Software Qual J (2015) 23:107–141
DOI 10.1007/s11219-013-9221-3

Eclipse API usage: the good and the bad

John Businge · Alexander Serebrenik · Mark G. J. van den Brand

Published online: 1 October 2013

Abstract Today, when constructing software systems, many developers build their systems on top of frameworks. Eclipse is such a framework that has been in existence for over a decade. Like many other evolving software systems, the Eclipse platform has both stable and supported interfaces ("good") and unstable, discouraged and unsupported interfaces ("bad"). In this study, we investigate Eclipse interface usage by Eclipse third-party plug-ins (ETPs) based on whether they use bad interfaces or not. The investigations, based on empirical analysis present the following observations. First, we discovered that 44 % of the 512 analyzed Eclipse third-party plug-ins depend on "bad" interfaces and that developers continue to use "bad" interfaces. Second, we have observed that plug-ins that use or extend at least one "bad" interface are comparatively larger and use more functionality from Eclipse than those that use only "good" interfaces. Third, the findings show that the ETPs use a diverse set of "bad" interfaces. Fourth, we observed that the reason why the bad interfaces are being eliminated from the ETPs' source code is, because (ETP developers believe) these non-APIs will cause incompatibilities when a version of the ETP is ported to new Eclipse SDK release. Finally, we observed that when developers eliminate problematic "bad" interfaces, they either re-implement the same functionality in their own API, find equivalent SDK good interfaces, or completely delete the entities in the ETPs' source code that use the functionality from the "bad" interfaces.

Keywords Eclipse · API usage · Software evolution

J. Businge · A. Serebrenik (✉) · M. G. J. van den Brand
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: a.serebrenik@tue.nl

J. Businge
e-mail: j.businge@tue.nl

M. G. J. van den Brand
e-mail: M.G.J.v.d.Brand@tue.nl

1 Introduction

Today, many software developers are building their systems on top of frameworks (Brugali et al. 2007; Tourwe and Mens 2003). This approach has many advantages, such as reuse of the functionality provided (Moser and Nierstrasz 1996) and increased productivity (Konstantopoulos et al. 2009). 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 (Bosch et al. 2000).

On the framework developer side, continuous evolution takes place as a result of refactoring and introduction of new functionality in the framework components (Tourwe and Mens 2003). 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 (Xing and Stroulia 2006). In practice, for successfully and widely adopted frameworks, such as Eclipse[1], 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 framework-based software systems may be constrained by two independent, and potentially conflicting, processes (Businge et al. 2010; Xing and Stroulia 2006): (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, the Eclipse SDK[2] distinguishes between stable and supported "good" interfaces[3] and unstable, discouraged and unsupported "bad" interfaces (des Rivières and J.: How to use the Eclipse API 2001). The latter are indicated by the substring `internal` in the package name. While "good" interfaces can be safely used in Eclipse-based systems, the use of non-APIs ("bad") is at risk of arbitrary change or removal without notice. It has been shown that many non-APIs are among packages that are most likely to introduce a post-release failure (Schröter et al. 2006).

In this paper we present results of the investigation on the usage of the Eclipse API and non-API[4] by Eclipse third-party plug-ins (ETPs). Our investigation was based on an empirical study of a total of 512 Eclipse third-party plug-ins altogether having a total of 1,873 versions collected from SourceForge and released between 2003 and 2010. The ETPs were categorized into five different groups based on how they use bad interfaces.

Specifically, our investigation answers the following questions: (1) *what extent do* third-party plug-ins use bad interfaces, (2) whether developers *continue to use* the bad interfaces, (3) what are the *differences* between the third-party plug-ins that use bad interfaces and those that do not, (4) what bad interfaces are *commonly used* by the plug-ins and (5) why do developers sometimes *eliminate* bad interfaces from the ETPs?

The remainder of the paper is organized as follows. In Sect. 2 we present the notion of Eclipse plug-ins and their interfaces. In Sect. 3 we present the methodology used in data collection. In Sect. 4 we quantitatively analyze two of the largest identified groups of Eclipse third-party plug-ins using size and dependency metrics. In Sect. 5 we continue the

[1] http://www.eclipse.org

[2] In the rest of the paper, the word SDK refers to Eclipse SDK.

[3] In this paper the use the term interfaces to refer to Java classes and interfaces.

[4] In this paper we will use the following words interchangeably: API and a good interface; non-API and a bad interface.

quantitative analysis of the two identified largest groups by looking at source compatibility of the ETPs with Eclipse SDKs. In Sect. 6 we qualitatively analyze the the three remaining groups of ETPs. In Sect. 7 we discuss the threats to validity. In Sect. 8 we discuss the related work and finally, in Sect. 9 we present the conclusions and future work.

2 Eclipse plug-in architecture

Eclipse SDK[2] 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.

We categorize the Eclipse plug-ins into three sets: 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 the ECPs but also may use functionality provided by EEPs.

Eclipse clearly states general rules for extending or using the interfaces it provides. The Eclipse guideline defines two types of interfaces (des Rivières: How to use the Eclipse API 2001): non-APIs ("bad" interfaces) and APIs ("good" interfaces) defined in addition to Java access (visibility) levels. According to Eclipse naming convention (des Rivières: How to use the Eclipse API 2001), 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 (des Rivières 2007). 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, APIs ("good") are found in packages that do not contain the segment *internal*. The 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.

3 ETP collection and classification

For our analysis, we collected ETPs from SourceForge. Since we wanted to have long enough history of the ETPs, 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 the search returned 1,350 hits (collected February 16, 2011).

During the ETP 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 removed two groups of ETPs. First, we excluded *incomplete* ETPs, i.e., ETPs having at least one version containing only binaries but no source files. Versions without source code will interrupt in finding evolution trends of the ETP. Second, we eliminated ETPs that do not import packages from ECPs. These are software programs returned as noise from SourceForge search engine. As mentioned earlier in Sect. 2, 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*, i.e., an ETP is considered to be *clean* if it has been released on SourceForge between January 1, 2003 and December 31, 2010, all its versions contain source files and it contains at least one `import` statement from an ECP. To facilitate replication and follow-up studies we have made the entire collection available on-line (Businge 2013c).

Table 1 shows 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. This 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 1 (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 (Businge et al. 2012b).

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—*good ETPs*;
II ETPs with all versions depending on a non-API—*bad ETPs*;

Table 1 Total numbers of ETPs collected, and the numbers of clean ETPs and their versions

	2003		2004		2005		2006		2007		2008		2009		2010	
	E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
Total																
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	*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

Numbers of the ETPs that were first released in a certain year are typeset in italics. E-ETPs and V-versions

III ETPs with earlier versions dependent solely on APIs and latter versions depending on a non-API—*good–bad ETPs*;

IV ETPs with earlier versions dependent on a non-API and latter versions depending solely on APIs—*bad–good ETPs*;

V ETPs oscillating between versions that depend solely on APIs and versions that depend on a non-API—*oscillating ETPs*.

Results of the classification are shown in Table 2. Figures 1 and 2 shows the graphs of the different classifications.

First, adding the number of ETPs along the diagonals in Table 2 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, Fig. 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. 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

Table 2 Classification of clean ETPs

		2003		2004		2005		2006		2007		2008		2009		2010	
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
I (good)	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 (bad)	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

Table 2 continued

		2003		2004		2005		2006		2007		2008		2009		2010	
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
III (bad–good)	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 (good–bad)	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

Table 2 continued

		2003		2004		2005		2006		2007		2008		2009		2010	
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
V (oscillating)	2003	*3*	*11*	3	10	2	10	3	10	3	11	1	1	1	6	2	5
	2004			*2*	*1*	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

Numbers of the ETPs that were first released in a certain year are typeset in italics

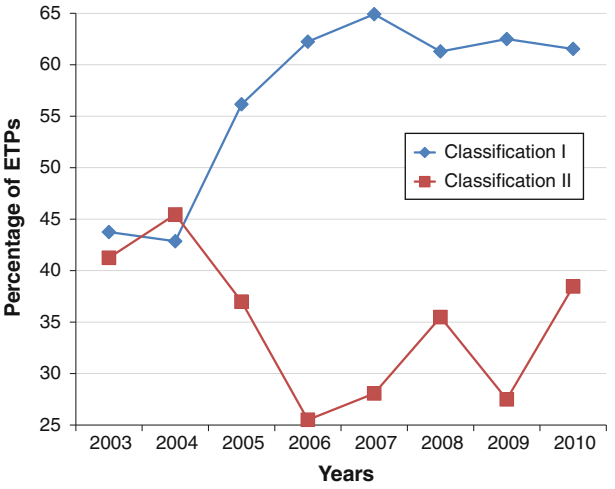

Fig. 1 Percentages of ETPs in Classifications I (*diamond*) and II (*square*). The *Y*-axis scale is normalized by getting the percentage of ETP in each Classification from the total number of clean ETPs in the corresponding year

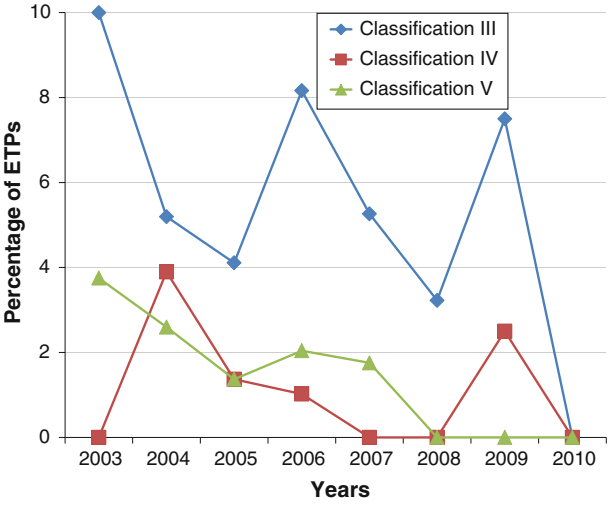

Fig. 2 Percentages of ETPs in Classifications III (*diamond*), IV (*square*) and V (*triangle*). The *Y*-axis scale is normalized by getting the percentage of ETP in each Classification from the total number of clean ETPs in the corresponding year

non-APIs (Classification III) compared to those that eliminate the non-APIs (Classification IV).

In Sects. 4 and 5 we quantitatively compare Classifications I (286 ETPs) and II (181 ETPs) ETPs. Classifications III–V ETPs are qualitatively discussed in Sect. 6 as they contain too few ETPs for quantitative analysis: 32, 6, and 8 ETPs, respectively.

4 Quantitative analysis classification I ETPs versus classification II ETPs

In our previous study (Businge et al. 2010), carried out on a limited sample of carefully selected ETPs, we 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.

4.1 Metrics analysis: classification I ETPs versus classification II ETPs

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 2). 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 2). Since we may have more than one version for an ETP released in 1 year, we select the last version in that year.

4.1.1 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 [Businge 2013a, Appendix B]. Figures 3 and 4 show 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 8 years (2003–2010) and two classifications I and II (Figs. 3, 4, respectively). In the histograms 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.

4.1.2 Hypothesis testing

To verify validity of the observations made in Sect. 4.1.1, 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 $m \in$ {NOF, NOF-D, D-Tot, D-Uniq}, year y, $2003 \leq y \leq 2010$, and data-set $d \in$ {I, 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 (Businge et al. 2010). Similarly, choice for "higher" for D-Tot and D-Uniq is based on a pilot study we carried out.

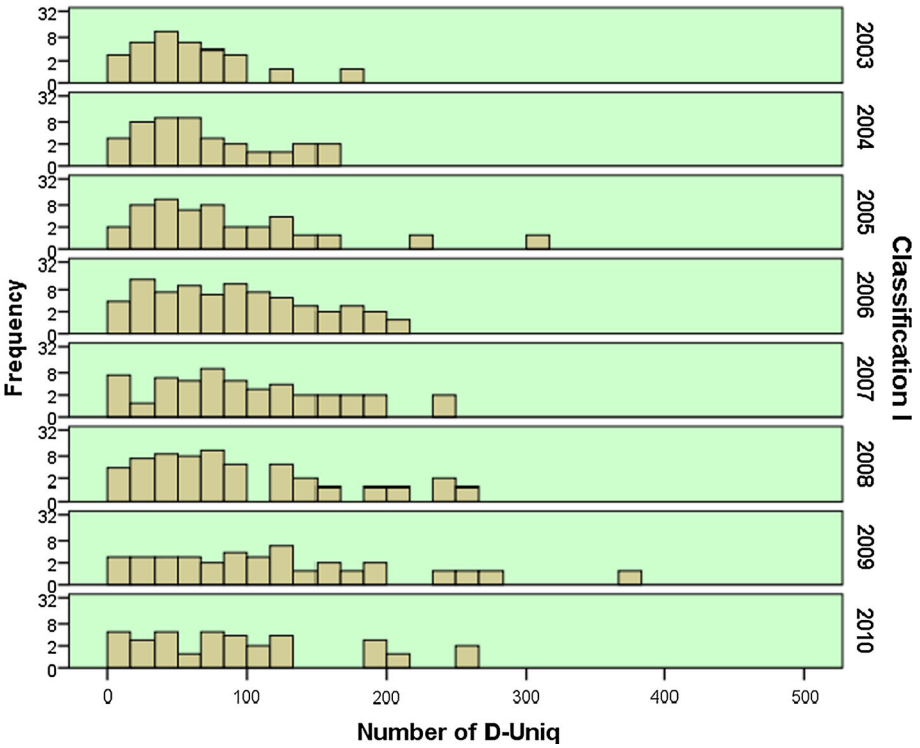

Fig. 3 Distributions of the metric D-Uniq for Classification I ETPs. The *y*-axis is given on a logarithmic scale

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* or a relatively high number of data points (Norušis 2008). In total we have to carry out sixty-four Mann-Whitney tests corresponding to four metrics, two data-sets and 8 years (2003–2010).

Table 3 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. Thus, for most year/metrics combinations in Data-set I we reject the null hypothesis and 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 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 lower 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 only eleven Classification II ETPs 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 this conjecture is considered as future work.

4.2 Metrics analysis: non-API usage

In this section we focus on Classification II and consider numbers of non-APIs used by the ETPs. Figure 5 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 Figs. 3, 4 and 5 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 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^{Tot,y,d_0}: metric values of *D-Tot* and *D-Tot-nonAPI* obtained for ETPs in data-set d and year y represent the same population;
- H^{Tot,y,d_a}: 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^{Uniq,y,d_0}: metric values of *D-Uniq* and *D-Uniq-nonAPI* obtained for ETPs in data-set d and year y represent the same population;
- H^{Uniq,y,d_a}: 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^{Tot,y,d_0} and accept H^{Tot,y,d_a} 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^{Uniq,y,d_0} and accept H^{Uniq,y,d_a} 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.[5] Table 4 summarizes the Hodges-Lehmann estimator values (Hodges and Lehmann 1963; Rosenkranz 2010). The estimator values support the observation made by comparing Figs. 3, 4, 5: although the ETPs in Classification II have a

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

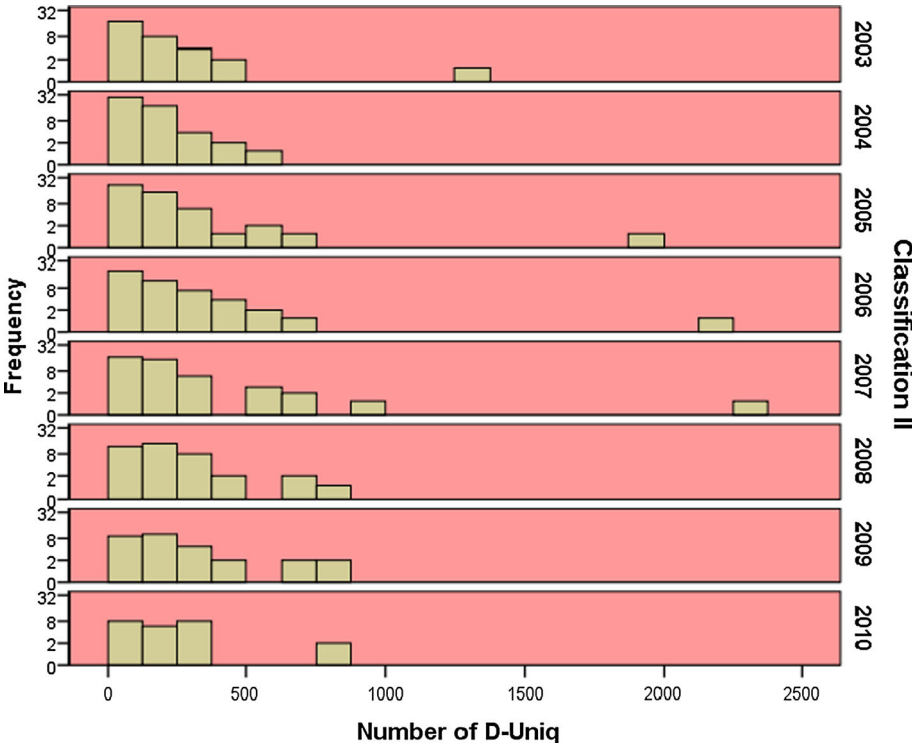

Fig. 4 Distributions of the metric D-Uniq for Classification II ETPs. The *y*-axis is given on a logarithmic scale

high dependency on ECP interfaces, medians in the box-plots reveal that the ETPs have a low dependency on 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 (Fig. 5), wrapping the functionality from non-APIs should be beneficial in case of any changes during the subsequent release of the SDKs.

4.3 Commonly used non-APIs

In this section we continue our discussion of Classification II ETPs and the non-APIs used by these ETPs. We analyze the used non-APIs by ETPs in order to get a quantitative insight of what kind of non-APIs are highly used and how their distribution looks like. This would help inform Eclipse SDK developers the non-APIs that would be good candidates to become 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. Table 5 shows the distribution of the frequency of use of the non-APIs. The distribution is positively skewed: there are many-non-

Table 3 *p*-values for Classification I (CI) and II (CII) in Data-sets I and II

	#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

Zeros indicate values too small to be precisely determined by SPSS. Values exceeding 0.05 are typeset in italics

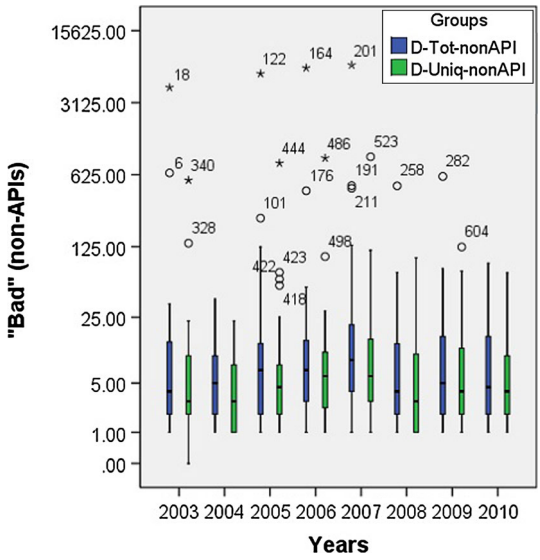

Fig. 5 Distributions of the metrics D-Uniq and D-Tot for non-APIs for Classification II ETPs. The *y*-axis is given on a logarithmic scale. The *stars* and *circles* above the box-plots are outlier cases. The *stars* are extreme cases while the *circles* are less extreme but likely influential cases

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 Table 6 while the complete list is available in [Businge 2013a, Appendix B].

Table 4 Hodges-Lehmann estimation of the median of the difference between the number of (unique) interfaces and the number of (unique) non-APIs

	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 5 Summary of the common non-APIs used by at least two of the 181 ETP-non-APIs

	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

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

By observing Table 6 one might conjecture that the most popular non-APIs are related to Eclipse Java development tools (JDT). This conjecture is confirmed by Fig. 6 showing the distribution of the number of non-APIs used by at least two ETPs across different Eclipse projects. The figure suggests 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. Distribution of the use, i.e., the total number of import statements referring to non-APIs, across different Eclipse projects follows a similar pattern.

5 Compatibility of the ETPs with eclipse SDK releases

In this section we discuss source compatibility of Classification I and II ETPs. Source compatibility analysis of the ETPs complements the findings from the metric analysis in Sects. 4.1 and 4.2.

There are three types of compatibilities between the ETP and the Eclipse SDK releases: (1) *API Binary Compatibility* that requires pre-existing binaries of the ETP to link and run with new releases of the Eclipse SDK without recompiling. (2) *API Source Compatibility* that requires the source code of the ETP needs to be recompiled to keep working with new releases of the Eclipse SDK but no changes have to be made in the sources. (3) The ETP may also be subject to runtime incompatibilities (Dig and Johnson 2009; Wu et al. 2010). We consider API source compatibility and we study compile time errors of the ETP. We plan to analyze binary and runtime incompatibilities in our follow-up research.

Table 6 A sample of commonly used non-APIs by the ETPs ranked according to the highest frequency to lowest frequency

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.ocre.PackageFragment	9
org.eclipse.jdt.internal.ui.wizards.TypedElementSelectionValidator	8
org.eclipse.core.internal.resources.Workspace	8
org.eclipse.jdt.internal.ui.ExceptionHandler	8
org.eclipse.jdt.internal.core.SourceType	8

The column *# ETPs* shows the number of ETPs of the 181 ETPs that use the corresponding non-API

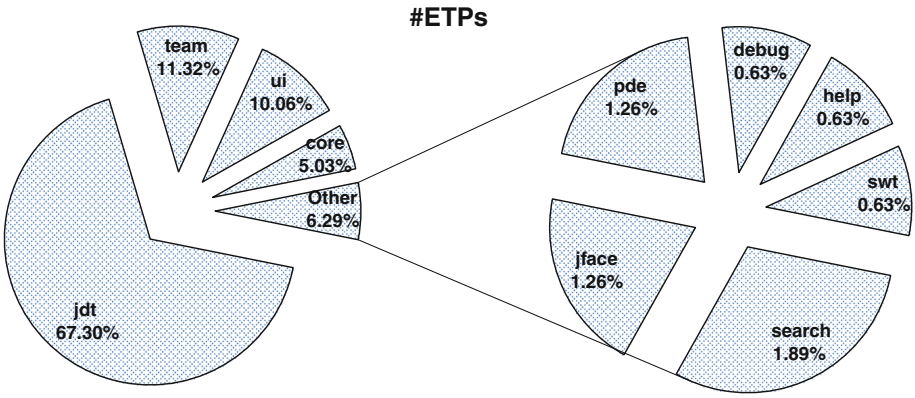

Fig. 6 Distribution of non-API usage per project. The number of non-APIs used by at least two ETPs across different Eclipse projects

ETPs commonly depend on multiple software components such as ECPs, EEPs, external libraries and another ETPs (cf. Fig. 7). We distinguish three types of dependencies that an ETP may have. Compulsory *direct dependency* links an ETP to at least one ECP. Recall that in Sect.3 we have excluded from consideration ETPs that do not depend on ECPs, i.e., all ETPs in our data indeed contain this direct dependency. Optional *direct dependency* is present if an ETP depends on an external library, an EEP or even another ETP. Finally, we talk about an optional *indirect dependency* if an ETP depends on an EEP or another ETP, and these components also depend on an API from ECP. This API is said to be an indirect dependency of the ETP being studied. The study of source compatibility of an ETP is challenging because of the complex structure of the dependencies.

5.1 Classification I and II source compatibility analysis

In this section we recapitulate our previous study about source compatibility of the ETPs in Classification I and II. A detailed analysis can be found in Businge et al. 2012c).

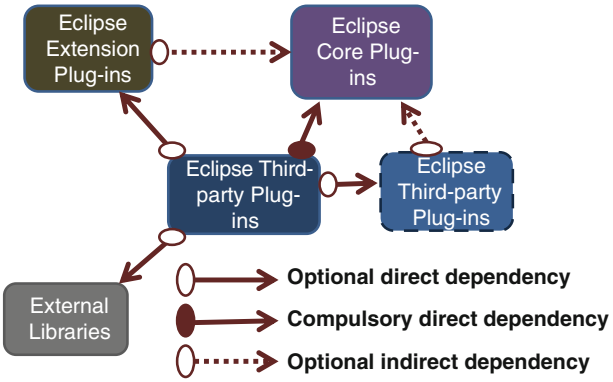

Fig. 7 Dependency structure of an ETP

To check for source compatibility of a version of an ETP with a given SDK release, we employ the methodology described in (Bolour 2003). At compile time, we augment the *build path* for compiling a dependent ETP with the jar files of all the components containing the APIs used in the ETP's source code. If the ETP has direct dependencies on *other components* (i.e., EEP, an external library, or another ETP), the jar files of these components are also included in the ETP's *build path*.

Checking for the appropriate versions of the components that are compatible with the ETP was a manual process due to missing information in the versions of the components. In checking for source compatibility of a given ETP, we focused only on one version of the ETP in each year since it would require a prohibitive effort if all versions of the ETP were considered.

Given the ETP p and a release year y, the manual version identification process follows the steps presented in Fig. 8. After initialization, we identified appropriate versions of the *other components* (Step 2). If no such versions of the *other components* exist we consider the previous version of p, and if there is none, we exclude p from our consideration. Once the appropriate versions of the *other components* have been identified, we check for source compatibility of the current version of p with each of the Eclipse SDK releases (Step 4). We stress that when checking the compatibility with another Eclipse SDK release, the versions of *other components* are kept unchanged in the build path. The rationale for this decision is twofold: (1) we are only interested in the relationship between ETP and ECP interfaces and (2) since the identification of the errors is done manually, only errors related to incompatibilities between the ETP under study and the ECP interfaces will appear in the Eclipse console.

We illustrate the version identification process with an example of an ETP, googlipse 0.5.4. In addition to the ECP dependencies, googlipse depends on two EEPs, J2EE Standard Tools (JST) and Web Standard Tools (WST). Since googlipse 0.5.4 was released in 2007, the versions of the jar files added to the *build path* of googlipse are Eclipse SDK 3.3, WST R-2.0 and JST R-2.0, all released in 2007. When the project is built, 29 errors are reported in the Eclipse console. When we trace the errors in the source code of googlipse, we find that the errors result from an unresolved class dependency from an ECP. This indicates that googlipse 0.5.4 is incompatible with the Eclipse SDK 3.3. The jar files from Eclipse SDK 3.3 are removed and replaced by the jar files of Eclipse SDK 3.2. The SDK replacement removes all errors from the Eclipse console. The same procedure would have

Input: ETP p, release year y

1. Initialization:
   (a) Select the latest version of p in a given release year y.
   (b) For every *other component* add its latest version released in y to the build path of p.
   (c) Add the SDK released in y to the build path of p.
2. Compile the current version of p with respect to the current version of the SDK (including current versions of the *other components*)
   (a) If compilation was successful, go to Step 4;
   (b) Identify the source of compilation errors:
      i. Errors due to a direct dependency on an *other component*: replace in the build path the versions of the components involved by the preceding ones.
      ii. Errors due to a direct dependency on the SDK: replace the SDK by the one released in the preceding year.
      iii. Go to Step 2.
3. If there is another version of p released in y,
   (a) take the preceding version of p, go to Step 2.
   (b) otherwise, exclude p from consideration.
4. For Eclipse SDK release r (1.0 to 3.7) repeat
   (a) Compile p.
   (b) If compilation was successful, report "success(p,r)".
   (c) Identify the source of compilation errors:
      i. Direct dependency on the SDK: report "failure(p,r)".
      ii. Indirect dependency only:
         A. and there is another version of p released in y, take the preceding version of p, go to Step 2.
         B. otherwise, exclude p from consideration.

Fig. 8 Manual version identification: tracing errors to their source (Steps 2b and 4c) is a manual process

been followed if any of EEP dependencies had caused errors. Hence, the compatible versions of the jar files required by googlipse 0.5.4 are found in WST R-2.0 and JST R-2.0. These versions of the jar files are used to determine source code compatibility of googlipse 0.5.4 with different SDK releases.

In (Businge et al. 2012c), we quantitatively analyzed the source compatibility of Classification I and Classification II ETPs. The following are three main observations: Classification I ETPs almost never fail in the subsequent Eclipse SDK releases unless these releases involve API-breaking changes (i.e., SDK releases 1.0, 2.0, and 3.0.). Second, we observed that recently released Classification II ETPs depend more on old non-APIs and less on newly introduced non-APIs. These ETP have a very high source compatibility success rate with new SDK releases. In our previous study (Businge et al. 2012a), we formally proved that old non-APIs are more stable than newly introduced non-APIs. Third, we observed that Classification I ETPs have a high source compatibility compared to Classification II. Moreover, Classification I ETPs have a relatively strong tendency to be forward source compatible compared to Classification II ETPs.

6 Qualitative analysis for classification III–V ETPs

As opposed to the quantitative analysis of Classifications I and II in Sects. 4 and 5, to study Classifications III-V we employ qualitative analysis since the ETPs in these classifications

are too few, i.e., Classification III, 32 ETP, Classification IV, six ETPs, and Classification V, eight ETPs. During the analysis, we investigate possible causes why and how ETPs move between "good" and "bad".

The following metrics will be used to analyze the ETPs in Sects. 6.1–6.3. We use the metrics *NOF* and *NFD* to give us an indication of the ETP we are analyzing. The *Date* metric to give us an indication of the possible SDK on top of which the version was built. The functionality metrics *NnP* and *Int* to give us an indication of the functionality from Eclipse used by a version of an ETP. The metric *NnP* is our main focus in the analysis. Finally, the *Compatibility* metric help us to trace the non-APIs that are problematic in new SDK releases.

- Ver: Version name of the ETP.
- Date: Date of release on SourceForge of a version of an ETP (represented the format *mm/yy*). With respect to the release dates of the SDK, the metric will give us an indication of the possible SDK on top of which the version was built.
- NnP: Number of unique non-APIs used by a given version of an ETP.
- NOF: Number of `.java` files in a given version of an ETP.
- NFD: Number of `.java` files in a given version of an ETP having dependencies on non-APIs. Since we compile the ETPs, we deleted all the unused imports from the source code.
- Int: The unique number of interfaces (good + bad) from Eclipse used by a given version of an ETP.
- Compatibility: This indicates the source compatibility of a given version on an ETP with the SDK releases (0–incompatibility and 1–compatibility).

6.1 Classification III: good to bad ETPs

In this section we qualitatively analyze the 32 ETPs in Classification III in Table 2. Recall that for all the ETPs in this classification, earlier versions dependent solely on APIs (good) and latter versions depending on a non-API (bad). We investigate possible reasons why ETPs change from the "good" category to the "bad" category.

For each ETP, we extracted and analysed the metrics defined in Sect. 6; Tables 7 and 8 present the metrics for two ETPs *Parfumball* and *Eclipse Corba*, respectively. The remaining 30 ETPs show similar trends.

For the versions of the ETPs that depend on only APIs (*good part*), we observe similar findings as those we observed for Classification I ETPs (good ETPs). For the versions of ETPs that have a dependency on non-APIs (*bad part*), we observe similar findings as those we observed in Classification II ETPs (bad ETPs).

For the *good part*, we observed that versions of ETPs that are compatible with a given SDK release continue to be compatible with later releases of the SDK. For the *bad part*, we observed that some of the versions of the ETPs are not source compatible in new SDK releases and those that continue to be compatible we observe that they use old non-APIs.

The reason why ETPs move from good to bad could be related to the Lehmans' law of continuous growth. The law states that the functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime (Lehman and Belady 1985). We observe that as the ETPs evolve, in accordance to the law of continuous growth, there is an increasing trend in the two metrics *Int* and *NOF* in Tables 7 and 8, respectively. Since the ETPs consume more functionality from the SDKs as they evolve, we conjecture that some of the functionality they require may be absent from the good part of the SDK.

Table 7 Parfumball

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.0.1	11/04	52	0	51	0	1	0	0	0	0	0	0	0
0.0.2	11/04	53	0	68	0	1	0	0	0	0	0	0	0
0.0.3	11/04	59	0	71	0	1	1	1	1	1	1	1	1
0.0.4	06/05	70	1	108	1	1	1	0	0	0	0	0	0

Compatibility with SDKs, 0–Compatible and 1–Incompatible

Table 8 Eclipse Corba

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.2.0	07/04	79	0	446	0	1	1	1	1	1	1	1	1
0.3.0	02/05	83	0	462	0	1	1	1	1	1	1	1	1
0.3.1	04/05	91	0	483	0	1	1	1	1	1	1	1	1
0.3.2	05/05	92	0	448	0	0	1	1	1	1	1	1	1
0.4.0	05/06	104	0	454	0	0	0	1	1	1	1	1	1
0.4.1	05/06	104	0	454	0	0	0	1	1	1	1	1	1
0.5.5	02/07	133	1	613	1	0	0	1	1	1	1	1	1
0.5.6	07/07	144	1	619	2	0	0	0	1	1	1	1	1
0.6.0	01/08	147	1	554	1	0	0	0	1	1	1	1	1
0.6.1	02/08	147	1	608	1	0	0	0	1	1	1	1	1
0.7.0	09/08	164	1	648	1	0	0	0	1	1	1	1	1

Our conjecture is strengthened by two ETP developers we interviewed and told us that the reason they use non-APIs is because the functionality they require can only be found in the non-APIs. In our follow-up study, we plan to carry out a survey on Eclipse interface usage with Eclipse solution developers to formally verify our conjecture.

6.2 Classification IV: bad to good ETPs

In this section, we analyze the six ETPs in Classification IV in Table 2. Recall that for all the ETPs in this classification, earlier versions dependent on a non-API (bad) and latter versions depending solely on APIs (good). In the analysis, we investigate possible reasons why and how ETPs change from "bad" category to the "good" category. We investigate why and how the non-APIs are eliminated from the source code of the ETP and how the removed non-API affects the ETP before and after removal.

Tables 9, 10, 11, 12 and 13 present the results of analysis for the ETPs in Classification IV (good to bad)[6]. Since the ETPs are few and also have very few versions, the tables present analysis for all the versions.

[6] One of the ETPs, ShellEd, had one unused non-API import in its early versions. This ETP now belongs Classification II ETPs. Five of the remaining ETPs are analyzed below.

Table 9 Strutsbox

Ver	Date	Int	NnP	NOF	NFD	Compatibility with SDKs							
						3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.0.1	07/04	200	4	220	4	1	0	0	0	0	0	0	0
1.0.2	01/05	205	4	224	4	1	0	0	0	0	0	0	0
1.0.3	05/05	207	4	279	4	1	1	0	0	0	0	0	0
1.0.4	06/05	207	3	290	3	1	1	0	0	0	0	0	0
1.1.0	08/05	192	3	252	3	1	1	0	0	0	0	0	0
1.1.1	10/05	156	2	247	2	1	1	0	0	0	0	0	0
2.0.0	01/06	146	0	235	0	0	1	1	1	1	1	1	1
2.0.1	02/06	147	0	242	0	0	1	1	1	1	1	1	1

6.2.1 Strutsbox

Strutsbox is a visual Eclipse plugin toolkit for developing applications with Jakarta Struts Framework. Strutsbox is 8 years old having a total of eight versions with its latest version released in 2006.

Table 9 presents the analysis results of the ETP Strutsbox. In version 1.0.1 and 1.0.2 in column *Ver*, the class de.strutsbox.ui.editor.p-arts.MessageDialogWithToggle calls a method org.eclipse.ui.int-ernal.WorkbenchMessages.getString. From SDK 3.0 to SDK 3.1, the method org.-eclipse.ui.internal.WorkbenchMessages.getString was removed from Eclipse. As can be observed from Table 9, version 1.0.1 and 1.0.2 fail with SDK 3.1 onwards.

In version 1.0.3, the result of the method org.eclipse.ui.interna-l.WorkbenchMessages.getString was replaced by an empty string (“”). The non-API org.eclipse.ui.internal.WorkbenchMessages was removed from version 1.0.3 and as can be observed it is compatible with SDK 3.1.

In version 1.1.0, the class de.strutsbox.ui.editor.parts.Stat-usDialog calls the method org-.eclipse.ui.internal.MessageLine.-setErrorStatus. In version 1.1.1, the class de.strutsbox.ui.edito-r.parts.StatusDialog was deleted from the plug-in. This is the reason the number of non-APIs reduces from 3 to 2.

In version 2.0.0, there was a major restructuring where the packages de.st-rutsbox.ui.editor, de.strut-sbox.ui.wizards and de.strutsbox-.visualizer.ui, were merged in version 2.0.1 into one package de.strutsb-ox.ui. All the non-APIs were eliminated in the plug-in during the restructuring. We can observe that the two versions are compatible with the SDK releases from SDK 3.1.

6.2.2 Eclipse coding tools

Eclipse Coding Tools is an Eclipse plug-in that adds some small refactoring tools, aimed at fixing logging code. The plug-in is 8 years old, having 11 versions with the latest version released in 2010.

In Table 10 column *Ver*, version 0.3.0 of the ETP accesses the value of the field org.eclipse.jdt.i-nternal.ui.text.IJavaPartitions.JAVA-PARTITIONING. In SDK 3.1, the non-API org.eclipse.jdt.internal-.ui.text.IJavaPartitions was removed from Eclipse. We can observe the decrease in the number of non-APIs used by the ETP from 15 to 14.

Table 10 Eclipse coding tools

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.2.0	10/04	232	15	138	9	1	0	0	0	0	0	0	0
0.3.0	10/04	231	15	142	9	1	0	0	0	0	0	0	0
0.4.0	03/05	229	14	115	9	1	0	0	0	0	0	0	0
0.5.0	12/05	223	14	202	8	1	1	1	1	0	0	0	0
0.6.0	05/10	164	0	172	0	0	1	1	1	1	1	1	1
1.0.0	05/10	70	0	44	0	0	0	0	1	1	1	1	1
1.0.1	05/10	70	0	44	0	0	0	0	1	1	1	1	1
1.0.2	05/10	69	0	44	0	0	0	0	1	1	1	1	1
1.0.3	05/10	69	0	44	0	0	0	0	1	1	1	1	1

Table 11 EuroMath2

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.1.9	09/04	141	1	695	3	1	1	1	1	1	1	1	1
1.2.0	12/04	154	1	775	3	1	1	1	1	1	1	1	1
1.2.1	04/05	132	1	623	1	1	1	1	1	1	1	1	1
1.3.0	10/05	154	0	630	0	1	1	1	1	1	1	1	1
1.3.1	11/05	153	0	630	0	0	1	1	1	1	1	1	1
1.4.0a	08/06	150	0	349	0	0	0	1	1	1	1	1	1
1.4.0	09/06	156	0	344	0	0	0	1	1	1	1	1	1

The removal of the non-API is one of the causes of compatibility problems between version 0.3.0 of the ETP with SDK 3.1–3.7. Version 0.4.0 of the ETP replaced org.eclipse.jdt.inte-rnal.ui.text.lJavaPartitions.JAVA_PAR-TITIONING with "_java_p-artitioning". "_java_partitioning" is the value that is assigned to the accessed field in version 0.3.0 of the ETP (i.e., copied from the non-API in SDK 3.0).

One of the non-APIs org.eclipse.jdt.internal.ui.dialogs.Ty-peSelectionDialog that is used by the ETP was removed in SDK 3.1. In version 0.5.0, all the source code parts that use the removed non-API are replaced by the services of another non-API org.eclipse.jdt.internal.ui-.dialogs.TypeSelectionDialog2. Version 0.5.0 continued to be compatible with the successive SDK releases until SDK 3.4 because the method org.eclipse.jdt.internal.corext.util.JavaModelUtil.getFul-lyQualifiedName, called by version 0.5.0 of the ETP, was deleted.

In version 0.6.0 all the non-APIs disappeared from the ETP as a result of a major restructuring where the package awilkins.eclipse.coding.templ-ates including its sub-packages were removed from the ETP. This package contained the classes that used non-APIs from Eclipse. In version 1.0.0 there was another major restructuring where the package awilkins.objectmodel was removed from the ETP, the reason we can observe the decrease in the NOF. Version 1.0.0–1.0.3 use APIs that were introduced in SDK 3.3 the reason they are not compatible with SDK 3.2 and below.

Table 12 Emonic

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.1.0	01/06	177	4	56	3	1	1	1	0	0	0	0	0
0.1.1	08/06	163	2	54	2	1	1	1	0	0	0	0	0
0.1.2	03/07	183	4	64	4	1	1	1	0	0	0	0	0
0.1.3	10/07	188	2	66	1	1	1	1	1	1	1	1	1
0.3.0	10/07	315	0	128	0	0	0	1	1	1	1	1	1
0.4.0	01/10	414	0	325	0	0	0	1	1	1	1	1	1

Table 13 Eclipse metrics

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.3.6	07/05	168	1	142	2	1	1	1	1	1	1	1	1
1.3.7	10/09	177	1	152	1	0	1	1	1	1	1	1	1
1.3.8	07/10	177	0	152	0	0	0	1	1	1	1	1	1

6.2.3 EuroMath2

EuroMath2 is an Eclipse plug-in that provides a platform for editors editing various XML files with multiple namespaces and also able to manage editors with WYSIWYG capability. The plug-in is 8 year old having the a total of 7 versions. The latest version was released in the year 2006.

As can be see in Table 11, the ETP had one dependency on a non-API from Eclipse in versions 1.1.9 to 1.2.1. This non-API did not cause incompatibility with the different SDKs. From version 1.3.0 onwards, the class with the non-API was deleted from the ETP. The sharp drop of the NOF from version 1.3.3 to 1.4.0 is not interesting for this study since these versions do not have dependency on non-APIs.

6.2.4 Emonic

Emonic (Eclipse-Mono-Integration) is a Eclipse-Plugin for C#. It provides color-highlighting, outline, word-completion and build mechanism via Ant or Nant. The plug-in is 6 years old having a total of six versions. The latest version of the plug-in was released 2010.

From Table 12 column *Ver*, we can observe a decrease in the number of non-APIs from version 0.1.0 to 0.1.1 and again an increase from version 0.1.1 to 0.1.2. In version 0.1.1 the method org.emonic.base.editors.CSharpEdi-tor.editorContextMenuAboutToShow that calls the non-API method org.eclipse.ui.internal.ViewerActionBuilder.readViewerCon-tributions in version 0.1.0 was deleted. The same method with the same implementation was later reintroduced in version 0.1.2. The package org.emonic-.base.actions, in version 0.1.0, that calls the non-API method org.eclips-e.jdt.internal.ui.packageview.PackageExplorerPart.getTree-Viewer was deleted in version 0.1.1. The same package was reintroduced in version 0.1.2.

From Table 12 column *Ver*, we can see a decrease on non-APIs from version 0.1.2 to 0.1.3. Version 0.1.2 calls the non-API constructor org.eclipse.u-i.internal.dialogs.ViewSorter. The non-API org.eclipse.ui.in-ternal.di-alogs.ViewSorter was deleted in SDK 3.3. This caused version 0.1.2 to fail in SDK 3.3 as indicated in Table 12. The failure was fixed in version 0.1.3 by getting rid of the services of the non-API org.eclipse.ui.inte-rnal.dialogs.ViewSorter and building its own API org.emonic.base.-views.OutlineViewerSorter. In version 0.1.3, the ETP replaced the services of the non-API org.eclipse.jdt.internal.ui.packageview.Pa-ckageExplorerPart with the services of the API org.eclipse.jdt.ui.I-PackagesViewPart.

From version 0.1.3 to 0.3.0, two non-APIs are removed the ETP. Version 0.1.3 calls two methods from the non-API org.eclipse.ui.internal.Wor-kbench as a work-around for Eclipse bug 75440 stated as a comment in the ETP's source code. Bug 75440 was reported on 10/2004 and fixed on 04/2005. In version 0.3.0, the ETP no longer uses the services in the non-API org.eclipse.ui.internal.Workbench. Furthermore, version 0.1.3 uses the services of the non-API org.eclipse.ui.internal.ViewerAction-Builder in the method org.emonic.base.editors.CSharpEditor.edi-torContextMenuAboutToShow. The same method in version 0.3.0 uses the services from API org.eclipse.jface.action.Separator but has different implementation.

6.2.5 *Eclipse metrics*

Eclipse Metrics is an Eclipse plug-in that provides metrics calculation and a dependency analyzer that detects cycles in package and type dependencies. The ETP has three versions on SourceForge al released between 2008 and 2009.

In Table 13, the ETP was calling the non-API method org.eclipse.jdt-.internal.core.Openable.getUnderlyingResource in the first two versions and in the last version replaced the non-API method with the API method org.eclipse.jdt.core.IJavaElement.getUnderlyingRe-source.

6.2.6 *Discussion*

From the analysis in Sect. 6.2.1–6.2.5 we can learn a number of lessons. First, when developers stop using the functionality of a given non-API, we have noticed three things:

1. Developers build their own API that has same functionality as the non-API (copy & paste). This occurred three times in the analysis presented. This way they avoid using the unstable non-API between SDK releases. For example, in the *Emonic* ETP the developers replaced functionality of the non-API that was problematic in version 0.1.2 with an own build API in version 0.1.3. From an interview with some ETP developers, re-implemented (copy & paste) code of the non-APIs can be difficult to maintain. Furthermore, the developers told us that copied & pasted code misses out from benefiting for improvements in the non-APIs in new SDK releases.
2. Developers find similar functionality offered by an API in the SDK. This occurred two times in our analysis. For example, in the *Eclipse metrics* ETP, the developer replaced the functionality of the non-API used in the version 1.3.6 and 1.3.7 with functionality of an API in version 1.3.8.
3. Developers completely eliminate the entities in the ETP source code that uses the functionality from the non-API. This occurred four times in our analysis. For example,

in the ETP *EuroMath2* a class that used the non-API one version was deleted in a new version.

Second, we have observed that the reason why the non-APIs are being eliminated from the ETPs' source code is, because (ETP developers believe) these non-APIs will cause incompatibilities when a version of the ETP is ported to new SDK release. The survey of Eclipse developers we have conducted (Businge et al. 2013) confirmed this finding.

6.3 Classification V: oscillating ETPs

In this section we analyze the eight ETPs in Classification V. Recall that these are ETPs oscillating between versions that depend solely on APIs and versions that depend on a non-API. Most of the ETPs in this classification have very many versions. We investigate why the ETPs alternate between the good and bad categories and possible caused of the alternation.

For ETP that alternate from bad–good–bad, we will investigate why they eliminate non-APIs and then reintroduce them. For the ETPs that alternate from good–bad–good, like in Sect. 6.2, we will investigate why they eliminate non-APIs.

Since the the ETPs in this Classification have many versions, the tables as well as in the discussions in this section will present versions where we observe elimination and introduction of non-APIs (reasons for introduction of non-APIs are similar to the ones already mentions in Sect. 6.1–Classification III–Good to Bad ETPs). Tables 14, 15, 16, 17, 18, 19 present the results of the analysis of six of the eight ETPs in Classification V (oscillating ETPs). One of the ETPs did not have dependencies on Eclipse[7] and in ETP *Eclipse ResourceBundle Editor* we did not observe elimination of non-APIs.

6.3.1 ClearCase

ClearCase Eclipse plug-in allow Eclipse users to have access to ClearCase functionality in the Eclipse workbench. The ETP has got 22 versions releases on SourceForge between 2003 and 2010 when we collected this data. Table 14 only shows versions where we observed elimination of non-APIs. ClearCase belongs to the bad–good–bad category.

From Table 14 we observe the disappearance of the 3 non-APIs from version 1.0.3 to 1.1.0. We also observe that version 1.0.3 is compatible with only SDK 3.1 and version 1.1.0 is compatible with SDK 3.2–3.7. From source code inspection, we observed that in SDK 3.1 a non-API package org.eclipse.core.i-nternal.resources.mapping was introduced. Version 1.0.3 started using three classes from this non-API package. In SDK 3.2, the three classes used by version 1.0.3 were graduated to API package org.eclipse.core.resource-s.mapping. Version 1.1.0 now uses these three classes that graduated.

In version 2.1.2 the package net.sourceforge.eclipseccase.ui, that exists in the rest of the versions, is missing. The developer did not commit it to the repository. From version 2.1.4 to the latest version 2.2.6 the non-API reappeared since the package net.sourceforge.eclipseccase.ui reappeared in the source code.

[7] The ETP *gted* had an unused non-API in one of the intermediate versions. This ETP should be categorized as a good ETP.

Table 14 ClearCase

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.0.3	08/05	165	3	68	1	0	1	0	0	0	0	0	0
1.1.0	11/05	166	0	69	0	0	0	1	1	1	1	1	1
2.1.1	08/08	166	2	69	7	0	0	0	1	1	1	1	1
2.1.2	08/08	45	0	16	0	0	0	1	1	1	1	1	1
2.1.4	01/09	164	2	71	2	0	0	0	1	1	1	1	1

Table 15 Eclipse platform extensions

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.2.0	10/03	57	2	12	1	0	0	0	0	0	0	0	0
1.2.1	12/03	55	0	12	0	1	1	1	1	1	1	1	1
3.2.0	02/06	66	2	30	1	0	1	1	1	1	1	1	1
3.2.1	02/06	65	0	57	0	0	1	1	1	1	1	1	1

Table 16 JUnitRunner

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.1.0	10/04	24	1	10	1	1	1	0	0	0	0	0	0
1.1.1	11/05	23	0	10	0	1	1	1	1	1	1	1	1
1.1.2	11/05	28	2	10	2	1	1	1	1	0	0	0	0
1.2.2	09/06	28	2	4	1	0	1	1	1	0	0	0	0
1.3.0	05/08	21	0	3	0	0	0	0	1	1	1	1	1

Table 17 Eclipse Verilog editor

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.5.2	06/07	191	1	74	1	0	1	1	1	1	1	1	1
0.6.0	09/07	227	0	295	0	0	0	0	1	1	1	1	1

6.3.2 Eclipse platform extensions

This is a set of Eclipse plug-ins that provide additional functionality to the Eclipse IDE. The ETP has got 18 versions released on SourceForge between 2003 to 2010. Table 15 versions where we observed elimination of non-APIs. The ETP belongs to the bad–good–bad category.

Table 18 MoreUnit

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.3.0	01/06	69	1	22	1	0	1	1	1	1	1	1	1
0.3.1	01/06	65	0	22	0	0	1	1	1	1	1	1	1
1.2.0	12/08	216	15	97	2	0	0	0	0	1	1	0	0

Table 19 PovClipse

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.6.0	11/06	222	2	325	1	0	1	1	1	1	1	1	1
0.6.1	12/06	249	0	341	0	0	1	1	1	1	1	1	1

From Table 15 we can observe a drop in the number of non-APIs from version 1.2.0 to 1.2.1. Version 1.2.0 uses the non-APIs org.eclipse.ui.intern-al.IPreferenceConstants and org.eclipse.ui.internal.Workbe-nchPlugin. These non-APIs cause the version to fail in SDK 3.0-3.7 as can be observed from Table 15. In version 1.2.1 the non-APIs were eliminated and replaced by two APIs. The implementation of the entities that uses the new APIs in version 1.2.1 is different from the implementation of the same entities that uses the non-APIs in version 1.2.0.

The next three versions from 1.2.0 did not have dependencies on non-APIs. In version 3.2.0, two new non-APIs, org.eclipse.ui.internal.HeapSt-atus and org.eclipse.ui.internal.util.PrefUtil were introduced in the source code. From version 3.2.1 to the latest version 3.6.1, the code that uses the two non-APIs was dropped from the source code and the non-APIs were deleted.

6.3.3 Eclipse resource bundle editor

This is an Eclipse plug-in for editing Java resource bundles. The ETP lists 26 versions, i.e., six version, 0.5.0 to 0.6.0 supported in SDK 2.x (comment on ETPs' SourceForge page) and 20 versions, 0.1.0 to 0.7.7, supported in SDK 3.x (comment on ETPs' SourceForge page). The versions were released between 2004 to 2007.

Indeed we have confirmed from compiling the six versions supported in 2.x that they are only compatible with SDK 2.0 and 2.1. Version 0.6.0 of the six versions has got a dependency on a non-API.

We have also confirmed that the next 20 versions supported in 3.x are compatible with SDK 3.0 to 3.7 and not with earlier SDKs. 11 of the 20 versions, i.e., 0.1.0 to 0.5.4, do not have a dependency on a non-API. Version 0.6.0 has a dependency on a non-API and the next eight versions, i.e., 0.7.0 to 0.7.7 have dependencies on tow non-APIs.

We have discovered that the six versions supported in SDK 2.x have the same version names and release dates as six of the version supported in SDK 3.x. From our observation, since the first version of the ETP was released in December 2004 and SDK 3.0 was released in June 2004, the ETP was initially designed for SDK 3.x. Later the developers

downgraded the six versions, 0.5.0 to 0.6.0, to work with SDK 2.x. This is the reason the ETP is categorised as oscillating. This means that the ETP belongs to the good–bad–good category.

6.3.4 JUnitRunner

This is an Eclipse plug-in that allows the user run/debug Junit test method using context popup menu. The ETP has got nine versions released between 2004 to 2009. We analyze only the versions where we have observed disappearance of non-APIs. The ETP belongs to the bad–good–bad category.

In Table 16, we can observe the drop in the number of non-APIs from version 1.1.0 to 1.1.1. Version 1.1.0 calls the non-API method org.eclipse.jdt-.internal.launching.JavaLaunchConfigurationUtils.getMain-Type. In SDK 3.2 the non-API org.eclipse.jdt.internal.launching.-JavaLaunchConfigurationUtils disappeared. This can be observed in Table 16 that version 1.1.0 fails from SDK 3.2–3.7. In version 1.1.1, a call to the non-API method was changed to a call to an API method org.eclipse.jd-t.core.ICompilationUnit.getJavaProject. As can be observed, version 1.1.1 is compatible with all the SDK releases. The next four versions from 1.1.2 to 1.2.2 have a dependency on two non-APIs, org.eclipse.jdt.inte-rnal.junit.launcher.JUnitBaseLaunchConfiguration and org.e-clipse.jdt.internal.junit.launcher.JUnitLaunchConfigurat-ion. The The ETP accesses two fields org.eclipse.jdt.internal.juni-t.launcher.JUnitBaseLaunchConfiguration.TESTNAME_ATTR and o-rg.eclipse.jdt.internal.junit.launcher.JUnitLaunchConfigu-ration.ID_JUNIT_APPLICATION from the two non-APIs. The two non-APIs used the the ETP were deleted in SDK 3.4. This is the cause of the incompatibilities from SDK 3.4 to 3.7. In version 1.3.0, the methods where the two non-API fields are accessed were deleted from the source code. We can see that version 1.3.0 is compatible with SDK 3.3 to 3.7.

6.3.5 Eclipse Verilog editor

This is an Eclipse plug-in that provides Verilog (IEEE-1364) and VHDL language specific code viewer, contents outline, code assist, etc. The ETP has got 22 versions released on SourceForge in 2004 to 2010. The ETP belongs to the good–bad–good category.

No version has a dependency on non-APIs except two intermediate versions 0.5.1 and 0.5.2. As can be observed from Table 17, there is one non-API in version 0.5.2 and this non-API disappears in version 0.6.0. Version 0.5.2 calls the non-API method org.eclipse.ui.internal.ide.actions.BuildUti-lities.saveEditors. In version 0.6.0 the call to the non-API method was replaced to a call a private method checkAndSaveEditors that was newly introduced in version 0.6.0.

6.3.6 MoreUnit

MoreUnit is an Eclipse plugin that assists developers in writing more unit tests. MoreUnit ETP has released 21 version on SourceFoge from 2006 to 2010. Table 18 only lists versions where we observed elimination of non-APIs. The ETP belongs to the bad–good–bad category.

From Table 18 column *Ver*, the first version, 0.2.0 had no dependency on non-APIs. Version 0.3.0 used one non-API org.eclipse.jdt.inte-rnal.core.JavaElement. However, the non-API used in version 0.3.0 did not cause incompatibilities as can be observed in

Table 18. The code that used the non-API in version 0.3.0 was commented out in version 0.3.1 and the non-API removed from the source code. The next 10 versions 0.3.1–1.1.4 did not use any non-API and the last versions, 1.2.0–2.2.0 used 15 and more non-APIs. In version 1.2.0 we can observe increased functionality used from Eclipse from the number of total interfaces in column *Int*. Like in Sect. 6.1, the possible reason for the use of non-APIs from version 1.2.0 to 2.2.0 could be that the functionality the developer requires is absent from the good part of Eclipse.

6.3.7 PovClipse

PovClipse is an Eclipse editor plugin for Povray (Persistence of Vision Raytracer) scene and include files. The ETP has got 14 versions released on SourceForge from 2006 to 2007. Table 19 only lists version of where we observed elimination of non-APIs. The ETP belongs to the good–bad–good category.

The first version (absent from the table) did not use any non-APIs. The next four versions, 0.4.0–0.6.0 used two non-APIs, org.eclipse.ui.intern-al.WorkbenchPlugin and org.eclipse.ui.internal.about.About-BundleData. The eight most recent versions from 0.6.1–1.1.0 did not have a dependency on a non-API. The functionality of the non-APIs in version 0.4.0–0.6.0 was replaced by functionality of APIs org.eclipse.update.configu-ration.IConfigured-Site, org.ecli-pse.update.configuration.I-InstallConfigurat-ion and org.ecl-ipse.update.configuration.-ILocalSite with a different implementation of these functionalities.

6.3.8 Discussion

From the analysis in Sects. 6.3.1 to 6.3.7, we observe a number things: First we have observed similar findings to those reported in Sect. 6.2.6 for the ETP in Classification IV (bad–good ETPs). Second, we have observed that some non-APIs (classes and interfaces) do graduate to an APIs in a new SDK release, i.e., they are removed from the package with substring internal in the fully qualified package name to one without the substring internal. For example we discovered this while analyzing the ETP *Clearcase* in Sect. 6.3.1, three non-APIs in the package org.eclipse.core.internal.resources.map-ping graduated to package org.eclipse.core.resources.mapping. When this scenario happens, developers just delete the substring internal from the fully qualified class or interface name and the code work.

From the above analysis, the reasons for alternation of ETPs between categories of good and bad are as follows: For ETPs that alternate from good–bad–good, it is because the developers eliminate the problematic non-APIs from the source code. For ETPs that alternate from bad-good-bad, developers eliminate problematic non-APIs from the source code and later they introduce new non-APIs possibly because the functionality they require can only be found in those non-APIs.

7 Threats to validity

7.1 Construct validity

This seeks agreement between a theoretical concept and a specific measuring device or procedure. In our analysis, *construct validity* may be threatened by the metrics analysis in Sects. 4. 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 excluded in the metrics. However, during the source compatibility experiment in Sects. 5 and 6 we reduced this threat by manually deleting the unused imports. Furthermore, during the source compatibility experiment in Sects. 5 and 6, construct validity may be threatened by the grouping of the ETPs based on whether they used non-APIs or not. During the grouping, we relied on the Eclipse naming convention (des Rivières 2007) rather than the API guidelines (des Rivières: How to use the Eclipse API 2001). Noise in our study might have been introduced if software systems deviate from the convention but stick to the guideline.

Finally, still in the source compatibility experiment in Sects. 5 and 6, construct validity may be threatened by operationalizing the construct "survival" into our measurement instruments, i.e., release counts and compilation errors. We opt for an "external" operationalization of survival, as it is perceived by the users of the plug-in as opposed to "internal" operationalization of survival as perceived by the plug-in developers (e.g. the level of activity in code development or mailing lists (Rainer and Gale 2005)). Internal operationalizations focus on the process leading to (non-)survival of a plug-in, while external operationalizations focus on the result, i.e., (non-)survival itself.

7.2 Internal validity

This is related to validity of the conclusion within the experimental context of the ETP collection considered above. We tried to mitigate these threats during the data collection, in Sect. 3, by removing the incomplete ETPs and ETPs with no ECP-depencies. Furthermore, we paid special attention to the appropriate use of statistical machinery.

7.3 External validity

This is the validity of generalizations based on this study. 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. Still, SourceForge seems to loose popularity to more recent forges such as google code and github, and it is possible that different conclusions might be made for these forges. Going beyond Eclipse we realize that the same study needs to be carried out on a different plug-in framework.

8 Related work

This work builds on and extends the previous work on the evolution of ETPs (Businge et al. 2010). In the previous study, we investigated Lehmans' laws of software evolution based on Eclipse interfaces used by ETPs on 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 suggested in (Businge et al. 2010).

This paper is an extension of our previous work on Eclipse API usage (Businge et al. 2012b). We classified ETPs into five different group (Classification I–V) and quantitatively analysed two of the groups, i.e., Classification I and II ETPs that is presented in Sect. 4 of

the current paper. Our contribution extends (Businge et al. 2012b) with the extensive qualitative analysis of Classification III, IV and V ETPs (Sect. 6).

Furthermore, our contribution in Sect. 6 also extends our previous work (Businge et al. 2012a, c). In (Businge et al. 2012c) we quantitatively analysed the source compatibility between Classification I ETPs (good ETPs) and Classification II ETPs (bad ETPs). We observed that the bad ETPs have a high failure rate when ported to new SDK releases. We also observed that the good ETPs almost never fail when ported to new SDK releases that do not include API breaking changes. In (Businge et al. 2012a), based on the Eclipse interfaces used in the ETPs' source code, we built a model to predict compatibility of a version of an ETP developed on top of a given SDK in a new SDK. In comparison to our contribution in Sect. 6, we have discussed compatibility Classification III, IV and V ETPs. We observed that the reason why the non-APIs are being eliminated from the ETPs' source code is, because (ETP developers believe) these non-APIs will cause incompatibilities when a version of the ETP is ported to new SDK release. Furthermore, we observed that developers replace problematic non-APIs with their own API. These observations, however, were solely based on the source code inspection. A complementary approach, recently carried out in (Businge et al. 2013), consisted in conducting a survey of the ETP developers and asking them whether they use non-APIs and why. Results of the survey strengthen the observations made in Sects. 6.1, 6.2.6 and 6.3.8: reasons for elimination of dependencies on non-API are related to (perceived) instability of these interfaces, while reasons for introduction of dependencies on non-API are related to the benets of non-APIs, such as unique functionality, being perceived as more important than their instability.

Finally, this work is part of the Ph.D. thesis of the first author (Businge 2013a), recently summarized in (Businge 2013b).

Besides our own previous work, the current study is related to two categories of existing studies: API usage in software systems and effect of API changes on survivability of framework-based applications.

8.1 API usage in software systems

Mileva et al. study popularity of APIs and state that, analyzing popularity of software projects is a relatively new research field (Mileva et al. 2010). Holmes and Walker (2007) present a prototype tool calculating a popularity measure for every API in a framework. As opposed to our study that spans a period of 8 years, Holmes and Walker focus on one specific version of Eclipse. Mileva et al. (2010) 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 et al. (2011). 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 (Grechanik et al. 2010).

8.2 Effect of API changes on survivability of framework-based applications

Dig and Johnson (2009) state that to better understand the requirements for API migration tools of evolving frameworks, one needs to understand API changes. To that end, the authors studied API changes in new versions of one proprietary and three open-source frameworks and one library. In all the studied systems, the authors discovered that over 80 % of the API-breaking changes are structural, behavior-preserving transformations (refactorings). The implications of the authors' findings confirm that refactoring plays an important role in the evolution of components. Migration tools should focus on support to integrate into applications those refactorings performed in the framework. In comparison to our study, we investigate the impact of API changes in the framework on applications that depend on them.

Other studies related to our work are based on tool support that guides application developers in adapting to API changes in the evolving framework. Nguyen et al. (2010) present a tool, LIBSYNC, that guides developers in adapting API usage code by learning complex API usage adaptation patterns from other clients that already migrated to a new library version as well as from the API usages within the library's test code. The tool can identify changes to API declarations by comparing two library versions, can extract associated API usage skeletons before and after library migration, and can compare the extracted API usage skeletons to recover API usage adaptation patterns. Dagenais and Robillard (2011) present a tool, SemDiff , that suggests adaptations to client programs by analyzing how a framework adapts to its own changes. Using a case study of Eclipse JDT framework and three client programs, SemDiff recommends relevant adaptive changes and detects non-trivial changes. Wu et al. (2010) present AURA that combines call dependency and text similarity analyzes to identify change rules for one-replaced-by-many and many-replaced-by-one methods in a framework. Unlike this line of research, we investigate the effects of API changes on the evolution of framework-based applications.

9 Conclusion 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 conclusions are based on empirical results for data collected on 512 Eclipse third-party plug-ins altogether having a total 1,873 versions. During the analysis, we made a number of observations:

1. We discovered that about 44 % of the 512 Eclipse third-party plug-ins depend on "bad" interfaces and also discovered that developers continue using "bad" interfaces in the new versions of the third-party plug-ins.
2. The subsequent empirical study of 467 plug-ins also showed that plug-ins that use or extend at least one "bad" interface are comparatively larger and also use more functionality from Eclipse than those that use or extend only "good" interfaces.
3. We found out that ETPs use a diverse set of "bad" interfaces, i.e., there are many-non-APIs-used-by-few-ETPs and there are few-non-APIs-used-by-many-ETPs.
4. We also observed that although the Eclipse third-party plug-ins have a heavy dependency on ECP interfaces, the percentage of "bad" interfaces used by the plug-ins is relatively low.
5. We observed that the reason why the non-APIs are being eliminated from the ETPs' source code is (ETP developers believe) these non-APIs will cause incompatibilities when a version of the ETP is ported to new SDK release.

6. When eliminating the use of problematic non-APIs in the ETPs source code, we have observed that developers perform one of the following things: i) build their own API that has same functionality as the non-API, ii) find similar functionality offered by an API in the SDK, iii) completely eliminate the entities in the ETP source code that uses the functionality from the non-API and iv) when a non-API matures into an API, developers replace the functionality of the non-API with the new API.

There are several implications of our findings. First, 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 (cf. Sect. 4) The feedback can be used for planning improved services in new releases of SDKs. Second, 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. Third, developers who must use bad interfaces should use the old ones since we observed that they are more stable. Fourth, ETP developers should avoid re-implemented (copy & paste) code of the non-APIs since it can be difficult to maintain and the (copy & paste) code misses out from benefiting from improvements in the non-APIs in new releases of the SDK.

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 and methods over time. Alternatively, a more fine-grained analysis can be performed by considering the use of non-APIs as reflected in a version control system (Poncin et al. 2011) as opposed to the officially released versions. Second, one should collect more ETPs that uses non-APIs and carry out a related study on the ETPs grouped according to the non-API usage. Third, using econometric techniques (Serebrenik and van den Brand 2010; Vasilescu et al. 2010, 2011) 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. Fourth, we intend to replicate our study on a different repository and also different framework to compare the findings with the current findings. Finally, we plan to replicate the study by employing binary and runtime incompatibilities in the methodology.

References

Bolour, A. (2003). Notes on the Eclipse plug-in architecture (2003). http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html. Accessed 01 Jan 2012.

Bosch, J., Molin, P., Mattsson, M., & Bengtsson, P. (2000). Object-oriented framework-based software development: problems and experiences. ACM Computing Surveys 32.

Brugali, D., Broten, G., Cisternino, A., Colombo, D., Fritsch, J., Gerkey, B., Kraetzschmar, G., Vaughan, R., & Utz, H. (2007) Trends in robotic software frameworks. In: Brugali, D. (Ed.), Software engineering for experimental robotics. Tracts in Advanced Robotics, vol. 30. Springer, pp. 259–266.

Businge, J. (2013a). Co-evolution of the Eclipse framework and its third-party plug-ins. Ph.D. thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

Businge, J. (2013b). Co-evolution of the Eclipse SDK framework and its third-party plug-ins. In: CSMR, pp. 427–430.

Businge, J. (2013c). Eclipse third-party plug-ins source code. doi:10.4121/uuid:ce5e73ba-4087-4a7a-afb1-e442b4b6c0ec.

Businge, J., Serebrenik, A., & van den Brand, M. G. J. (2010). An empirical study of the evolution of Eclipse third-party plug-ins. In: EVOL-IWPSE'10, pp. 63–72.

Businge, J., Serebrenik, A., & van den Brand, M. G. J. (2012a). Compatibility prediction of Eclipse third-party plug-ins in new Eclipse releases. In: SCAM, pp. 164–173.
Businge, J., Serebrenik, A., & van den Brand, M. G. J. (2012b) Eclipse API usage: the good and the bad. In: SQM, pp. 54–62.
Businge, J., Serebrenik, A., & van den Brand, M. G. J. (2012c). Survival of Eclipse third-party plug-ins. In: ICSM, pp. 368–377.
Businge, J., Serebrenik, A., & van den Brand, M. G. J. (2013) Analyzing the Eclipse API usage: Putting the developer in the loop. In: CSMR, pp. 37–46.
Dagenais, B., & Robillard, M. P. (2011). Recommending adaptive changes for framework evolution. ACM Trans. Softw. Eng. Methodol. 20:19:1–19:35.
Dig, D., & Johnson, R. (2009). How do APIs evolve? A story of refactoring. *J. Softw. Maint. Evol. 18*, 83–107.
Grechanik, M., McMillan, C., DeFerrari, L., Comi, M., Crespi, S., Poshyvanyk, D., Fu, C., Xie, Q., & Ghezzi, C. (2010). An empirical investigation into a large-scale Java open source code repository. In: ESEM'10, pp. 11:1–11:10.
Hodges, J. L., & Lehmann, E. L. (1963). Estimates of location based on rank tests. *The Annals of Mathematical Statistics 34(2)*, 598–611.
Holmes, R., & Walker, R.J. (2007). Informing Eclipse API production and consumption. In: OOPSLA'07, pp. 70–74.
Konstantopoulos, D., Marien, J., Pinkerton, M., & Braude, E. (2009). Best principles in the design of shared software. In: COMPSAC'09, pp. 287–292.
Lämmel, R., Pek, E., & Starek, J. (2011) Large-scale, AST-based API-usage analysis of open-source Java projects. In: SAC'11, pp. 1317–1324.
Lehman, M. M., & Belady, L. A. (1985). Program evolution: processes of software change. Academic Press, London.
Mileva, Y. M., Dallmeier, V., & Zeller, A. (2010). Mining API popularity. In: TAIC PART'10, pp. 173–180.
Moser, S., & Nierstrasz, O. (1996). The effect of object-oriented frameworks on developer productivity. Computer 29(9): 45–51.
Nguyen, H. A., Nguyen, T. T., Wilson Jr., G., Nguyen, A. T., Kim, M., & Nguyen, T. N. (2010). A graph-based approach to API usage adaptation. In: OOPSLA, pp. 302–321.
Norušis, M.J. (2008). SPSS 16.0 guide to data analysis. Prentice Hall Inc., Upper Saddle River, NJ.
Poncin, W., Serebrenik, A., & van den Brand, M. G. J. (2011). Process mining software repositories. In: CSMR, pp. 5–14.
Rainer, A., & Gale, S. (2005). Evaluating the quality and quantity of data on open source software projects. In: ICOSS, pp. 11–15.
des Rivières, J. (2001) How to use the Eclipse API. http://www.eclipse.org/articles/article.php?file=Article-API-Use/index.html. Accessed 01 Jan 2012.
des Rivières, J. (2007). Evolving Java-based APIs (2007). http://wiki.eclipse.org/Evolving_Java-based_APIs. Accessed 01 Jan 2012.
Rosenkranz, G. K. (2010). A note on the Hodges-Lehmann estimator. *Pharmaceutical statistics 9(2)*, 162–167.
Schröter, A., Zimmermann, T., & Zeller, A. (2006). Predicting component failures at design time. In: ISESE'06, pp. 18–27.
Serebrenik, A., & van den Brand, M. G. J. (2010) Theil index for aggregation of software metrics values. In: ICSM'10, pp. 1–9.
Tourwe, T., & Mens, T. (2003). Automated support for framework-based software. In: ICSM'03, pp. 148–157.
Vasilescu, B., Serebrenik, A., & van den Brand, M. G. J. (2010). Comparative study of software metrics' aggregation techniques. In: BENEVOL, pp. 1–5.
Vasilescu, B., Serebrenik, A., & van den Brand, M. G. J. (2011). You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In: ICSM'11, pp. 313–322.
Wu, W., Guéhéneuc, Y. G., Antoniol, G., & Kim, M. (2010). AURA: A hybrid approach to identify framework evolution. In: ICSE, pp. 325–334.
Xing, Z., & Stroulia, E. (2006). Refactoring practice: How it is and how it should be supported: an Eclipse case study. In: ICSM'06, pp. 458–468.

Author Biographies

John Businge is a recent PhD graduate at Software Engineering and Technology (SET), in the Computer Science department at Eindhoven University of Technology (TU/e). His research is in the area evolution of software ecosystems. Software ecosystems are collections of software systems, developed and coevolving in the same environment. Prior to joining TU/e John has obtained M.Sc. Computing Science from University of Groningen (The Netherlands) and Bachelor in Computer Science from Makerere University, Kampala (Uganda).

Alexander Serebrenik is an associate professor of software evolution at Eindhoven University ofTechnology. His research interests include software evolution and maintenance, social media, programanalysis, and transformation. Serebrenik received a PhD in computer science from KatholiekeUniversiteit Leuven. He's a member of IEEE and the European Research Consortium for Informatics andMathematics Working Group on Software Evolution. Contact him at a.serebrenik@tue.nl or on Twitter@aserebrenik.

Mark van den Brand is a full professor of model-driven software engineering at Eindhoven University of Technology and also a visiting professor at the Computer Science Department of Royal Holloway University of London (RHUL). He is a president of the European Association for Programming Languages and Systems. His research focuses on model-driven software engineering, model transformation and design of domain-specific languages.