained in a total of $1,308$ non-API classes. Whereas the value 233 in (E-2.1, E-2.0-T2) = $233//1,308$ indicates that 233 of the $1,308$ non-API classes contain five of more non-API methods. Due to space limitations, in Table III we only present the results of Eclipse 1.0, ..., 3.4.

The detailed lists of stable non-API methods in different Eclipse releases is available on-line[2].

C. RQ1-Findings

The main reason for Eclipse interface providers to label the interfaces as non-API methods is the expectation of evolutionary changes. Once a non-API is through these evolutionary changes it is supposed to be promoted to an API [24].

To understand why some non-API methods remain unchanged for a very long time during the evolution of the framework, we contacted the Director of Open Source Projects–The Eclipse Foundation, Wayne Beaton. According to Mr. Beaton promotion of the non-API methods to APIs is something that the project teams decide for themselves. He also believes that "internal API oftentimes becomes official API when somebody steps up to do the work to make it so".

High number of stable non-API methods in Table II suggests that promotion indeed does not happen often. In addition to lack of developers interested in the promotion task suggested by Mr. Beaton, lack of promotion can be attributed to presence of developers dependent on non-API methods. However, at the same time plug-in developers might prefer to avoid these non-API methods since they are advertised as being unstable and likely to change. Hence, a developer intending to carry out such a promotion should balance the costs incurred on the developers making use of a non-API and the potential gains obtained by engaging more developers that might have been reluctant to use the non-API (cf. [25], [26]).

We identify methods in the same classes that have been stable for a long time. For example in Table II, the in cell (E-4.6, E-1.0-T1) = 295 non-API methods introduced in E-1.0 and have not changed (Clone Type-1) until E-4.6. We identify these stable non-API methods as candidates of promotion. We understand that Eclipse providers normally perform non-API promotion at the level of a `class`. Promotion at the `method` level seems to be fine grained. We have therefore summarized the non-API methods that are possible candidates for promotion to APIs to the class-level. Instead of promoting a single non-API method, the providers can, hence, promote groups of non-API methods that are stable together in a single class.

[2]https://sites.google.com/view/eclipse-saner-2019/home

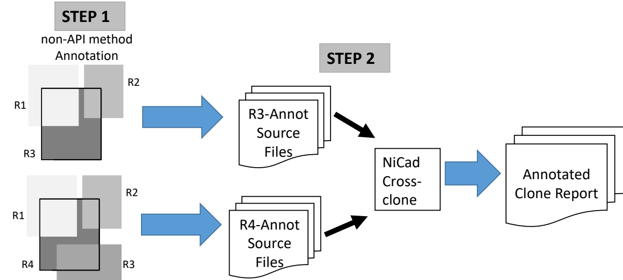

Fig. 3: Illustration diagram of data collection of RQ2.

V. RQ2: Predicting non-API stability

Next we want to predict whether non-API methods present in a given Eclipse release will remain stable in the subsequent Eclipse releases. We want to investigate possible factors that relate to the stability of the non-API methods. We hypothesize that in a given Eclipse release, *length* of the methods, the *number of parameters* of the method signature and the *age* of the non-API methods could have an influence on the stability of the non-API methods in the next Eclipse release.

A. Approach

We want to compare the importance of multiple metrics of methods in a given Eclipse release. For the dependent variable, we consider non-API stability–`Stability`. `Stability` on a non-API method in the subsequent Eclipse release can take one of two values—stable or unstable.

As independent variables we consider metrics of lengths of a methods–`mLengths`, number of methods parameters–`mParams`, age of a method–`Age` on the non-API. We want to build a random-forest classifier to predict whether an internal interface will be promoted, given the values of the metrics. We choose the random-forest classifier because it is known to have several advantages, such as being robust to noise and outliers [27], [15]. In selecting the training and testing set, we rely on the built-in internal evaluation of performance of random forests based on **out-of-bag dataset** [28]. In the implementation of the random forest algorithm, each tree is trained on about 2/3 of the total training data also known as the *bootstrapped* data. As the forest is built, each tree can thus be tested on the *out-of-bag* data (similar to leave one out cross validation) on the samples not used in building that tree.

To assess the effectiveness of the classifier in predicting stability of the non-API methods we use such common measures as precision, recall, F-measure, and AUC [29], [27], [15], [30], [31].

B. Data extraction

As explained earlier, a newer Eclipse release is composed of newly introduced non-API methods as well as non-API methods carried over from older Eclipse releases. We illustrate the different steps we used collect the data for RQ2

Listing 2: Illustration of method signatures in Eclipse 4.6 before annotation

```
org/ eclipse /core/ internal /runtime/ Activator  public  Location   getInstallLocation  ():199:211
org/ eclipse /core/ internal / resources /XMLWriter private  static   String  getReplacement(char  c):112:128
org/ eclipse /pde/ internal /core/product/Product  public  void  addPlugins( IProductPlugin []  plugins ):408:425
org/ eclipse /pde/ internal /ui/ editor / feature / LicenseFeatureSection   public  boolean  doGlobalAction( String   actionId ):354:377
```

Listing 3: Illustration of method signatures in Eclipse 4.6 after annotation

```
org/ eclipse /core/ internal /runtime/ Activator  public  Location  getInstallLocation_E4_2 ():199:211
org/ eclipse /core/ internal / resources /XMLWriter private  static   String  getReplacement_E3_6(char c):112:128
org/ eclipse /pde/ internal /core/product/Product  public  void  addPlugins_E3_5(IProductPlugin []  plugins ):408:425
org/ eclipse /pde/ internal /ui/ editor / feature / LicenseFeatureSection   public  boolean  doGlobalAction_E4_6(String  actionId ):354:377
```

Listing 4: Nicad Clone cross-Report between the annotated Eclipse Releases of 4.2 and 4.6

```
<clones>
    <clone nlines="79"  similarity ="100">
        void m4_E1_0(C1 cls, int x){<source="E_4.2/org/eclipse /p4/ internal /sp4/pk4/f4 . java">
        void m4_E1_0(C1 cls, int x){<source="E_4.6/org/eclipse /p4/ internal /sp4/pk4/f4 . java">
    </clone>
    <clone nlines="90"  similarity ="100">
        public  C2 m5_E2_0(){<source="E_4.2/org/eclipse/p6/ internal /pk6/f6 . java">
        public  C2 m6_E2_6(){<source="E_4.6/org/eclipse/p6/ internal /pck6/f6 . java">
    </clone>
    <clone nlines="90"  similarity ="100">
        public  int  m8_E4_6(){<source="E_4.2/org/eclipse/p8/ internal /pk8/f7 . java">
        public  int  m8_E4_6(){<source="E_4.6/org/eclipse/p8/ internal /pk8/f8 . java">
    </clone>
</clones>
```

using Listings 2, 3, and Figure 3. Using the `NiCad` tool, from sources of the different Eclipse releases, we extracted signatures of non-API methods, the fully qualified name of Java files, `start-line` and `end-line` of the method in the file. Using set differencing on the lists of non-API method signatures, we isolate the newly introduced non-API methods in each Eclipse release.

In Step 1 of Figure 3, we automatically annotate non-API methods in each `.java` file in a given Eclipse release with the Eclipse release where these methods have been introduced. For example in Step 1, the release of interest R4 has methods that were introduced in R1–R3 as well as those newly introduced in R4. After annotating the non-API methods in R4 that are uniquely identified from which Eclipse release they were introduced, the output is `R4-Annot`. In Listing 2 we present an extract of four method signatures from Eclipse release 4.6 before method annotation. The methods in the listing show the fully qualified methods name, its parameter list, the method start line–number after the first full colon and end line–number after the second full colon. In Listing 3 we show the corresponding methods after the annotation. The non-API method annotations are on the method name–E4_2, E3_6, and E3_6 and E4_6). For example, `getInstallLocation_E4_2` tells us that the method `getInstallLocation` was introduced in Eclipse 4.2 but still present in Eclipse 4.6.

For building the models, we considered data from Eclipse releases 4.2, 4.3, 4.4, 4.5, and 4.6. Looking at Listing 3, we can determine the values for the metrics `mLengths`—difference between method end-line and start-line, `mParams`—count of the method parameters. For the `Age` metric, newly introduced non-API methods in Eclipse release under study have an `Age` of zero, those introduced in previous release `Age` of one, etc. For example in Eclipse 4.2, the newly introduced non-API methods in 4.2 would have zero years, those introduced in 3.7 would have one year and those introduced in 1.0 would be 11 years.

Extracting data for dependent variable–`Stability`, we subject two annotated Eclipse releases to `NiCad` cross-project (see Step 2 in Figure 3). The output of Step 2 is an annotated clone report in XML format. We configure `NiCad` cross-project to extract Type-2 and Type-3 clone pairs between two Eclipse releases. We do not extract Type-1 clones since the method annotations eliminate the occurrences of Type-1. In Listing 4 we present an XML file that is an illustration of the output of the `NiCad` cross-project between Eclipse 4.2 and the target Eclipse 4.6. Considering for example the total number of non-API methods in Eclipse 4.2, one is able to determine those that remained stable in the target Eclipse 4.6—*those that appear in the annotated clone report* and the unstable non-API methods—*the rest of the non-API methods in Eclipse 4.2 not*

TABLE IV: Stability prediction results (percentages) of non-API between a pair of Eclipse releases.

			4.2				4.3				4.4				4.5			
			Prec	Rec	F-m	AUC	Prec	Rec	F-m	AUC	Prec	Rec	F-m	AUC	Prec	Rec	F-m	AUC
Target	4.3	T2	82	96	86	97												
		T3	86	97	91	98												
	4.4	T2	71	98	82	95	73	99	84	95								
		T3	83	98	90	97	82	99	89	95								
	4.5	T2	65	97	78	93	67	99	80	94	75	99	85	96				
		T3	80	97	88	97	79	99	88	95	82	99	90	97				
	4.6	T2	56	98	71	92	57	99	72	92	66	97	79	93	70	99	82	94
		T3	78	97	86	96	76	99	86	94	78	99	87	96	82	98	90	97

[T2] Type-2 Clones
[T3] Type-3 Clones

TABLE V: Stability prediction results (percentages) of non-API between a pair of Eclipse releases.

			4.2			4.3			4.4			4.5		
			L	P	A	L	P	A	L	P	A	L	P	A
Target	4.3	T2	98	2	0									
		T3	86	14	0									
	4.4	T2	89	11	0	97	3	0						
		T3	99	1	0	64	36	0						
	4.5	T2	86	14	0	90	10	0	80	20	0			
		T3	99	1	0	92	8	0	75	25	0			
	4.6	T2	96	4	0	91	9	0	96	4	0	78	22	0
		T3	74	26	0	98	2	0	74	26	0	97	3	0

[L] `mLength`
[P] `mParams`
[A] `Age`

in the clone report.

We extracted cross-project clone reports for Type-2 and Type-3 between pairs of Eclipse releases 4.2 to 4.6. A total of 20 cross-project clone reports were obtained from the considered Eclipse releases. Thereafter, we carried out random forest predictions of the stability of non-API methods between all the 20 pairs of Eclipse releases. We used a commercial tool called *Salford Predictive Modeler*[3] to perform the predictions. In the tool, some of the key controls of the random forest process were set as follows: number of trees to 200, the number of predictors considered for each tree node to two and the parent node minimum cases to two records.

C. Model Evaluation

To evaluate our models, we assess the effectiveness of the classifier in correctly predicting stability of the new Eclipse releases. We use precision, recall, F-measure and AUC (area under curve) to measure its effectiveness, which are commonly adopted in classification tasks [32], [27]. Precision and recall measure the correctness and completeness, respectively, of the classifier in predicting whether a non-API is stable. F-measure is the harmonic mean of precision and recall. AUC is a commonly used measure to judge predictions in binary classification problems, and it refers to the area under the Receiver Operating Characteristic (ROC) curve. AUC is robust toward unbalanced data [33]. AUC of 70% is considered reasonably good [34], [35], [27].

[3]https://www.salford-systems.com/

D. Results RQ2

The number of non-API methods present in Eclipse 4.2 to 4.5 range from 147K to 152K. The number of stable non-API methods in the 20 different pairs of Eclipse releases 4.2 to 4.6 for Clone Type-2 and Type-3 range from 33K to 40K (about 19 to 28% of the total target observations). This implies that the majority of the values in the dependent variable in the 20 different datasets are `class-0`–unstable interfaces. To ensure that our models are not suffering from over-fitting/under-fitting, we employed other model testing methods other than out-of-bag like: *test sample contained in a separate file* and *fraction of testing selected at random*. The two testing cases did not yield different results from the out-of-bag method. The predictions success results reveal that in solving the classification problem, the random forest model shows that the out-of-bag internal validation performance as being in the range of 82 to 94% correct in correctly classifying stability `class-0`–unstable interfaces and a range of 96 to 99% correct in correctly classifying `class-1`–stable interfaces.

Table IV and V present the prediction results of RQ2. Table IV shows effectiveness of the classifier in correctly predicting the non-API stability using precision, recall, AUC and F-measure. For example, the value in cell (4.4-T2, 4.2-Prec) = 71 indicates that the non-API methods present in Eclipse 4.2 were predicted to be stable/unstable in the target Eclipse release 4.4 with a precision of 71%.

The results show that the precision is between 56%–86%, recall 96%–99%, F-Measure 81%–89%, and AUC 92%–98% for all the data of clone Type-2 & 3. From the results presented we observe relatively high prediction values for

all the considered measures. We observe very high recall compared to the precision. *Precision* is the ratio of correctly predicted positive observations to the total predicted positive observations (i.e., $Precision = TP/(TP + FP)$). *Recall* is defined as the ratio of correctly predicted positive observations to the total predicted positive observations (i.e., $Recall = TP/(TP + FN)$). The reason we have very high values of recall is that, as stated earlier, the prediction success of the class-1 was very high (i.e., all the models revealed very few cases of incorrectly classified stable non-API methods–FN). In Software Engineering domain the value of AUC between 70% is considered a reasonably good measure [15], [27], [30], [35].

In Table V we show the predictor importance that summarizes how the individual variables have contributed to the prediction accuracy. The *Gini importance* was used to indicate how large a variable's overall discriminative value was for the classification problem. We observe that the predictor `mLength` of the non-API methods is the most influential followed by `Params` and finally `Age`. As can be seen, the variable `Age` is insignificant in separating the two target classes.

E. RQ2-Findings

Yes we can predict the stability of a non-API in subsequent Eclipse releases with a precision of 56% and higher, a recall of 96% and higher and an AUC of 92% and higher and an F-measure of 81% and higher. From the results presented in Section V-D we can observe that the number of parameters and the lengths of a non-API method have a significant effect in determining the non-API stability in new Eclipse releases. Developers who use the non-API methods in their applications can use our dataset to estimate the stability of the non-API methods in new Eclipse releases. We have also observed that the age of a non-API has an insignificant impact on the stability of a non-API in a new Eclipse release. Our observation contradicts with the earlier observation of Businge et al. [10], [9] that older non-API methods are more stable than the more recently introduced ones.

VI. Threats to Validity

We observe a *construct validity* threat related to the results of RQ1. An object-oriented API may much more complex than it is assumed in the paper. To use the functionality provided by an API, in some cases one be required to instantiate an object and call two or more methods from this object.

Furthermore, we determine non-API stability using clone Type 1 & 2. Using Type 2 clones could threaten the construct validity since the changes in the stable non-API determined by clone Type 2 might actually make the non-API unstable.

Internal validity threat related to the tool (i.e., NiCad) used to extract the data used in our experiments. However, studies who have compared NiCad tool with other clone detection tools have observed that NiCad tool has the highest precision and recall of any existing code clone detector tools [22], [36].

Threats to *external validity* concern the possibility to generalize our results. Our study focus on only one framework. Validation on other frameworks and libraries developed in the same setting like Eclipse like those we presented in Section I (i.e., JDK, MSDN, jBPM and JUnit) is desirable.

VII. Related Work

In the previous sections, we implicitly discussed how the current work relates to the previous work [12], [11], [9], [37], [10], [13]. In general, the previous studies 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 [13] 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 no alternative stable interfaces offering the same functionality. This study was based on understanding the evolutionary trend followed by the Eclipse non-APIs in successive Eclipse releases. We use code clone detection analysis to carry out our investigation.

Another study that is directly related to ours is that of Hora et al. [15] who 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 the 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 an 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. However, in comparison to our study, we use clone detection techniques on the Eclipse releases to determine stable internal interfaces that we recommend as possible promotion candidates.

Other work related to ours includes [38], [39], [40], [41], [42], [43], [44], [45], [46] Sawant et al. [38] studied the effects of deprecation of Java API artifacts on their clients. Their work expands upon a similar study done on the Smalltalk ecosystem. The main differences between the two studies is in the type systems of the language targeted (static type vs dynamic type), the scale of the dataset (25,357 vs 2600 clients)

and the nature of the dataset (third- party APIs vs third-party and language APIs). They found that few API clients update the API version that they use. In addition, the percentage of clients that are affected by deprecated entities is less than 20% for most APIs—except for Spring where the percentage was unusually low. In the case of the JDK API, they saw that only 4 clients were affected, and all of them were affected by deprecation because they introduced a call to the deprecated entity at the time it was already deprecated, thereby limiting the probability of a reaction from these clients.

Wu et al. [39] analyzed and classified API changes and usages in 22 framework releases from the `Apache`, `Eclipse` ecosystems and their client programs. They discovered that framework APIs are used on average in 35% of client classes and interfaces and that about 11% of APIs usages could cause ripple effects in client programs when these APIs change. The authors also found that missing classes and methods happen more often in frameworks and affect client programs more often than the other API change types do. Mastrangelo et al. [40] discovered that client projects often use internal interfaces provided by JDK. Brito et al. [41] introduced APIDIFF, a tool to identify API breaking and non-breaking changes between two versions of a Java library. The tool detects changes on three API elements: types, methods, and fields. We also report usage scenarios of APIDIFF with four real-world Java libraries. McDonnell et al. [42] investigated API stability and adoption on Android ecosystem. They discovered that Android is evolving fast with an average of 115 API updates per month but developers are resistant to embrace unstable fast evolving APIs quickly. They found out that 28% of Android references in client code are out-of-date with a median lag of 16 months. Dig and Johnson [43] studied the role of refactorings in API evolution and found that 80% of the changes that break client applications are API-level refactorings.

Henkel et al. [44] propose CatchUp, a tool that uses an IDE to capture and replay refactorings related to API evolution. Hora et al. [47], [48] propose tools to keep track of API evolution by mining fine-grained code changes. Hou [49] studied the evolution of Eclipse Java editor by exploring the Eclipse source code for six releases. He discovered that although there are changes to the design of individual features, architecturally, the editor benefits from the MVC based design laid out in the outset of the project. Hou and Wang [50], [51] analyzed release note entries in seven releases of the Eclipse IDE both quantitatively and qualitatively. The authors found that majority of the changes were refinements or incremental additions to the feature architecture set up in early releases.

API evolution has also been studied for many other platforms. Jezek et al. [52] investigated the API changes and their impacts on Java programs. They found out that API instability is common and will eventually cause problems. Hora et al. [45] studied how developers react to API evolution for the Pharo system, they discovered that API evolution can have a large impact on a software ecosystem in terms of client systems, methods, and developers. Hou and Yao [46] explored the intent behind API evolution by by analyzing the evolution of a production API in detail. The authors discovered that a large part of API evolution is minor correctives.

VIII. Conclusion and Future Work

In this study we have carried out an extensive investigation on the evolution of Eclipse non-APIs. 1) In RQ1, we observed that indeed there exist non-API methods that remain stable subsequent Eclipse releases during the evolution of Eclipse. 2) In RQ2, We have shown that the metrics of length of a method and number of method parameters in a non-API method are good predictors for the stability of the non-API method in subsequent Eclipse releases. We have observed that age of an non-API is not a significant predictor of the non-API's stability in subsequent framework releases.

We determined non-API stability in subsequent Eclipse releases using clone detection techniques. Our recommendations/contributions to Eclipse interface providers are as follows:

1) We provide a dataset of 327K stable non-API methods that can be used by both Eclipse interface providers as possible candidates of promotion. Instead of promoting non-APIs which are too fine-grained, we summarized of non-API methods groups in given classes that are stable together and presented class-level non-APIs that possible candidates promotion. The Eclipse providers can use our data as a starting point to analyze and promote stable non-APIs to APIs.
2) We have shown that it is possible to predict the stability of a non-API in subsequent Eclipse releases with a precision of 56%, a recall of 96% and an AUC of 92% and an F-measure of 81%. We have also shown that the metrics of length of a method and number of method parameters in a non-API method are very good predictors for the stability of the non-API in successive Eclipse releases. The results provided can help the API providers to estimate a priori how much work could be involved in performing the promotion.

As future work we would extend the current study by 1) gathering data about how often each stable non-API methods is used (popularity), and 2) carrying out a survey with the interface providers to investigate why non-APIs remain unstable for a long time. In the survey, we shall also get the opinion of the developers on how best our work can be packaged for them the dataset for non-APIs that are candidates for promotion.

Acknowledgement

This work was supported by the Sida/BRIGHT project under the Makerere-Sweden bilateral research programme 2015-2020. We would like to thank Prof. Thorsten Berger who helped us with proofreading our manuscript.

References

[1] T. Tourwe and T. Mens, "Automated support for framework-based software," in *ICSM*, Sept 2003, pp. 148–157.
[2] D. Konstantopoulos, J. Marien, M. Pinkerton, and E. Braude, "Best principles in the design of shared software," in *COMPSAC*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 287–292.

[3] S. Moser and O. Nierstrasz, "The effect of object-oriented frameworks on developer productivity," *Computer*, vol. 29, no. 9, pp. 45–51, Sep 1996.

[4] J. des Rivières, "How to use the Eclipse API," http://www.eclipse.org/articles/article.php?file=Article-API-Use/index.html, consulted October, 2018.

[5] Oracle, "Why developers should not write programs that call 'sun' packages," https://www.oracle.com/technetwork/java/faq-sun-packages-142232.html, consulted October, 2018.

[6] T. jBPM Team, "The jbpm api," http://docs.jboss.org/jbpm/v5.0/userguide/ch05.html#d0e2099, consulted October, 2018.

[7] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, and C. Stein, "Junit 5 user guide," https://junit.org/junit5/docs/current/user-guide/#api-evolution, consulted October, 2018.

[8] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *ICSE*. New York, NY, USA: ACM, 2008, pp. 481–490.

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

[10] ——, "Compatibility prediction of Eclipse third-party plug-ins in new Eclipse releases," in *SCAM*, 2012, pp. 164–173.

[11] J. Businge, M. G. J. van den Brand, and A. Serebrenik, "Eclipse API usage: The good and the bad," *Software Quality Journal*, vol. 23, pp. 107–141, 2013.

[12] 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.

[13] ——, "Analyzing the Eclipse API usage: Putting the developer in the loop," in *CSMR*, 2013, pp. 37–46.

[14] S. Kawuma, J. Businge, and E. Bainomugisha, "Can we find stable alternatives for unstable eclipse interfaces?" in *ICPC*. IEEE, 2016, pp. 1–10.

[15] A. Hora, M. T. Valente, R. Robbes, and N. Anquetil, "When should internal interfaces be promoted to public?" in *FSE*. ACM, 2016, pp. 278–289.

[16] J. des Rivières, "Evolving Java-based APIs," http://wiki.eclipse.org/Evolving_Java-based_APIs, consulted October, 2018.

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

[18] R. Koschke, "Large-scale inter-system clone detection using suffix trees and hashing," *Journal of Software: Evolution and Process*, vol. 26, no. 8, pp. 747–769, 2014.

[19] E. Project, "Eclipse project archived download," http://archive.eclipse.org/eclipse/downloads/index.php, consulted January, 2018.

[20] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *ICPC*, 2011, pp. 219–220.

[21] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *ICSME*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 321–330.

[22] ——, "Evaluating clone detection tools with bigclonebench," in *ICSME*, Sept 2015, pp. 131–140.

[23] C. K. Roy and J. R. Cordy, ": Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC*, June 2008, pp. 172–181.

[24] Eclipse, "Provisional api guidelines update proposal," https://wiki.eclipse.org/Provisional_API_Guidelines_Update_Proposal, consulted October, 2018.

[25] C. Bogart, C. Kästner, J. D. Herbsleb, and F. Thung, "How to break an API: cost negotiation and community values in three software ecosystems," in *FSE*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds., 2016, pp. 109–120.

[26] M. Valiev, B. Vasilescu, and J. D. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 644–655.

[27] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? a case study on free android applications," in *ICSME*, 2015, pp. 301–310.

[28] S. Janitza, "On the overestimation of random forest's out-of-bag error," Department of Medical Informatics, Biometry and Epidemiology, University of Munich, Tech. Rep. 204, January 2017.

[29] J. Businge, M. Openja, D. Kavaler, E. Bainomugisha, F. Khomh, and V. Filkov, "Studying android app popularity by cross-linking github and google play store," in *SANER*, 2019.

[30] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," *CoRR*, vol. abs/1502.06757, 2015. [Online]. Available: http://arxiv.org/abs/1502.06757

[31] 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 *PROMISE*, 2017.

[32] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70773

[33] F. Provost and T. Fawcett, "Robust classification for imprecise environments," *Machine Learning*, vol. 42, no. 3, pp. 203–231, 2001. [Online]. Available: https://doi.org/10.1023/A:1007601015854

[34] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, July 2008.

[35] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *WCRE*, 2012.

[36] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *ICSTW*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 157–166.

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

[38] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of clients of 4 + 1 popular java apis and the jdk," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2158–2197, Aug 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9554-9

[39] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2366–2412, 2016.

[40] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at your own risk: the java unsafe api in the wild," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 695–710, 2015.

[41] A. Brito, L. Xavier, A. C. Hora, and M. T. Valente, "Why and how java developers break apis," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, 2018, pp. 255–265. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330214

[42] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *ICSM*, 2013, pp. 70–79.

[43] D. Dig and R. Johnson, "The role of refactorings in api evolution," in *ICSM*. IEEE, 2005, pp. 389–398.

[44] J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support api evolution," in *ICSE*. ACM, 2005, pp. 274–283.

[45] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to api evolution? the pharo ecosystem case," in *ICSME*. IEEE, 2015, pp. 251–260.

[46] D. Hou and X. Yao, "Exploring the intent behind api evolution: A case study," in *WCRE*, Oct 2011, pp. 131–140.

[47] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "Apievolutionminer: Keeping api evolution under control," in *CSMR-WCRE*. IEEE, 2014, pp. 420–424.

[48] A. Hora and M. T. Valente, "apiwave: Keeping track of api popularity and migration," in *ICSME*. IEEE, 2015, pp. 321–323.

[49] D. Hou, "Studying the evolution of the eclipse java editor," in *OOPSLA Workshop on Eclipse Technology eXchange*. New York, NY, USA: ACM, 2007, pp. 65–69. [Online]. Available: http://doi.acm.org/10.1145/1328279.1328293

[50] D. Hou and Y. Wang, "Analyzing the evolution of user-visible features: A case study with eclipse," in *ICSM*, 2009, pp. 479–482.

[51] ——, "An empirical analysis of the evolution of user-visible features in an integrated development environment," in *CASCON*, 2009, pp. 122–135.

[52] K. Jezek, J. Dietrich, and P. Brada, "How java apis break–an empirical study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.