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 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" APIs? We also study whether developers continue to use "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 analyzed Eclipse third-party plug-ins depends on "bad" APIs and that developers continue to use "bad" APIs. Furthermore, the empirical study shows that plug-ins that use or extend at least one "bad" API are comparatively larger and also use more functionality from Eclipse than those that use or extend only "good" APIs. We conjecture that since the larger plug-ins use extensive functionality, some of the functionality they require may be absent from the officially supported, "good" APIs.

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

I. INTRODUCTION

Today, many software developers are building their tools on top of frameworks. This approach has many advantages, such as: reuse of the functionality provided, ease of maintenance because they employ a standard development structure used by the framework, technical support from the component developers.

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. On the framework side, continuous evolution takes place as a result of refactoring and introduction of new functionality in the framework components. 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 [1]. In practice, for successfully and widely adopted frameworks, it may not be possible to achieve this fully. For developers who depend on the APIs of the framework have also potential challenges. 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 [1], [2], [3]: 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 stake holders of the systems themselves (general evolution).

To overcome the aforementioned challenges, one needs studies related to the framework and the software systems that reuse or extend the framework. There is a substantial amount of work that has been carried out for framework developers and framework user that is helpful in evolving the software systems.

- ***Tool Support:*** A number of tools have been developed to support developers, e.g. JDevAn [4] can be queried to provide the evolution history of the software system, UMLDiff [3] automatically detects changes made on a framework's interfaces, QUASOLEDO [5] automatically measures documentation quality in Eclipse projects. Goeminne and Mens proposed a framework for analyzing and visualizing open source software ecosystems [6]. Holmes et al. present a tool PopCon for mining Eclipse API [7]. The tool can be used by both by API developers and API users.
- ***Documentation:*** Documents guide developers in evolving their software systems e.g. Eclipse migration guides [8] aid developers on how to port their software system to a new release of Eclipse. Evolving Java-based APIs [9] guides framework developers as best as possible while ensuring compatibility with any hypothetical client extending or reusing the framework.
- ***Case studies:*** Schreck et al. [5] report on the evolution of documentation of the Eclipse platform. The study found that there exists a strong difference across modules for internal and non-internal-packages. Mileva et al. [10] report on API popularity. In the study, they present the use trend over time (increase or decrease) of popular APIs in the studied projects. Shihab et al. [11] study prediction of re-opened bugs through a case study of an Eclipse project.

Most studies carried out so far, mainly focus on facilitating the evolution of single systems. Lungu et al. states that software systems do not exist by themselves, but instead they exist in a lager context called "software ecosystems" [12]. Understanding the interconnection of the ecosystem, i.e., the framework, and the individual systems that compose it, i.e., the plug-ins that use and/or extend the framework, is critical for assessing the usage of the framework and the impact on the changes performed to it during evolution.

In our study, we carried out a detailed investigation on Eclipse SDK API usage, a case study on its third-party plug-ins. Eclipse SDK an example of framework that was introduced over a decade ago and has so far has 11 major releases. Like many evolving software systems, the Eclipse SDK has got both stable and supported APIs ("good") and unstable, discouraged and unsupported APIs ("bad"). However, despite being discouraged by Eclipse, in our experience, "bad" APIs are relatively often used in practice.

In this paper, we investigate to *what extent* developers use "bad" APIs? and also whether developers *continue to use* "bad" APIs? 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, a popular open source repositories. The third-party plug-ins collected date back from the beginning of the year 2003 to the end of 2010.

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 Section IV we analyze metrics for the identified groups of Eclipse third-party plug-ins. Section V we discuss the threats to validity. In Section VI we discuss the related work and finally, in Section VII we present the conclusion and future work.

II. Eclipse Plug-in Architecture

Eclipse 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 in the form of code libraries (Java classes with public Application programming interfaces (APIs)), platform extensions, or even documentation [13], [14].

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):*** These are plug-ins present in Eclipse SDK [15]. The plug-ins are shipped as part of the Eclipse SDK. The plug-ins provide core functionality upon which all plug-in extensions are built. The plugins also provide the runtime environment in which plug-ins are loaded, integrated, and executed.
- ***Eclipse Extension Plug-ins (EEP):*** These 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. Some common EEPs include: JST, EMF, PDT, CDT [15]. 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.
- ***Eclipse Third-Party Plug-ins (ETP):*** All the ETPs use at least some functionality provided by ECPs but also may use functionality provided by EEPs.

The Core plug-ins and the Extension plug-ins can be found on the Eclipse website [15]. Many of the third-party plug-ins can be found on Eclipse Marketplace [16] and SourceForge [17].

Eclipse clearly states general rules for extending or using the services it provides. The Eclipse guideline defines two types of interfaces [18], [9], [19]:

- ***ECP non-APIs ("bad"):*** In addition to Java access (visibility) levels, Eclipse has got another level called *internal implementations* (non-APIs). According to Eclipse naming convention [19], 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 [9]. 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.
- ***ECP APIs ("good"):*** These are the public Java classes or interfaces that can be found in packages that do not contain the segment "internal", public or protected methods, or fields in such class or interface. These APIs are considered to be stable and therefore can be used by any plug-in developer without any risk.

III. Data Collection

For our analysis, we collected ETPs from SourceForge, one of the most popular Open Source repositories. 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.

	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>

Table I: Number of ETPs collected and their corresponding years of release on SourceForge.

A. Downloading and initial 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 where 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.

Table I shows us the overall summary of the data collected from Sourceforge. The collection amounted to a total of 591 ETPs (sum of the diagonal), having a total of 1,323 versions. Cell entry (2004,2004)=<83,192> indicates that there are a total of 83 that were first released in the year 2004 on SourceForge altogether having a total of 192 versions. Cell entry (2004,2005)=<20,57> indicates that there were 20 ETPs of the 83 that had new versions released in the year 2005. The 20 ETPs have a total of 57 versions in the year 2005.

We observe that the trend on the evolution in the number of ETPs is non-monotone, for example looking at the number of ETPs, cell (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.

B. Removal of incomplete ETPs

To ensure that we have as few threats to validity as possible, we wrote *scripts* to remove ETPs *without dependencies* on ECPs and *incomplete* ETPs. During the data collection, we identified that some of the versions listed on SourceForge for some ETPs contained only *binaries* but no *sources* (but had at least one version with sources). We decided to label such ETPs as incomplete and excluded them.

Dependencies-on-ECPs are *import statements* from ECPs starting with the prefix `org.eclipse` in the source code of an ETP. 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.

Table III shows the *ETPs with no dependencies* on ECPs and Table II shows the *incomplete ETPs*. The ETPs that remained are considered to be *clean* ETPs (Table IV).

Tables IV (*clean* ETPs), have sub-tables (a) *Detailed* and (b) *Summarized*. The explanation for the entries in sub-tables (a) is as the same as the explanation of Table I in Section III-A, except for the column totals and row totals. In the *row totals*, for example in Table IV, the number of ETPs do not carry any message and therefore we omitted them. The number of versions, for example in cell (2004,Total)=<-,323> of Table IV-(a) indicate that 77 ETPs that were initially released in the year 2004 (cell (2004, 2004)) have altogether accumulated total of 323 versions for the time period of 7 years (2004-2010). In the *column totals*, the entries indicate the accumulated number ETPs and the corresponding accumulated number versions supported in a given ECP release (column header). For example, in Table IV, cell entry (Total,2005)=<113,247>, indicate that we have a total of 113 ETPs which is as a result of the sum of 73 ETPs that first existed during year 2005 and 19 and 21 ETPs that are a result of evolving ETPs from those that initially existed years 2004 and 2003 respectively.

Sub-table (b) shows the summarized version of the detailed sub-table (a). For example, Table IV-(b), row 2005=<73,322> indicates that there are 73 ETPs that first existed in the year 2005. Until 2010, these ETPs have altogether accumulated a total of 322 versions. The row Total=<512,1873> indicates that there are total of 512 ETP having a total of 1,873 versions for the group *clean ETPs*. The explanation for the entries in Table IV also holds for the entries in Table V-IX.

From Table IV-(b), we see that the total collection of clean ETPs is 512 (corresponds to sum of the diagonals in Table IV-(a)) having a total of 1,873 versions.

C. Final classification

Because an ETP could start by depending on "good" ECPs (ETP-API) in earlier versions and in later versions starts depending on "bad" ECPs (ETP-non-API) and the reverse is also possible, we identified a number of classifications for

	2003	2004	2005	2006	2007	2008	2009	2010	Total
2003	<7,38>	<3,7>	<1,2>	<1,1>	<1,1>	<0,0>	<0,0>	<1,2>	**<-,51>**
2004		<2,12>	<0,0>	<2,6>	<1,1>	<1,1>	<1,2>	<1,2>	**<-,24>**
2005			<8,19>	<4,13>	<3,5>	<5,7>	<1,9>	<0,0>	**<-,53>**
2006				<7,25>	<2,2>	<1,1>	<0,0>	<0,0>	**<-,28>**
2007					<15,40>	<3,5>	<2,2>	<4,6>	**<-,53>**
2008						<8,32>	<4,9>	<0,0>	**<-,41>**
2009							<4,9>	<1,2>	**<-,11>**
2010								<1,3>	**<-,3>**
Total	**<7,38>**	**<5,19>**	**<9,21>**	**<14,45>**	**<22,49>**	**<18,46>**	**<12,31>**	**<8,15>**	**<-,264>**

Table II: Number of ETPs and their versions collected, considered incomplete and their corresponding on SourceForge. ETPs having at least one version without sources amongst the versions listed on SourceForge.

	2003	2004	2005	2006	2007	2008	2009	2010	Total
2003	<4,7>	<1,5>	<1,3>	<1,2>	<1,2>	<0,0>	<0,0>	<0,0>	**<-,19>**
2004		<4,9>	<1,3>	<0,0>	<0,0>	<0,0>	<0,0>	<0,0>	**<-,12>**
2005			<7,19>	<3,6>	<2,2>	<1,3>	<0,0>	<0,0>	**<-,30>**
2006				<3,3>	<0,0>	<0,0>	<0,0>	<0,0>	**<-,3>**
2007					<1,3>	<1,4>	<0,0>	<0,0>	**<-,7>**
2008						<4,5>	<0,0>	<0,0>	**<-,5>**
2009							<3,4>	<0,0>	**<-,4>**
2010								<1,1>	**<-,1>**
Total	**<4,7>**	**<5,14>**	**<9,25>**	**<7,11>**	**<4,7>**	**<6,12>**	**<3,4>**	**<1,1>**	**<-,81>**

Table III: Number of ETPs and their versions collected, considered to have no dependencies on ECPs and their corresponding years of release on SourceForge.

	2003	2004	2005	2006	2007	2008	2009	2010	Total
2003	<80,195>	<37,87>	<21,39>	<16,27>	<12,25>	<6,8>	<2,8>	<3,6>	**<-,395>**
2004		<77,171>	<19,54>	<14,34>	<9,25>	<10,24>	<3,6>	<4,9>	**<-,323>**
2005			<73,154>	<23,60>	<14,40>	<12,21>	<9,23>	<8,24>	**<-,322>**
2006				<97,223>	<24,70>	<7,34>	<3,13>	<11,28>	**<-,368>**
2007					<57,134>	<18,37>	<8,17>	<6,12>	**<-,200>**
2008						<62,119>	<14,27>	<4,8>	**<-,154>**
2009							<40,59>	<8,13>	**<-,72>**
2010								<26,39>	**<-,39>**
Total	**<80,195>**	**<114,258>**	**<113,247>**	**<150,344>**	**<116,294>**	**<115,243>**	**<79,153>**	**<70,139>**	**<-,1873>**

(a) Detailed

ECP	# <ETP,Ver>
2003	<80,395>
2004	<77,323>
2005	<73,322>
2006	<97,368>
2007	<57,200>
2008	<62,154>
2009	<40,72>
2010	<26,39>
Total	**<512,1873>**

(b) Summarized

Table IV: Number of ETPs and their versions collected, considered as clean and their corresponding years of release on SourceForge.

	2003	2004	2005	2006	2007	2008	2009	2010	Total
2003	<35,62>	<10,20>	<3,4>	<1,1>	<1,4>	<2,2>	<0,0>	<0,0>	**<-,93>**
2004		<33,68>	<4,11>	<4,9>	<2,3>	<2,2>	<0,0>	<0,0>	**<-,93>**
2005			<41,66>	<10,21>	<4,5>	<3,4>	<1,1>	<1,1>	**<-,98>**
2006				<61,111>	<7,13>	<1,1>	<0,0>	<2,3>	**<-,128>**
2007					<37,83>	<12,22>	<4,6>	<6,12>	**<-,123>**
2008						<38,74>	<7,12>	<2,4>	**<-,90>**
2009							<25,30>	<3,4>	**<-,34>**
2010								<16,28>	**<-,28>**
Total	**<35,62>**	**<43,88>**	**<48,81>**	**<76,142>**	**<51,108>**	**<58,105>**	**<37,49>**	**<30,52>**	**<-,687>**

(a) Detailed

ECP	# <ETP,Ver>
2003	<35,93>
2004	<33,93>
2005	<41,98>
2006	<61,128>
2007	<37,123>
2008	<38,90>
2009	<25,34>
2010	<16,28>
Total	**<286,687>**

(b) Summarized

Table V: Number of ETPs are ETP-API throughout the versions released: Classification I

the different evolution scenarios and wrote scripts categorize them.

- Classification I (only good): ETPs that have all the versions collected as ETP-API.
- Classification II (only bad): ETPs that have all the versions collected as ETP-non-API.
- Classification III (good-bad): ETPs that have earlier versions conforming to Classification I and in their later versions start conforming to Classification II.
- Classification IV (bad-good): ETPs that have earlier versions conforming to Classification II and in their later versions start conforming to Classification I.
- Classification V (remainder): ETPs that oscillate between ETP-API *and* ETP-non-API.

	2003	2004	2005	2006	2007	2008	2009	2010	Total
2003	<33,91>	<18,45>	<11,19>	<8,13>	<6,8>	<2,4>	<1,2>	<1,1>	**<-,183>**
2004		<35,77>	<8,24>	<4,9>	<4,13>	<4,15>	<1,2>	<2,3>	**<-,143>**
2005			<29,60>	<10,26>	<9,31>	<7,15>	<7,19>	<5,20>	**<-,171>**
2006				<25,61>	<10,32>	<3,19>	<2,11>	<5,15>	**<-,138>**
2007					<16,31>	<2,3>	<1,2>	<0,0>	**<-,36>**
2008						<22,42>	<7,15>	<1,3>	**<-,60>**
2009							<11,11>	<3,7>	**<-,18>**
2010								<10,11>	**<-,11>**
Total	**<33,91>**	**<53,122>**	**<48,103>**	**<47,109>**	**<45,115>**	**<40,98>**	**<30,62>**	**<24,60>**	**<-,760>**

(a) Detailed

ECP	# <ETP,Ver>
2003	<33,183>
2004	<35,143>
2005	<29,171>
2006	<25,138>
2007	<16,36>
2008	<22,60>
2009	<11,18>
2010	<10,11>
Total	**<181,760>**

(b) Summarized

Table VI: Number of ETPs that are ETP-non-API throughout the versions released: Classification II

	2003	2004	2005	2006	2007	2008	2009	2010	Total
2003	<8,31>	<5,12>	<5,6>	<3,3>	<2,2>	<1,1>	<0,0>	<0,0>	**<-,55>**
2004		<4,11>	<2,3>	<2,7>	<2,6>	<2,5>	<1,1>	<0,0>	**<-,33>**
2005			<3,28>	<2,12>	<1,4>	<1,1>	<0,0>	<0,0>	**<-,45>**
2006				<9,34>	<4,14>	<2,11>	<0,0>	<2,5>	**<-,64>**
2007					<3,11>	<3,8>	<2,4>	<0,0>	**<-,23>**
2008						<2,3>	<0,0>	<1,1>	**<-,4>**
2009							<3,16>	<1,1>	**<-,17>**
2010								<0,0>	**<-,0>**
Total	**<8,31>**	**<9,23>**	**<10,37>**	**<16,56>**	**<12,37>**	**<11,29>**	**<6,21>**	**<4,7>**	**<-,241>**

(a) Detailed

ECP	# <ETP,Ver>
2003	<8,55>
2004	<4,33>
2005	<3,45>
2006	<8,64>
2007	<3,23>
2008	<2,4>
2009	<3,17>
2010	<0,0>
Total	**<31,241>**

(b) Summarized

Table VII: Number of ETPs that initially were ETP-API and later ETP-non-API: Classification III

	2003	2004	2005	2006	2007	2008	2009	2010	Total
2003	<0,0>	<0,0>	<0,0>	<1,0>	<0,0>	<0,0>	<0,0>	<0,0>	**<-,0>**
2004		<3,5>	<3,10>	<2,4>	<0,0>	<0,0>	<0,0>	<1,5>	**<-,24>**
2005			<1,1>	<1,1>	<0,0>	<1,1>	<1,3>	<1,2>	**<-,8>**
2006				<1,2>	<1,3>	<0,0>	<0,0>	<1,1>	**<-,6>**
2007					<0,0>	<0,0>	<0,0>	<0,0>	**<-,0>**
2008						<0,0>	<0,0>	<0,0>	**<-,0>**
2009							<1,2>	<1,1>	**<-,3>**
2010								<0,0>	**<-,0>**
Total	**<0,0>**	**<3,5>**	**<4,11>**	**<5,7>**	**<1,3>**	**<1,1>**	**<2,5>**	**<4,9>**	**<-,41>**

(a) Detailed

ECP	# <ETP,Ver>
2003	<0,0>
2004	<3,24>
2005	<1,8>
2006	<1,6>
2007	<0,0>
2008	<0,0>
2009	<1,3>
2010	<0,0>
Total	**<6,41>**

(b) Summarized

Table VIII: Number of ETPs that initially were ETP-non-API and later ETP-API: Classification IV

	2003	2004	2005	2006	2007	2008	2009	2010	Total
2003	<3,11>	<3,10>	<2,10>	<3,10>	<3,11>	<1,1>	<1,6>	<2,5>	**<-,64>**
2004		<2,10>	<2,6>	<2,5>	<1,3>	<2,2>	<1,3>	<1,1>	**<-,30>**
2005			<0,0>	<0,0>	<0,0>	<0,0>	<0,0>	<0,0>	**<-,0>**
2006				<2,15>	<2,8>	<1,3>	<1,2>	<1,4>	**<-,32>**
2007					<1,9>	<1,4>	<1,5>	<0,0>	**<-,18>**
2008						<0,0>	<0,0>	<0,0>	**<-,0>**
2009							<0,0>	<0,0>	**<-,0>**
2010								<0,0>	**<-,0>**
Total	**<3,11>**	**<5,20>**	**<4,16>**	**<7,30>**	**<7,31>**	**<5,10>**	**<4,16>**	**<4,10>**	**<-,144>**

(a) Detailed

ECP	# <ETP,Ver>
2003	<3,64>
2004	<2,30>
2005	<0,0>
2006	<2,32>
2007	<1,18>
2008	<0,0>
2009	<0,0>
2010	<0,0>
Total	**<8,144>**

(b) Summarized

Table IX: Number of ETPs that have been oscillating between ETP-API and ETP-non-API: Classification V

Tables V-IX present the ETPs in the different groups. The explanation for the entries in Tables V-IX is similar as the entries in Table IV in Section III-B above. Figure 1, graphs for all the considered classifications. The Y-axis scale is normalized by getting the percentage of ETP in each classification from the total in Table IV.

D. Discussion of the Results

From the results presented in Tables IV-IX and in Figure 1, we discuss a number of observations:

- Looking at Table V-(b) there are 286 ETP-APIs (55.8%) in Classification I and in Table VI-(b) there are 224 (44.2%) ETP-non-APIs Classification II-V (alt least one version of each ETP in the groups depends

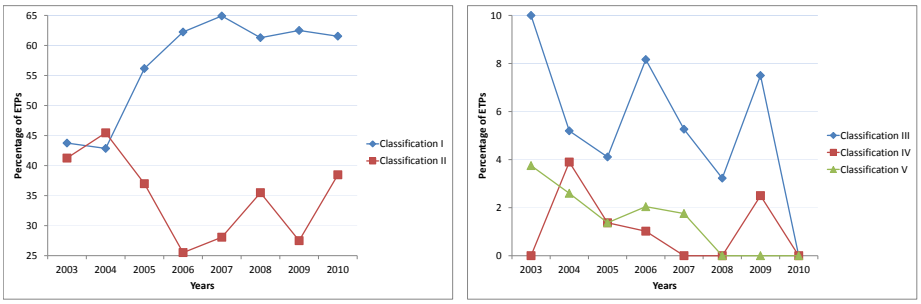


Figure 1: Graphs of all the different Classifications. The Y-axis scale is normalized by getting the percentage of ETP in each Classification from the total in Table IV

on at least one non-API). From this observation, we conclude that there is a significant number of ETPs depending on ECP non-APIs.

- Looking at the trends of the percentage of new ETPs, for Classification I and II, in Figure 1, we observe over time that there an increase followed by stabilization trend for Classification I and no clear trend for Classification II. One of the possible reasons is that Eclipse is making it harder for developers of ETPs in Classification II.
- Looking at the number of evolving ETPs and their corresponding number of versions in Classification II, Table VI (the rows), we can observe that a number of ETPs have a continuous dependency on non-APIs. The observation that ETPs continuous dependency on non-APIs can also be explained by comparing the numbers of ETPs in Classification II in Table VI and Classification IV in Table VIII, we observe that there are very high numbers in Classification II and very low numbers in Classification IV. This indicates that the elimination of non-APIs in the evolving ETP is very minimal. When we also compare the numbers in Classification III and Classification IV in Tables VII and Table VIII respectively, we observe that there are more ETPs that start depending on non-APIs (Classification III) compared to those that try to eliminate the non-APIs (Classification IV). The reason to the continued use of non-APIs could be that developers have no choice but to continue using the non-APIs.

The next section, using metrics, we investigate why ETPs in Classification I constantly depend on "good" APIs and why ETPs in Classification II constantly depend on at least one "bad" API. Classification III to V ETPs are not considered since there are very few number of ETPs in the classifications to make sound statistical investigations.

IV. Metrics Analysis

The motivation for using metrics to investigate why ETPs in Classification I constantly depend on "good" APIs and why ETPs in Classification II constantly depend on at least one "bad" API, is got from our previous paper [2]. In the previous study, we performed a general study on the evolution of ETPs. During that study, we investigated Lehman's laws of software evolution on a sample size of carefully selected ETPs using a number of metrics. During the metric analysis in this previous study, the data suggested that ETPs that depended on at least one non-APIs were larger systems compared to those that depended on only APIs.

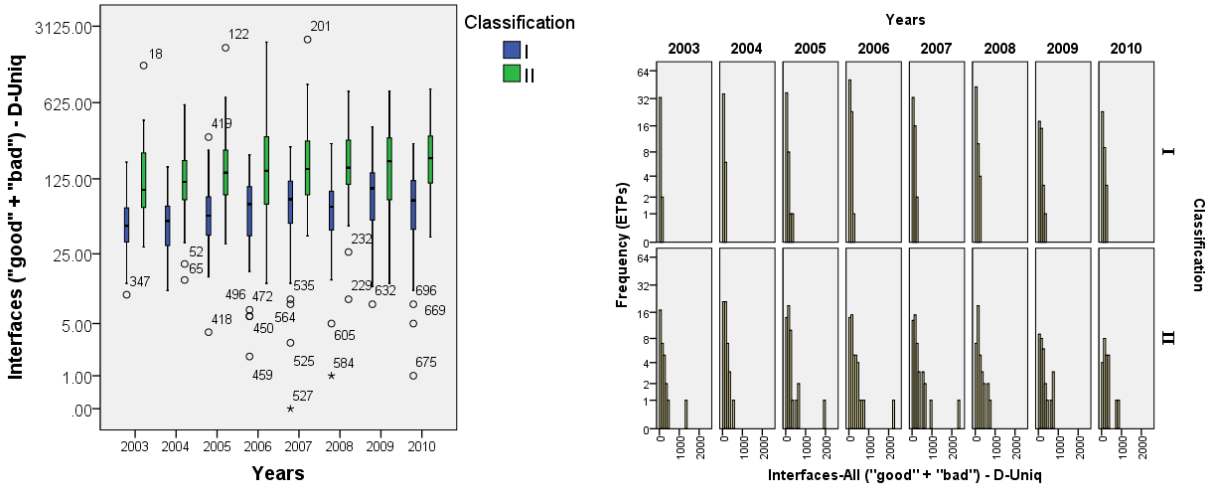
In this study, we decided to go further and prove our previous conjecture. We also thought that if it is apparent that ETPs that depend on both ECP-API and ECP-non-APIs are comparatively larger than those that depend on only ECP-APIs, it would give us possible clues on why ETPs constantly depending on APIs and why ETPs constantly depending on at least one non-API.

The metrics related to measuring system-size of ETPs and amount-of-functionality from ECPs interfaces consumed by the ETPs from the previous study were selected for the current study. They include:

- **NOF**: This metric measures the size of an ETP. It is obtained by counting the *number of .java files* in a selected version of the ETP.
- **NOF-D**: This metric measures the the size of an ETP. It is obtained by counting only the *number of .java files* that have *at least one* `import` statement related to ECP classes or interfaces.
- **D-Tot**: This metric measures the of amount of functionality from ECPs used by an ETP. The total count of `import` statements related to ECP classes and interfaces in an ETPs source code represent D-Tot.
- **D-Uniq**: This metric also measures the amount of functionality from ECP used by an ETP. The unique count of `import` statements related to ECP classes and interfaces in an ETPs source code represent D-Uniq.

A. Hypothesis

We formulated two sets of hypotheses that relate size of the ETPs and amount of functionality from ECP used by the ETPs.



(a) Box-Plot–Data-set II

(b) Histogram–Data-set II

Figure 2: Plots showing the distributions of the number of ECP Interfaces depended on by ETPs. Each data point in a box-plot is an ETP and the height of the data point is the count of D-Uniq used by the ETP. The histogram shows the count of ETPs (y-axis) falling in a given range of ECP Interface count (x-axis). On both plots, the y-axis is given on a logarithmic scale.

	2003		**2004**		**2005**		**2006**		**2007**		**2008**		**2009**		**2010**	
	CI	CII	CI	CII	CI	CII	CI	CII	CI	CII	CI	CII	CI	CII	CI	CII
# ETPs	35	33	33	35	40	29	61	25	37	16	38	22	25	11	16	10
NOF	**.001**		**.011**		**.004**		.424		.068		**.010**		.614		**.027**	
NOF-D	**.000**		**.004**		**.000**		**.029**		**.018**		**.003**		.556		**.007**	
D-Tot	**.000**		**.000**		**.000**		**.017**		**.005**		**.001**		.527		**.000**	
D-Uniq	**.000**		**.000**		**.000**		**.013**		**.007**		**.000**		.527		**.000**	

Table X: $p\text{-}values$ for Classification I (CI) and II (CII) for the considered metrics. Diagonal entries in Tables V and Table VI (Data-set I)

	2003		**2004**		**2005**		**2006**		**2007**		**2008**		**2009**		**2010**	
	CI	CII	CI	CII	CI	CII	CI	CII	CI	CII	CI	CII	CI	CII	CI	CII
# ETPs	35	33	42	53	47	48	76	47	51	45	57	40	37	30	30	24
NOF	**.001**		**.002**		**.001**		**.002**		**.002**		**.000**		**.027**		**.011**	
NOF-D	**.000**		**.000**		**.000**		**.000**		**.000**		**.000**		**.008**		**.001**	
D-Tot	**.000**		**.000**		**.000**		**.000**		**.000**		**.000**		**.009**		**.000**	
D-Uniq	**.000**		**.000**		**.000**		**.000**		**.000**		**.000**		**.002**		**.000**	

Table XI: $p\text{-}values$ for Classification I (CI) and II (CII) for the considered metrics. The number of ETPs include contribution from evolved ETPs previously supported in earlier ECPs (column totals in Tables V and Table VI (Data-set II)).

- ***Null hypotheses (NOF/NOF-D)***. $H1_0$: *The population mean values of the ETP-size for the two groups, Classification I and II, are the same for metrics* ***NOF*** *and* ***NOF-D***.
- ***Alternative hypotheses***. $H1_a$: *Classification II ETP-size population mean values are higher than Classification I ETP-size population mean values for metrics* ***NOF*** *and* ***NOF-D***.
- ***Null hypotheses (D-Tot/D-Uniq)***. $H2_0$: *The population mean values of the amount of functionality used by ETPs for the two groups, Classification I and II, are the same for metric* ***D-Tot*** *and* ***D-Uniq***.
- ***Alternative hypotheses***. $H2_a$: *The population mean values of the amount of functionality used by ETPs for the two groups, Classification I and II, are different for metric* ***D-Tot*** *and* ***D-Uniq***.

Hypotheses $H1$ is based on result of the observation from the previous study [2] where the the data suggested that ETPs that depended on at least one non-APIs were larger systems compared to those that depended on only APIs.

The sample sizes (data points) for Classification I and II are relatively low, especially the number of ETPs in the recent years (number of ETPs in the diagonals). Therefore, for testing whether the population mean in Classification I

has smaller values than those of Classification II, we chose to use the 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 [20].

To test the hypotheses, we used two data-sets: 1) *Data-set I*: the number of ETPs in the diagonal entries for the two groups, Classification I and II (Table V and Table VI). 2) *Data-set II*: the number of ETPs in the column totals (sum of diagonal ETPs and the contribution of the evolving ETPs that were first released in earlier years, Table V and Table VI). When performing the experiments in the two sets of data presented, for each of the two groups to be compared, Classification I and II, we selected one version per ETP. Since we may have more than one version for an ETP released in one year, we decided to select the last version in that year.

B. Results

The box plot in Figure 2-(a) and histogram in Figure 2-(b) show the distribution of the number of ECP Interfaces in Data-set II, a comparison of ETPs in Classification I and II. From the box-plot we can observe that the median for Classification II is higher than that of Classification I in all the years. From the histogram, we can observe from Classification I that the concentration of the ECP Interfaces is on the lower side of x-axis for all the years and for Classification II, the distribution is a little bit spread. The other studied metrics show a similar trend and can be found at [21]. In all the metrics the data points are *heavily skewed* to the *right* and also have a number of *outliers*. The outliers in the data are real since we can have very large ETPs that have a heavy dependency on ECP interfaces.

Table X and Table XI show the $p\text{-}values$ from the analysis for Data-set I and Data-set II respectively for the stated hypotheses above. If we consider a threshold of ($p\text{-}value \leq 0.05$) for both data-sets, we can see that only a few $p\text{-}values$ entries (not in bold) data-set I in Table X are above the threshold. The reason the high values in the $p\text{-}values$ could be attributed to the very low numbers of the data points in CI compared CII in Table XI. For data-set II $p\text{-}values$, we can see that all the entries are below the threshold value since we have all the numbers in the data-set for both CI and CII ≥ 30 except one.

From the results presented, we can confidently reject the null hypotheses, $H1_0$ and $H2_0$, stating that there is no difference in ETP-size and amount of functionality for ECP used by ETPs between ETPs in Classification I and II. We can instead confidently accept the alternative hypotheses, $H1_a$ and $H2_a$, stating that ETP-sizes and amount of functionality for ECP used by ETPs in Classification II have larger values than ETP-sizes and amount of functionality for ECP used by ETPs in Classification I since it is statistically significant from the $p\text{-}values$ presented.

Apparently, large systems that also consume extensive functionality from ECP interfaces, in Classification II compared those to Classification I. One possible reason could be that since ETPs in Classification II are relatively large and at the same time consume extensive functionality from ECPs interfaces, the functionality they require may be absent in "good" APIs. We also conjecture that much as developers are aware that the "bad" APIs are volatile and unsupported, they may prefer to suffer the consequences of using these "bad" APIs than building their own APIs.

Furthermore in Classification II, we wanted to observe the percentage of "bad" APIs used by the ETPs. Figure 3-(a) and Figure 3-(b) show the distribution of the "bad" APIs in Data-set I and Data-set II, respectively. Although the ETPs in Classification II have a very high dependency on ECP Interfaces as can be observed in Figure 2, the ETPs have a low dependency on "bad" APIs as can be observed in Figure 3 by looking at the location of medians in the box-plots.

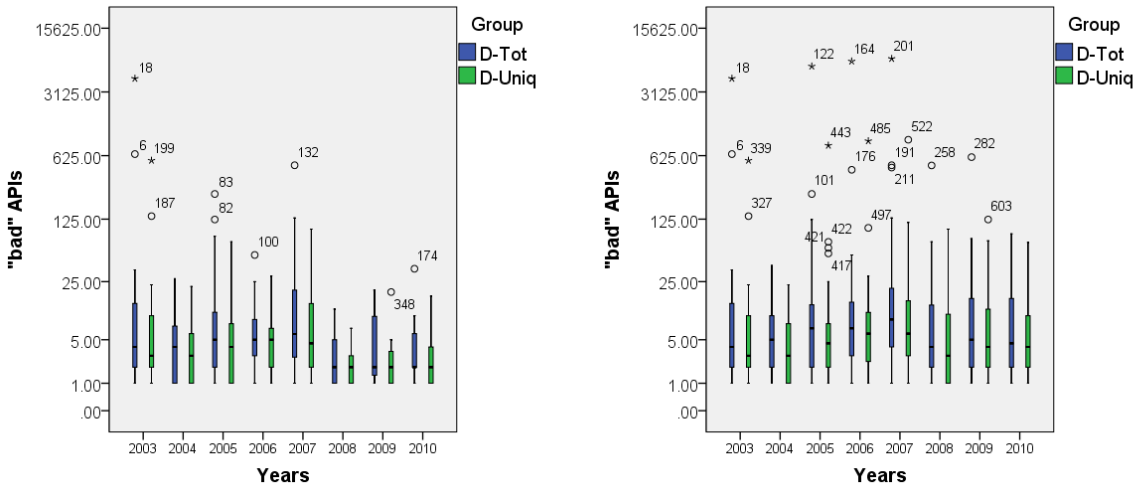
In our preliminary investigation, we also found that most ETPs use "bad" APIs directly without using *wrappers*. Since from Figure 3, on average not so many "bad" APIs are being used and the number of D-Tot is greater than D-Uniq. One would save some of time by wrapping the functionality from "bad" APIs in case of any changes during the subsequent release of the ECPs.

V. Threats to Validity

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

In our analysis, construct validity may threaten by the results in metric analysis in Section IV as we can have imports in the form of `org.eclipse.*` resulting to incorrect counts in D-Tot and D-Uniq. However, since we consider a number of metrics, for example NOF-D, D-Tot and D-Uniq, the threats' impact is lessened since we have statistically significant results for all these metrics. Furthermore, we run tests to check the amount of `org.eclipse.*` and `org.eclipse.internal*` (selected from the last version of an ETP–See Section IV-A for the detailed explanation) imports and we found out that for ETPs released in 2006 and earlier, we have less than 3% of the total imports and even less from 2007 and later. The other threat that we did not have control of is unused imports. In our fact extraction since we do not build plug-ins, unused imports are not excluded in the metrics. In the data extraction, we relied on the on the Eclipse naming convention [9] rather than the actual API guidelines [19]. If software systems deviate from the convention but stick to the guideline, then that would introduce noise in our study.

We tried to mitigate internal validity threats during the data collection by removing the incomplete ETPs Section III-B and grouping the ETPs according to the classification in Section III-C.



(a) Data-set I

(b) Data-set II

Figure 3: Box-plots showing the distribution of the number of ”bad” APIs depended on by ETPs in Classification II. Each data point in a box-plot is an ETP and the height is the count of ”bad” APIs (D-tot and D-Uniq) in the ETP

For external validity, it is possible that our results may not be generalizable since we only considered ETPs from Sourceforge. For generalization, one may need to consider ETPs from other repositories so as to find out whether the results obtained are similar.

VI. Related Work

This work complements the previous work on the evolution of ETPs [2]. In previous study, we investigated the constrained evolution of 21 carefully selected ETPs on Lehman's software evolution laws. Specifically, we investigated the evolution of ETPs' dependencies-on-ECPs in the new releases of the ETPs without differentiating the "good" ETPs and the "bad" ETPs. The current study is one of the follow-up studies we pointed out in our previous work.

Mileva et al. state that, the research field of analyzing API usage in software projects is relatively new [10]. In our search, we also did not find a lot of studies that investigate API usage. In this section, we present and discuss a few that relate to our study. The information reported in our study and the ones we are going to discuss can be of great value for both API developers and API users.

Holmes and Walker present PopCon a prototype tool that can help API developers understand how their APIs are being used as well as help API users locate important API and get examples of how it is used in practice [7]. The tool calculates a popularity measure for every API in a framework by analyzing its static structure. The authors used data from a fixed point in time and investigated the usage of Eclipse API and therefore their study is very focused. In comparison to our study, the data that we used spans a period of eight years and goes through the code history for 512 projects. And our study is more focused in the sense that it separates the "good" and the "bad" Eclipse APIs.

The work of Mileva et al. [10] is very much related to that Holmes and Walker [7]. The authors investigate API popularity on data collected from 200 open-source projects that spans a period of one year. In their study, the authors developed a tool prototype that analyzes the collected information and plots API element usage trends. As opposed to our focus on Eclipse, their study is a general study on all kinds of APIs usage in any given project.

The study that is more closely related to ours, is by Lämmel, Pek and Starek [22]. 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 a corpus of projects studied. The authors extract the API usage from the based on three approaches: built projects, reference projects and unbuilt projects. In comparison to our study, our API usage analysis considers APIs from the same domain, 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.

VII. Conclusion and Future Work

In this paper, we have investigated Eclipse SDK API usage a case study of Eclipse third-party plug-ins with more emphasis on the use of ["good" and "bad"] APIs. Our conclusion is based on empirical results on data collected

on 512 Eclipse third-party plug-ins altogether having a total 1,873 versions. We discovered that about 44% of Eclipse third-party plug-ins depend on "bad" APIs and also discovered that developers continue using "bad" APIs in the new versions of the third-party plug-ins. The empirical study also showed that plug-ins that use or extend at least one "bad" API are comparatively larger and also use more functionality from Eclipse than those that use or extend only "good" APIs. Furthermore, we also observed that although the Eclipse third-party plug-ins have a heavy dependency on ECP interfaces, the percentage of "bad" APIs used by the plug-ins is relatively very low. Based on the observation of low use of "bad" 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 the continued use of "bad" APIs was coarse grained. In our follow up study we intend to expand the data set and in detail investigate the the continued use of "bad" APIs by looking at the number of ETP classes, methods over time. Second, in addition to the conjecture in Section IV-B that possible reason why Classification II ETPs depend on "bad" APIs could be that the functionality they require may be absent in the "good" APIs, we intend to investigate possible reasons why developers depend on "bad" APIs. Third, collect more ETPs that uses "bad" APIs and carry out a related study on the ETPs grouped according according to number of "bad" API usage (low, medium, high). It will also be interesting to investigate commonly used ECP "bad" APIs. Fourth, compare the failure rate of ETPs that depend on "good" APIs and those that depend on at least one "bad" API when ported to new ECP releases. 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] Z. Xing, E. Stroulia, Refactoring practice: How it is and how it should be supported – an Eclipse case study, in: ICSM'06, IEEE Computer Society, 2006, pp. 458–468.

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

[3] Z. Xing, E. Stroulia, API-evolution support with Diff-catchup, IEEE Tran. Soft. Eng 33 (12) (2007) 818–836.

[4] Z. Xing, E. Stroulia, The JDEvAn tool suite in support of object-oriented evolutionary development, in: ICSE Companion'08, 2004, pp. 123–128.

[5] D. Schreck, V. Dallmeier, T. Zimmermann, How documentation evolves over time, in: IWPSE'07, ACM, 2007.

[6] M. Goeminne, T. Mens, A framework for analysing and visualising open source software ecosystems, in: IWPSE-EVOL'10, 2010, pp. 42–47.

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

[8] Eclipse plug-in migration guide. http://dsdp.eclipse.org/help/latest/index.jsp (Consulted on January 20, 2011).

[9] J. des Rivires, Evolving Java-based APIs, 2007, Tech. rep., http://wiki.eclipse.org/Evolving_Java-based_APIs (Consulted on January 01, 2011).

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

[11] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. Hassan, K. Matsumoto, Predicting re-opened bugs: A case study on the Eclipse project, in: WCRE'10, 2010, pp. 249 –258.

[12] M. Lungu, R. Robbes, M. Lanza, Recovering inter-project dependencies in software ecosystems., in: ASE'10, 2010, pp. 309–312.

[13] A. Bolour, Notes on the Eclipse plug-in architecture, 2003, Tech. rep., http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html (Consulted on January 01, 2011).

[14] S. Delap, Understanding how Eclipse plug-ins work with OSGi, 2006, Tech. rep., http://www.eclipse.org/articles/article.php?file=Article-API-Use/index.html (Consulted on July 19, 2011).

[15] Eclipse projects. http://www.eclipse.org/projects/listofprojects.php (Consulted on June 01, 2010).

[16] Eclipse marketplace. http://marketplace.eclipse.org/ (Consulted on June 01, 2010).

[17] Sourceforge. http://sourceforge.net/ (Consulted on July 20, 2010).

[18] Provisional API guidelines. http://wiki.eclipse.org/provisional_api_guidelines (Consulted on January 20, 2011).

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

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

[21] J. Businge, http://www.win.tue.nl/∼jbusinge/histograms_and_box-plots.pdf.

[22] 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.