Eclipse API Usage: The Good and The Bad

John Businge
Eindhoven University of Technology
Eindhoven, The Netherlands
j.businge@tue.nl

Alexander Serebrenik
Eindhoven University of Technology
Eindhoven, The Netherlands
a.serebrenik@tue.nl

Mark van den Brand
Eindhoven University of Technology
Eindhoven, The Netherlands
m.g.j.v.d.brand@tue.nl

***Abstract*—Today, when constructing a new software system, many developers build their systems on top of frameworks. Eclipse is such a framework that has been in existence for over a decade and has so far released 11 major releases. Like many other evolving software systems, the Eclipse platform has both stable and supported APIs ("good") and unstable, discouraged and unsupported non-APIs ("bad"). However, despite being discouraged by Eclipse, in our experience, the usage of "bad" APIs is relatively common in practice. In this paper, we study to what extent developers depend on "bad" non-APIs. We also study whether developers continue to use "bad" APIs, and what are the differences between the third-party plug-ins that use non-APIs and those that do not. Furthermore, we also study the commonly used "bad" APIs.**

To answer these questions, we have conducted an empirical investigation based on a total of 512 Eclipse third-party plug-ins, altogether having a total of $1,873$ versions. We discovered that 44% of the 512 analyzed Eclipse third-party plug-ins depends on "bad" non-APIs and that developers continue to use "bad" non-APIs. The empirical study also shows that plug-ins that use or extend at least one "bad" non-API are comparatively larger and use more functionality from Eclipse than those that use only "good" APIs. Furthermore, the findings show that the third-party plug-ins use a diverse set of "bad" APIs.

***Keywords*-Eclipse; Third-party plug-ins; APIs; non-APIs;**

I. Introduction

Today, many software developers are building their systems on top of frameworks. This approach has many advantages, such as reuse of the functionality provided [1] and increasing productivity [2]. However, in spite of these benefits, there are potential challenges that come along, for both the framework developer and the developer who uses the functionality from the framework [3]. On the framework developer side, continuous evolution takes place as a result of refactoring and introduction of new functionality in the framework components [4]. A potential challenge as a result of performing these activities is that the framework developers have to refrain from changing the existing application programming interfaces (APIs) because such change may cause applications that depend on these APIs to fail [5]. In practice, for successfully and widely adopted frameworks, it may not be possible to achieve this fully. On the framework user side, if developers are to take advantage of the better quality and new functionality introduced as a result of the evolution of the framework, then the evolution of these software systems may be constrained by two independent, and potentially conflicting, processes [5], [6]: 1) evolution to keep up with the changes introduced in the framework (framework-based evolution); 2) evolution in response to the specific requirements and desired qualities of the stakeholders of the systems itself (general evolution).

To facilitate framework based evolution Eclipse SDK distinguishes between stable and supported APIs ("good") and unstable, discouraged and unsupported non-APIs ("bad") [7]. The latter are indicated by the substring `internal` in the package name. While "good" APIs can be safely used in Eclipse-based systems, the use of "bad" non-APIs is at risk of arbitrary change or removal without notice. It has been shown that many "bad" non-APIs are among packages that are most likely to introduce a post-release failure [8].

In this paper we present preliminary results on usage of Eclipse API and non-API. We investigate to *what extent* third-party plug-ins use non-APIs, whether developers *continue to use* them and what are the *differences* between the third-party plug-ins that use non-APIs and those that do not. We also investigate commonly used non-APIs by the plug-ins. The answers to these questions provide Eclipse SDK developers with feedback on the current use of APIs and non-APIs as opposed to the expected use. Furthermore, these answers also create a solid basis for continuation studies related to APIs, non-APIs and their use in third-party plug-ins.

To answer these questions, we conducted an empirical study based on a total of 512 Eclipse third-party plug-ins altogether having a total of 1,873 versions collected from SourceForge. The third-party plug-ins collected date back from 2003 to 2010. The number of versions considered requires a light-weight analysis approach (cf. [9]). Therefore, our analysis incorporates naming conventions and package imports rather than more precise but also more computationally challenging techniques such as call graph construction.

The remainder of the paper is organized as follows. In Section II we present the notion of Eclipse plug-ins and their interfaces. In Section III we present the methodologies used in data collection. In Sections IV and V we analyze metrics for the identified groups of Eclipse third-party plug-

ins. In Section VI we analyze commonly used non-APIs. In Section VII we discuss the threats to validity. In Section VIII we discuss the related work and finally, in Section IX we present the conclusions and future work.

II. Eclipse Plug-in Architecture

Eclipse SDK is an extensible platform that provides a core of services for controlling a set of tools working together to support programming tasks. Tool builders contribute to the Eclipse platform by wrapping their tools in pluggable components, called *Eclipse plug-ins*, which conform to the Eclipse's plug-in contract. Plug-ins are bundles of code and/or data that contribute functions to a software system. Functions can be contributed, e.g., in the form of code libraries, platform extensions or documentation.

We categorize the Eclipse plug-ins into three groups: Core plug-ins, Extension plug-ins, and Third-party plug-ins. *Eclipse Core Plug-ins (ECP)* are plug-ins present in and shipped as part of the Eclipse SDK. These plug-ins provide core functionality upon which all plug-in extensions are built. The plug-ins also provide the runtime environment in which other plug-ins are loaded, integrated, and executed. *Eclipse Extension Plug-ins (EEP)* are plug-ins built with the main goal of extending the Eclipse platform. Most EEPs are large, generic, applications frameworks with tool plug-ins to build other specialized applications. Popular EEPs include J2EE Standard Tools, Eclipse Modeling Framework and PHP Development Tools. Like the ECPs, fully qualified names of EEP packages start with `org.eclipse`, but as opposed to the ECPs, the EEPs are not shipped as part of the Eclipse SDK. Finally, *Eclipse Third-Party Plug-ins (ETP)* are the remaining plug-ins. All the ETPs use at least some functionality provided by ECPs but also may use functionality provided by EEPs.

Eclipse clearly states general rules for extending or using the services it provides. The Eclipse guideline defines two types of interfaces [7]: ECP non-APIs ("bad") and ECP APIs ("good"). In addition to Java access (visibility) levels, Eclipse has another level called internal implementations (*non-APIs*). According to Eclipse naming convention [7], the non-APIs are artifacts found in a package with the segment `internal` in a fully qualified package name. These artifacts in the package may include public Java classes, interfaces, public or protected methods, or fields in such a class or interface. Users are strongly discouraged from adopting any of the non-APIs since they may be unstable [10]. Eclipse clearly states that users 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. As opposed to non-APIs, ECP APIs ("good") are found in packages that do not contain the segment `internal`. The ECP APIs may include public Java classes or interfaces, public or protected methods, or fields in such class or interface. Eclipse considers these APIs to be stable and therefore can be safely used by any plug-in developer.

Total								
	2003	2004	2005	2006	2007	2008	2009	2010
2003	*91,240*	41,99	23,44	18,30	14,28	6,8	2,8	4,8
2004		*83,192*	20,57	16,40	10,26	11,25	4,8	5,11
2005			*88,192*	30,79	19,47	18,31	10,32	8,24
2006				*107,251*	26,72	8,35	3,13	11,28
2007					*73,177*	22,46	10,19	10,18
2008						*74,156*	18,36	4,8
2009							*47,72*	9,15
2010								*28,43*
Total	91,40	124,291	131,293	171,400	142,350	139,301	94,188	79,155
Clean								
	2003	2004	2005	2006	2007	2008	2009	2010
2003	*80,195*	37,87	21,39	16,27	12,25	6,8	2,8	3,6
2004		*77,171*	19,54	14,34	9,25	10,24	3,6	4,9
2005			*73,154*	23,60	14,40	12,21	9,23	8,24
2006				*97,223*	24,70	7,34	3,13	11,28
2007					*57,134*	18,37	8,17	6,12
2008						*62,119*	14,27	4,8
2009							*40,59*	8,13
2010								*26,39*
Total	80,195	114,258	113,247	150,344	116,294	115,243	79,153	70,139

Table I: Total numbers of ETPs collected, and the numbers of clean ETPs and their versions. Numbers of the ETPs that were first released in a certain year are typeset in italics.

III. Data Collection and Classification

For our analysis, we collected ETPs from SourceForge, one of the most popular Open Source repositories. Because we wanted to have long enough history of data, we chose SourceForge as opposed to recently popular repositories like google code and github. On the SourceForge search page, we typed the text "*Eclipse AND plugin*" and it returned 1,350 hits at the time we collected the data (February 16, 2011).

During the data collection, three major steps were considered to extract and classify the *useful ETPs* from the 1,350 hits. These are: downloading and initial classification, removal of incomplete ETPs, and final classification.

Manual download was the only way to obtain source code of the plug-ins considered, due to on the one hand the structure of SourceForge, and on the other hand differences in the packaging and source code storage of different ETPs. Since we wanted to have a long enough history of ETPs supported in the different ECPs and a substantial amount of data to draw sound statistical conclusions, we decided to collect ETPs that were released on SourceForge from January 1, 2003 to December 31, 2010. The ETPs for which none of the versions had source code were omitted. During the collection process, each ETP was categorized according to the year of its first release on SourceForge.

To reduce threats to validity as much as possible, we remove two groups of ETPs. First, exclude *incomplete* ETPs, i.e., ETPs having at least one version containing only

binaries but no source files. Second, we eliminate ETPs that do not import packages from ECPs. As mentioned earlier in Section II, ECPs and EEPs share a common prefix `org.eclipse`. To ensure that only the `import` statements from ECPs in the ETPs source code are considered, a list of all possible import statements from ECPs was compiled. The remaining ETPs are considered to be *clean*.

Table I shows us the overall summary of the data collected from SourceForge as well as data on clean ETPs. In the upper part of the table the cell entry (2004,2004), typeset in italics, contains a pair 83,192 indicating that there are a total of 83 ETPs that were first released in the year 2004 on SourceForge altogether having a total of 192 versions. The pair 20,57 in the cell (2004,2005) means that there were 20 ETPs of the 83 that had new versions released in the year 2005 with a total of 57 versions in that year. We observe that the trend on the evolution in the *total* number of ETPs is non-monotone, for example, (2004,2006)=16, (2004,2007)=10, (2004,2008)=11. The indicates that, while an ETP may have version(s) in a given year, it does not release any version(s) in the subsequent year(s) and resumes releasing later. Similarly, in the lower part of Table I 77,171 in the cell (2004,2004) indicates that out of 83 ETPs that were first released in the year 2004, 77 are considered clean. These 77 ETPs amount to 171 versions. Cell entry (2004,2005) indicates that 19 of 77 clean ETPs had new versions released in the year 2005 with a total of 54 versions in that year. We see that the total number of clean ETPs is 512 having a total of 1,873 versions, corresponding to the sum of the lower pair values in diagonal cells. Detailed information on number of incomplete ETPs and ETPs with no dependencies on ECPs can be found in [11].

We classify the clean ETPs based on evolution of their dependencies on "good" APIs and "bad" non-APIs:

- I ETPs with all versions dependent solely on APIs;
- II ETPs with all versions depending on a non-API;
- III ETPs with earlier versions dependent solely on APIs and latter versions depending on a non-API;
- IV ETPs with earlier versions dependent on a non-API and latter versions depending solely on APIs;
- V ETPs oscillating between versions that depend solely on APIs and versions that depend on a non-API.

Results of the classification are shown in Table II. Figure 1 shows the graphs for the two largest classifications, I and II.

First, adding the number of ETPs along the diagonals in Table II we observe that 286 ETPs (55.8%) belong to Classification I, i.e., do not depend on non-APIs, while 224 (44.2%) ETPs belong to Classifications II–V, i.e., there is at least one version of each ETP that depends on at least one non-API. There is, therefore, a significant number of ETPs depending on ECP non-APIs. Second, Figure 1 shows an increasing trend followed by stabilization for the percent of ETPs in Classification I and no clear trend for the percent of ETPs in Classification II. One of the possible reasons is that Eclipse is making it harder for developers of ETPs in Classification II. Third, we observe that many more ETPs continuously depend on non-APIs (Classification II) than those that removed dependencies on non-APIs at some point of time (Classification IV). This indicates that the elimination of non-APIs in the evolving ETP is very limited. Finally, comparing the numbers in Classifications III and IV we observe that there are more ETPs that start depending on non-APIs (Classification III) compared to those that eliminate the non-APIs (Classification IV). A possible reason for the use of non-APIs could be that the

		2003	2004	2005	2006	2007	2008	2009	2010
I	2003	*35,62*	10,20	3,4	1,1	1,4	2,2	0,0	0,0
	2004		*33,68*	4,11	4,9	2,3	2,2	0,0	0,0
	2005			*41,66*	10,21	4,5	3,4	1,1	1,1
	2006				*61,111*	7,13	1,1	0,0	2,3
	2007					*37,83*	12,22	4,6	6,12
	2008						*38,74*	7,12	2,4
	2009							*25,30*	3,4
	2010								*16,28*
	Total	35,62	43,88	48,81	76,142	51,108	58,105	37,49	30,52
II	2003	*33,91*	18,45	11,19	8,13	6,8	2,4	1,2	1,1
	2004		*35,77*	8,24	4,9	4,13	4,15	1,2	2,3
	2005			*29,60*	10,26	9,31	7,15	7,19	5,20
	2006				*25,61*	10,32	3,19	2,11	5,15
	2007					*16,31*	2,3	1,2	0,0
	2008						*22,42*	7,15	1,3
	2009							*11,11*	3,7
	2010								*10,11*
	Total	33,91	53,122	48,103	47,109	45,115	40,98	30,62	27,60
III	2003	*8,31*	5,12	5,6	3,3	2,2	1,1	0,0	0,0
	2004		*4,11*	2,3	2,7	2,6	2,5	1,1	0,0
	2005			*3,28*	2,12	1,4	1,1	0,0	0,0
	2006				*9,34*	4,14	2,11	0,0	2,5
	2007					*3,11*	3,8	2,4	0,0
	2008						*2,3*	0,0	1,1
	2009							*3,16*	1,1
	2010								*0,0*
	Total	8,31	9,23	10,37	16,56	12,37	11,29	6,21	4,7
IV	2003	*0,0*	0,0	0,0	0,0	0,0	0,0	0,0	0,0
	2004		*3,5*	3,10	2,4	0,0	0,0	0,0	1,5
	2005			*1,1*	1,1	0,0	1,1	1,3	1,2
	2006				*1,2*	1,3	0,0	0,0	1,1
	2007					*0,0*	0,0	0,0	0,0
	2008						*0,0*	0,0	0,0
	2009							*1,2*	1,1
	2010								*0,0*
	Total	0,0	3,5	4,11	4,7	1,3	1,1	2,5	4,9
V	2003	*3,11*	3,10	2,10	3,10	3,11	1,1	1,6	2,5
	2004		*2,10*	2,6	2,5	1,3	2,2	1,3	1,1
	2005			*0,0*	0,0	0,0	0,0	0,0	0,0
	2006				*2,15*	2,8	1,3	1,2	1,4
	2007					*1,9*	1,4	1,5	0,0
	2008						*0,0*	0,0	0,0
	2009							*0,0*	0,0
	2010								*0,0*
	Total	3,11	5,20	4,16	7,30	7,31	5,10	4,16	4,10

Table II: Classification of clean ETPs. Numbers of the ETPs that were first released in a certain year are typeset in italics.

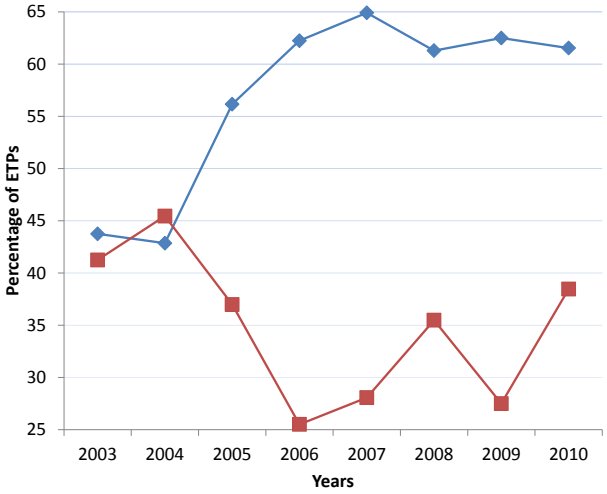


Figure 1: Percentages of ETPs in Classifications I (diamond) and II (square). The Y-axis scale is normalized by getting the percentage of ETP in each Classification from the total number of clean ETPs in the corresponding year.

functionality required is available solely via these non-APIs.

In the metrics analysis discussed in the next section, we focus solely on Classifications I and II. Classifications III–V are excluded as they contain too few ETPs. Classification I contains 286 plug-ins, Classification II—181 plug-ins.

IV. Metrics Analysis: Classification I ETPs vs. Classification II ETPs

In our previous study [6], carried out on a limited sample of carefully selected ETPs, we have observed that ETPs that depended on at least one non-APIs are larger systems compared to those that depended on APIs only. In this section, we replicate the previous study on a larger scale. Hence, we consider two metrics related to the size of a plug-in: *NOF*, the number of Java files, and *NOF-D*, the number of Java files that have at least one `import` statement related to ECP classes or interfaces. Furthermore, we want to understand whether Classifications I ETPs and II ETPs differ in the amount of ECP functionality used. We consider, therefore, two metrics related to the amount of ECP functionality imported by a plug-in: *D-Tot*, the number of `import` statements related to ECP classes and interfaces, and *D-Uniq*, the number of unique `import` statements related to ECP classes and interfaces.

We conducted our study using two data-sets of ETPs. For each year considered *Data-set I* includes one version of every Classification I or II ETP that has been first released in that year (cf. diagonal cells in Table II). Similarly, for each year considered *Data-set II* includes one version of every Classification I or II ETP that has been first released in that year or earlier (cf. total cells in Table II). Since we may have more than one version for an ETP released in one year, we select the last version in that year.

A. Metrics distribution

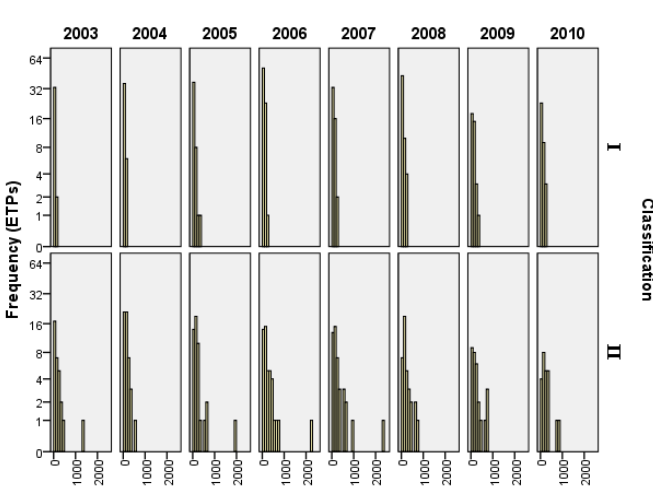
The distribution of all metrics studied is similar for both data sets: skewed to the right and with outliers. The outliers in the data are real since there are very large ETPs that have a numerous dependencies on ECP interfaces. Details of metrics' distributions can be found in [12]. Figure 2-(a) shows an example of metric distribution histograms: distribution of the D-Uniq values for Data-set II. Per metrics and per data-set we present 16 histograms corresponding to eight years (2003–2010) and two classifications (I and II). In the histograms in Figure 2-(a) we observe that (1) the median for Classification II is higher than that of Classification I in all the years; and (2) the concentration of the D-Uniq values in Classification I is on the lower side of x-axis for all the years, and the distribution is more spread in Classification II.

B. Hypothesis testing

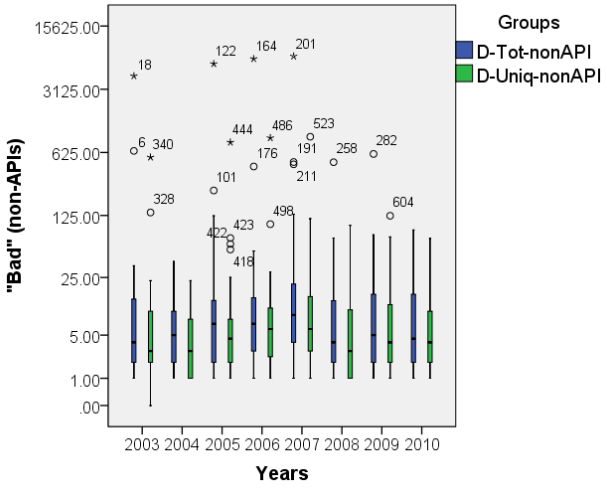
To verify validity of the observations made in Section IV-A, we perform statistical hypothesis testing. We formulate the null hypotheses, $H_0^{m,y,d}$ and the alternative hypotheses, $H_a^{m,y,d}$ for each metrics $m \in \{\text{NOF}, \text{NOF-D}, \text{D-Tot}, \text{D-Uniq}\}$, year y, $2003 \leq y \leq 2010$, and data-set $d \in \{\text{I}, \text{II}\}$. The null hypotheses state that on the data-set d the values of m for Classification I and II ETPs released in year y originate from the same distribution, The alternative hypotheses state that values of the metrics m for Classification II are higher than for Classification I. The choice for the "higher" alternative for NOF and NOF-D is based on the result of the previous study [6]. Similarly, choice for "higher" for D-Tot and D-Uniq is based on a pilot study we carried out.

Since the number of ETPs for Classification I and II are relatively low, especially in the recent years, we chose a less stringent nonparametric test, *Mann-Whitney U*, as opposed to the *two-independent-sample t* test that depends on the assumption of *normality* and relatively high number of data points [13]. In total we have to carry out sixty-four Mann-Whitney tests corresponding to four metrics, two data-sets and eight years (2003-2010).

Table III shows the p-values for the sixty-four Mann-Whitney tests. Assuming a common threshold of 0.05, for Data-set I we can reject the null-hypotheses for most year/-metric combinations, except for those indicated in italics in Table III. Thus, for most year/metrics combinations in Data-set I we accept the corresponding alternative hypotheses. For Data-set II the null-hypotheses can be rejected for all years and all metrics. Hence, for Data-set II we can confidently claim that the metrics values for ETPs in Classification II are higher than for Classification I, i.e., ETPs in Classification II have more Java files, more files dependent on ECPs, more



(a) Distributions of D-Uniq. The y-axis is given on a logarithmic scale.



(b) Distributions of D-Uniq and D-Tot for "bad" non-APIs

Figure 2: The ETPs in Classification II have higher dependency on ECP than those of Classification I, but their dependency on "bad" non-APIs remains low.

	#CI	#CII	NOF	NOF-D	D-Tot	D-Uniq
Data-set I						
2003	35	33	0.001	0	0	0
2004	33	35	0.011	0.004	0	0
2005	40	29	0.004	0	0	0
2006	61	25	*0.424*	0.029	0.017	0.013
2007	37	16	*0.068*	0.018	0.005	0.007
2008	38	22	0.010	0.003	0.001	0
2009	25	11	*0.614*	*0.556*	*0.527*	*0.527*
2010	16	10	0.027	0.007	0	0
Data-set II						
2003	35	33	0.001	0	0	0
2004	42	53	0.002	0	0	0
2005	47	48	0.001	0	0	0
2006	76	47	0.002	0	0	0
2007	51	45	0.002	0	0	0
2008	57	40	0	0	0	0
2009	37	30	0.027	0.008	0.009	0.002
2010	30	27	0.011	0	0	0

Table III: p-values for Classification I (CI) and II (CII) in Data-sets I and II. Zeros indicate values too small to be precisely determined by SPSS. Values exceeding 0.05 are typeset in italics.

dependencies on ECPs and more unique dependencies on ECPs.

Comparing the results obtained for Data-set I and Data-set II we observe that the results are similar: null hypotheses can be rejected for most years and metrics both for Data-set I and for Data-set II with 2009 and NOF being exceptions. In general, higher p-values on Data-set I can be attributed to low numbers of the data points in this data-set as opposed to Data-set II. Specifically, inability to reject $H_0^{m,2009,I}$ can be explained by the fact that eleven Classification II ETPs only were released in 2009 and included in Data-set I.

One possible reason for Classification II ETPs being larger than Classification I, may be that the functionality required by ETPs is absent from "good" APIs. We also conjecture that although developers might be aware that the "bad" non-APIs are volatile and unsupported, they may prefer to suffer the consequences of using these "bad" non-APIs than building their own APIs. Verifying or refuting this conjecture is considered as future work.

V. Metrics analysis: Non-API usage

In this section we focus on Classification II and consider numbers of "bad" non-APIs used by the ETPs. Figure 2-(b) presents ETPs of Data-set II and visualizes *D-Tot-nonAPI*, the number of `import` statements related to non-APIs, and *D-Uniq-nonAPI*, the number of unique `import` statements related to non-APIs. Comparing Figure 2-(a) and Figure 2-(b) we observe that although the ETPs in Classification II have a high dependency on ECP interfaces, medians in the box-plots reveal that the ETPs have a low dependency on "bad" non-APIs.

To formalize this observation, we conduct a statistical hypothesis testing whether *D-Tot-nonAPI* and *D-Tot*, as well as *D-Uniq-nonAPI* and *D-Uniq* represent different populations. We formulate, therefore, the following null and alternative hypotheses:

- $H_0^{\text{Tot},y,d}$: metric values of *D-Tot* and *D-Tot-nonAPI* obtained for ETPs in data-set d and year y represent the same population;

	1	2	3	4	5	6	7	8	9	10	11	12	13
# ETPs	28	15	14	12	11	9	8	7	6	5	4	3	2
# non-APIs	1	1	1	1	1	2	4	5	9	9	25	64	200

Table IV: Summary of the common non-APIs used by at least two of the 181 ETP-non-APIs. For example, in column 6, each of the 2 non-APIs is commonly used by 9 of the 181 ETP-non-APIs and in column 10, each of the 9 non-APIs is commonly used by 5 of the 181 ETP-non-APIs. The non-APIs considered in this table still exist in Eclipse SDK 3.7.

	Data-set I		Data-set II	
	Tot	Uniq	Tot	Uniq
2003	345	118.5	345	118.5
2004	281.92	111.07	308.5	121.5
2005	517.72	155	449.03	148.5
2006	306	108.5	594.14	176
2007	490.5	123.75	777.5	172.5
2008	320.5	128	597	178.37
2009	377.75	92.25	754.5	194.5
2010	221	114.5	683.53	192.3

Table V: Hodges-Lehmann estimation of the median of the difference between the number of (unique) interfaces and the number of (unique) non-APIs.

- $H_a^{\text{Tot},y,d}$: metric values of *D-Tot* obtained for ETPs in data-set d and year y are higher than those obtained for *D-Tot-nonAPI*;
- $H_0^{\text{Uniq},y,d}$: metric values of *D-Uniq* and *D-Uniq-nonAPI* obtained for ETPs in data-set d and year y represent the same population;
- $H_a^{\text{Uniq},y,d}$: metric values of *D-Uniq* obtained for ETPs in data-set d and year y are higher than those obtained for *D-Uniq-nonAPI*.

Our choice for "greater" as the directed alternative stems from the fact that non-API dependencies, counted by *D-Tot-nonAPI* and *D-Uniq-nonAPI* are a special kind of dependencies, while *D-Tot* and *D-Uniq* include both API and non-API dependencies. By the same argument the metric values are related, i.e., we have to conduct a paired two-sample test. We start by conducting a series of Shapiro-Wilk tests to check whether metric values are distributed normally. Depending on the outcome of these tests we should either use the well-known t test for two dependent samples (if the metric values are distributed normally) or its non-parametric counterpart, the paired two-sample Wilcoxon test.

The p-values obtained in the series of Shapiro-Wilk tests never exceeded 0.0003, i.e., normality hypothesis can be confidently rejected. Therefore, we conducted a series of paired two-sample Wilcoxon tests. Based on the p-values obtained we reject $H_0^{\text{Tot},y,d}$ and accept $H_a^{\text{Tot},y,d}$ for all years y and both data-sets d: the p-values never exceeded 0.001. Similarly, based on the p-values obtained we reject $H_0^{\text{Uniq},y,d}$ and accept $H_a^{\text{Uniq},y,d}$ for all years y and both data-sets d: the p-values never exceeded 0.001.

To provide better insights in the differences between *D-Tot* and *D-Tot-nonAPI* as well as between *D-Uniq* and *D-Uniq-nonAPI* we estimate the median of the difference between the corresponding metric values.[1] Table V summarizes the HodgesLehmann estimator values [14], [15]. The estimator values support the observation made by comparing Figure 2-(a) and Figure 2-(b): although the ETPs in Classification II have a high dependency on ECP interfaces, medians in the box-plots reveal that the ETPs have a low dependency on "bad" non-APIs.

Furthermore, in a preliminary investigation, we also found that most ETPs use "bad" non-APIs directly without using *wrappers*. Since on average not so many "bad" non-APIs are being used and the number of D-Tot is greater than D-Uniq (Figure 2-(b)), wrapping the functionality from "bad" non-APIs should be beneficial in case of any changes during the subsequent release of the ECPs.

VI. Commonly used non-APIs

In this section we continue our discussion of Classification II ETPs and the "bad" non-APIs used by these ETPs. We analyze the used "bad" APIs by ETPs in order to get a quantitative insight of what kind of "bad" APIs are highly used and how their distribution looks like. This would help inform Eclipse SDK developers the "bad" APIs that would be good candidates to become "good" APIs in the future.

We looked at the 181 ETPs in Classification II. Since we have a number of versions for each ETP, we decided to choose the last version for each ETP. A total of 1,717 unique non-APIs were extracted from the ETPs. 1,525 of the 1,717 non-APIs still exist in Eclipse SDK 3.7, the latest major release. We wrote a script that searches through the non-APIs for all ETPs and returns the ETPs that use a given non-API. Figure 4 shows the distribution of the frequency of use of the non-APIs. The distribution is positively skewed: there are many-non-APIs-used-by-few-ETPs and there are few-non-APIs-used-by-many-ETPs. Furthermore, the ETPs in the many-non-APIs-used-by-few-ETPs are just a small subset of the total 181 ETPs. This indicates that the ETPs use a diverse set of non-APIs. The most popular non-APIs are presented in Figure 3 while the complete list of non-APIs and their frequencies is available on-line.[2]

[1] We stress that the median of the difference is not the same as the difference in medians.

[2] http://www.win.tue.nl/∼jbusinge/CommonlyUsednonAPIs.xlsx

non-APIs	# ETPs
org.eclipse.jdt.internal.ui.JavaPlugin	28
org.eclipse.jdt.internal.core.JavaProject	15
org.eclipse.ui.internal.ide.IDEWorkbenchPlugin	14
org.eclipse.jdt.internal.corext.util.JavaModelUtil	12
org.eclipse.jdt.internal.ui.JavaPluginImages	11
org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor	9
org.eclipse.jdt.internal.core.PackageFragment	9
org.eclipse.jdt.internal.ui.wizards.TypedElementSelectionValidator	8
org.eclipse.core.internal.resources.Workspace	8
org.eclipse.jdt.internal.ui.util.ExceptionHandler	8
org.eclipse.jdt.internal.core.SourceType	8

Figure 3: A sample of commonly used non-APIs by the ETPs ranked according to the highest frequency to lowest frequency. The column *# ETPs* shows the number of ETPs of the 181 ETPs that use the corresponding non-API.

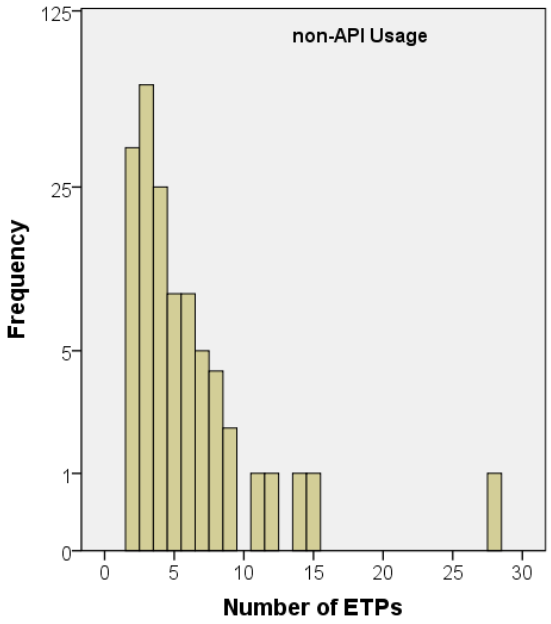
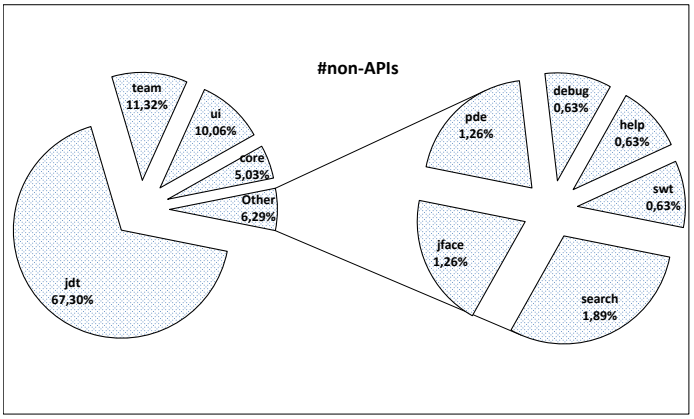


Figure 4: Distribution of frequency of non-API use. Non-APIs used by only one ETP have not been included.

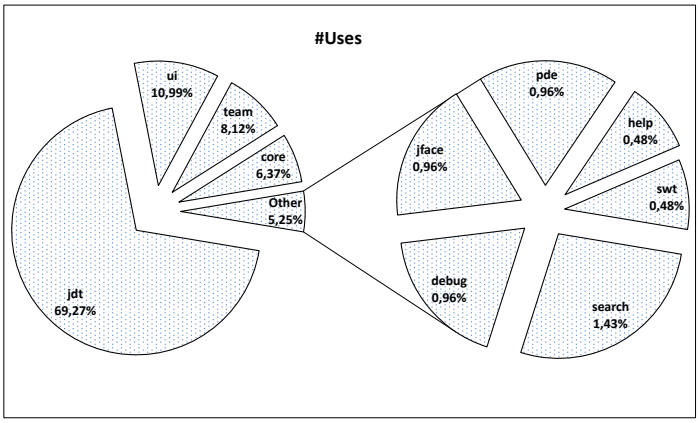
By observing Figure 3 one might conjecture that the most popular non-APIs are related to Eclipse Java development tools (JDT). This conjecture is confirmed by Figure 5. Figure 5a shows the distribution of the number of non-APIs used by at least two ETPs across different Eclipse projects. Similarly, Figure 5b shows the distribution of the use, i.e., the total number of import statements referring to non-APIs, across different Eclipse projects. Both figures suggest that the lion share indeed belongs to the JDT project. However, not less than ten different Eclipse projects deliver non-APIs used by at least two ETPs.

VII. Threats to Validity

As any other empirical study, our analysis may have been affected by validity threats. In our analysis, *construct validity* may be threatened by the metrics analysis in Sections IV and V. Indeed, presence of `org.eclipse.*` imports can result in incorrect counts in D-Tot and D-Uniq. However, for ETPs released in 2006 and earlier, less than 3% of the total imports have this form. The figure is even lower for ETPs released in 2007 or later. Additional threat pertains to unused imports. In our fact extraction unused imports are not excluded in the metrics. We tried to mitigate *internal validity* threats during the data collection by removing the incomplete ETPs and ETPs with no ECP-depencies. For *external validity*, it is possible that our results may not be generalizable since we only considered ETPs from Sourceforge. Sourceforge is, however, a big and well-known repository, and, therefore, our ETPs' collection can be considered representative.



(a) The number of non-APIs used by at least two ETPs across different Eclipse projects.



(b) The total number of import statements referring to non-APIs (used by at least two ETPs), across different Eclipse projects.

Figure 5: Distribution of non-API usage per project.

VIII. Related Work

Mileva et al. study popularity of APIs and state that, analyzing popularity of software projects is a relatively new research field [16]. Holmes and Walker [17] present a prototype tool calculating a popularity measure for every API in a framework. As opposed to our study that spans

a period of eight years, Holmes and Walker focus on one specific version of Eclipse. Mileva et al. [16] investigate API popularity on data collected from 200 open-source projects. The authors developed a tool prototype that analyzes the collected information and plots API element usage trends. As opposed to our focus on Eclipse, they present a general study of APIs usage in any given project.

The study that is more closely related to ours, is by Lämmel, Pek and Starek [18]. The authors demonstrate a scalable approach to AST-based API-usage analysis for a large-scale corpus of open-source projects. Their investigation reports on usage of 77 APIs from different domains extracted from built projects, reference projects and unbuilt projects. In comparison to our study, our API usage analysis considers APIs from the same domain, namely, Eclipse APIs. Since we only consider usage by only looking at the imports in a project, we did not need to build the studied projects. Finally, Grechanik et al. explore API usage in Java programs on the large scale [19].

This work builds on and extends the previous work on the evolution of ETPs [6]. In the previous study, we investigated the evolution of dependencies on ECPs of 21 carefully selected ETPs. We, however, did not distinguish between "good" APIs and "bad" non-APIs. The current study is one of the follow-up studies we pointed out in [6].

IX. Conclusions and Future Work

In this paper, we have investigated Eclipse SDK API usage by means of a case study of Eclipse third-party plug-ins. Our conclusion is based on empirical results for data collected on 512 Eclipse third-party plug-ins altogether having a total 1,873 versions. We discovered that about 44% of the 512 Eclipse third-party plug-ins depend on "bad" non-APIs and also discovered that developers continue using "bad" non-APIs in the new versions of the third-party plug-ins. The subsequent empirical study of 467 plug-ins also showed that plug-ins that use or extend at least one "bad" non-API are comparatively larger and also use more functionality from Eclipse than those that use or extend only "good" APIs. We also found out that ETP-non-APIs use a diverse set of "bad" APIs. This information provides Eclipse SDK developers with feedback on the current use of APIs and non-APIs in ETPs as opposed to the expected use. Furthermore, it reveals the characteristics of these ETPs that use the APIs and non-APIs. The information can be used as a starting point to answer questions like: why these situation occur and how they can be mitigated in future API releases.

Furthermore, we also observed that although the Eclipse third-party plug-ins have a heavy dependency on ECP interfaces, a percentage of "bad" non-APIs used by the plug-ins is relatively low. Based on the observation of low use of "bad" non-APIs, we suggested that to reduce the amount of effort spent on fixing the incompatibilities of the ETPs in the next ECP releases, ETP developers should use wrappers around non-APIs.

The results we have presented so far are encouraging. We have also identified possible ways in which the study should be extended. First, the investigation of a continued use of non-APIs was coarse grained. In our follow-up study we intend to expand the data set and in detail investigate the continued use of non-APIs by looking at the number of ETP classes, methods over time. Second, we intend to investigate possible reasons why developers depend on non-APIs. In Section IV-B we have already conjectured that absence of the required functionality from the APIs might be a possible reason for developers depend to choose for non-APIs. Third, collect more ETPs that uses non-APIs and carry out a related study on the ETPs grouped according according to number of non-API usage. It will also be interesting to investigate commonly used ECP non-APIs. Fourth, compare the failure rate of ETPs that depend on APIs and those that depend on at least one non-API when ported to new ECP releases. Fifth, we would like to extend this work and perform a comparable study of Eclipse Extension Plug-ins (EEPs). Sixth, using econometric techniques [20], [21] we plan to study whether the use of non-APIs is focused in a limited number of classes or is spread through the entire ETP. Finally, we intend to replicate our study on a different repository and also different framework to compare the findings with the current findings.

References

[1] S. Moser, O. Nierstrasz, The effect of object-oriented frameworks on developer productivity, Computer 29 (9) (1996) 45–51.

[2] D. Konstantopoulos, J. Marien, M. Pinkerton, E. Braude, Best principles in the design of shared software, in: COMPSAC'09, 2009, pp. 287 –292.

[3] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson, Object-oriented framework-based software development: problems and experiences, ACM Computing Surveys 32.

[4] T. Tourwe, T. Mens, Automated support for framework-based software, in: ICSM'03, 2003, pp. 148–157.

[5] Z. Xing, E. Stroulia, Refactoring practice: How it is and how it should be supported – an Eclipse case study, in: ICSM'06, 2006, pp. 458–468.

[6] 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, 2010, pp. 63–72.

[7] 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).

[8] A. Schröter, T. Zimmermann, A. Zeller, Predicting component failures at design time, in: ISESE'06, 2006, pp. 18–27.

[9] M. W. Godfrey, D. M. German, J. Davies, A. Hindle, Determining the provenance of software artifacts, in: IWSC'11, 2011, pp. 65–66.

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

[11] J. Businge, A. Serebrenik, M. van den Brand, Eclipse API usage: the good and the bad, Computer Science reports 11-15, Technische Universiteit Eindhoven (2011).

[12] J. Businge, Histograms and boxplots, http://www.win.tue.nl/~jbusinge/Histograms_and_Box-plots.pdf (2011).

[13] M. J. Norušis, SPSS 16.0 Guide to Data Analysis, Prentice Hall Inc., Upper Saddle River, NJ, 2008.

[14] J. L. Hodges, E. L. Lehmann, Estimates of location based on rank tests, The Annals of Mathematical Statistics 34 (2) (1963) 598–611.

[15] G. K. Rosenkranz, A note on the Hodges-Lehmann estimator., Pharmaceutical statistics 9 (2) (2010) 162–167.

[16] Y. M. Mileva, V. Dallmeier, A. Zeller, Mining API popularity, in: TAIC PART'10, 2010, pp. 173–180.

[17] R. Holmes, R. J. Walker, Informing Eclipse API production and consumption, in: OOPSLA'07, 2007, pp. 70–74.

[18] R. Lämmel, E. Pek, J. Starek, Large-scale, AST-based API-usage analysis of open-source Java projects, in: SAC'11, 2011, pp. 1317–1324.

[19] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, C. Ghezzi, An empirical investigation into a large-scale Java open source code repository, in: ESEM'10, 2010, pp. 11:1–11:10.

[20] A. Serebrenik, M. G. J. van den Brand, Theil index for aggregation of software metrics values, in: ICSM'10, 2010, pp. 1–9.

[21] B. Vasilescu, A. Serebrenik, M. G. J. van den Brand, You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics, in: ICSM'11, 2011, pp. 313–322.