Survival of Eclipse Third-party Plug-ins

John Businge, Alexander Serebrenik, Mark van den Brand
Eindhoven University of Technology
Eindhoven, The Netherlands
{*j.businge,a.serebrenik,m.g.j.v.d.brand*}*@tue.nl*

***Abstract*—Today numerous software systems are being developed on top of frameworks. In this study, we analyzed the survival of 467 Eclipse third-party plug-ins altogether having 1,447 versions. We classify these plug-ins into two categories: those that depend on only stable and supported Eclipse APIs and those that depend on at least one of the potentially unstable, discouraged and unsupported Eclipse non-APIs.**

Comparing the two categories of plug-ins, we observed that the plug-ins depending solely on APIs have a very high source compatibility success rate compared to those that depend on at least one of the non-APIs. However, we have also observed that recently released plug-ins that depend on non-APIs also have a very high forward source compatibility success rate. This high source compatibility success rate is due to the dependency structure of these plug-ins: recently released plug-ins that depend on non-APIs predominantly depend on old Eclipse non-APIs rather than on newly introduced ones. Finally, we showed that the majority of plug-ins hosted on SourceForge do not evolve beyond the first year of release.

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

I. Introduction

Today, many software developers build systems on top of frameworks [1] (e.g., currently Eclipse marketplace[1] reports over 1.4 millions of Eclipse solutions). This approach has many advantages, such as reuse of the functionality provided [2] and increasing productivity [3]. However, these benefits are accompanied by co-evolutionary challenges that come along with using frameworks [4]: as the framework evolves, it makes changes to its APIs and these changes may cause the applications that use them to fail [5]–[7]. Framework-based applications are, therefore, subject to two dimensions of survival: 1) survival related to releasing new versions of the application itself, and 2) survival related to new framework releases[2].

In this work we focus on a popular framework, Eclipse, and study survival of Eclipse third-party plug-ins (ETPs). As opposed to the previous work [11]–[15], we investigate *both* dimensions of survival. For the survival related to releasing new versions of the software system, we study the rate at which new versions are released. Indeed, infrequent release of new versions indicates that a software system is not being actively maintained and hence by Lehman's law of continuous change [16], the system is likely to become unsatisfactory and die. For the survival related to incompatibilities between the software system and the new framework releases, we study source compatibility between the ETPs and the Eclipse SDK releases. To measure source compatibility we count the number of Eclipse SDK releases an ETP can successfully compile with.

Understanding both dimensions of survival is essential for the users of the ETPs and their maintainers. Users of the ETPs need to understand whether the ETP is likely to operate when the Eclipse SDK is updated to a new release, and whether the ETP will produce new versions. Maintainers of the ETPs should be clearly aware of the impact of the changes in the SDK on their plug-in. This understanding is complicated by the fact that while some SDK interfaces, *APIs*, are stable and supported, other SDK interfaces, *non-APIs*, are subject to arbitrary change or removal without notice [17]–[19]. ETP maintainers are strongly discouraged from adopting any of the non-APIs [18], and indeed, it has been shown that many non-APIs are among interfaces that are most likely to introduce a post-release failure [20]. However, despite this, in our previous study we have observed that the use of non-APIs is not uncommon: 44.2% of the ETPs on SourceForge have at least one version that depends on at least one non-API [21].

Therefore, in this paper we focus on the impact of APIs and non-APIs on survival of the ETPs. Both in our study of the release of new versions and in our study of source compatibility, we distinguish between ETPs that solely depend on Eclipse APIs (ETP-APIs) and those that depend on at least one Eclipse non-API (ETP-non-APIs). Differences in survival of ETP-APIs and ETP-non-APIs should be understood by users of these ETPs and taken into account when making a decision whether to use an ETP.

The remainder of the paper is organized as follows: In Section II we introduce Eclipse plug-ins and their interfaces and in Section III we explain the data collection process. Our contributions involve studies of the version release rate (Section IV) and incompatibility with the Eclipse framework (Section V). In Section VI we discuss the threats to validity. In Section VII we review the related work and finally, in Section VIII we present the conclusions and future work.

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

[2]Here and elsewhere our notion of survival is *not* related to a branch of statistics known as survival analysis [8], recently applied to study of software [9], [10]. We do consider application of the survival analysis to Eclipse third party plug-ins as future work.

II. Eclipse Plug-in Architecture

Eclipse SDK is an extensible framework comprising a set of tools working together to support programming tasks. Eclipse SDK is composed from *Eclipse Core Plug-ins (ECP)* providing core functionality of the framework.When the user downloads Eclipse SDK, she downloads all ECPs.

Tool builders can build on the ECPs and contribute to the framework by wrapping their tools in pluggable components. *Eclipse Extension Plug-ins (EEP)* are plug-ins built with the main goal of extending the Eclipse platform. Like the ECPs, fully qualified names of EEP packages start with org.eclipse, but as opposed to the ECPs, the EEPs are not considered to be a part of the Eclipse SDK. An example of an EEPs is J2EE Standard Tools (JST).

Eclipse Third-Party Plug-ins (ETP) are the remaining plug-ins. They use at least some functionality provided by ECPs but may also use functionality provided by EEPs. Many ETPs can be found in open-source repositories, e.g., SourceForge[3]. It is the survival of these ETPs across different versions of Eclipse SDK that we study in this paper.

Specifically, we study how the survival of ETPs depends on the way the ETPs use interfaces provided by ECPs, the APIs and the non-APIs. The non-APIs are ECPs' interfaces intended for internal implementations within the Eclipse framework. They are found in packages with the sub-string internal in the fully qualified names [19]. APIs are provided to the framework users such as developers and maintainers of the ETPs. They are interfaces found in Eclipse packages that do not contain the sub-string internal. A more elaborate discussion of the notions related to Eclipse plug-in architecture can be found in [21].

III. Data Collection

For our analysis, we use part of the data we collected in our previous study [21]. In the previous study, we collected software projects from SourceForge, one of the most popular Open Source repositories. Search for "*Eclipse AND plugin*" on SourceForge returned 1,350 hits of ETPs at the time of data collection (February 16, 2011). Since we wanted to have a long enough history of ETPs supported in the different Eclipse SDK releases 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.

In the previous study [21], we categorized the ETPs according to the year of the first release on SourceForge. ETPs that did not have dependencies on ECPs (i.e., import statements related to ECPs), were excluded since these ETPs will obviously not be affected by new releases of Eclipse SDK. As explained in Section II, the import statements from 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.

[3]www.sourceforge.net

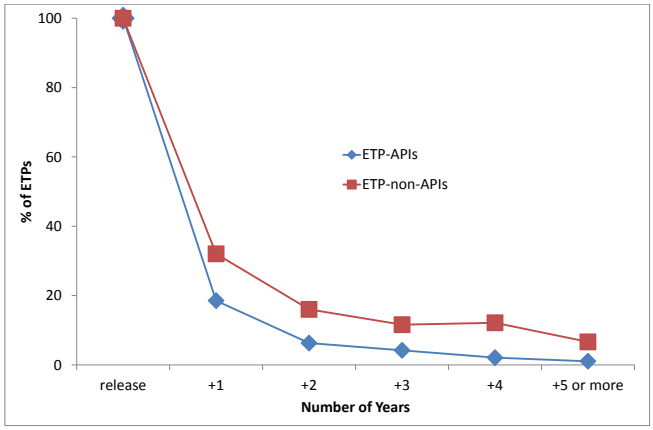
		Released in															
		'03		'04		'05		'06		'07		'08		'09		'10	
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
ETP-APIs first released in	'03	*35*	*62*	10	20	3	4	1	1	1	4	2	2	0	0	0	0
	'04			*33*	*68*	4	11	4	9	2	3	2	2	0	0	0	0
	'05					*41*	*66*	10	21	4	5	3	4	1	1	1	1
	'06							*61*	*111*	7	13	1	1	0	0	2	3
	'07									*37*	*83*	12	22	4	6	6	12
	'08											*38*	*74*	7	12	2	4
	'09													*25*	*30*	3	4
	'10															*16*	*28*
	Total	35	62	43	88	48	81	76	142	51	108	58	105	37	49	30	52
ETP-non-API first released in	'03	*33*	*91*	18	45	11	19	8	13	6	8	2	4	1	2	1	1
	'04			*35*	*77*	8	24	4	9	4	13	4	15	1	2	2	3
	'05					*29*	*60*	10	26	9	31	7	15	7	19	5	20
	'06							*25*	*61*	10	32	3	19	2	11	5	15
	'07									*16*	*31*	2	3	1	2	0	0
	'08											*22*	*42*	7	15	1	3
	'09													*11*	*11*	3	7
	'10															*10*	*11*
	Total	33	91	53	122	48	103	47	109	45	115	40	98	30	62	27	60

Table I: For the given first release year y_1 and an additional release year y the table shows the number of ETPs (E) first released in y_1 and also released in y, and the total number of versions (V) for these ETPs. If $y_1 = y$ we show the number of all ETPs released in this year and their versions.

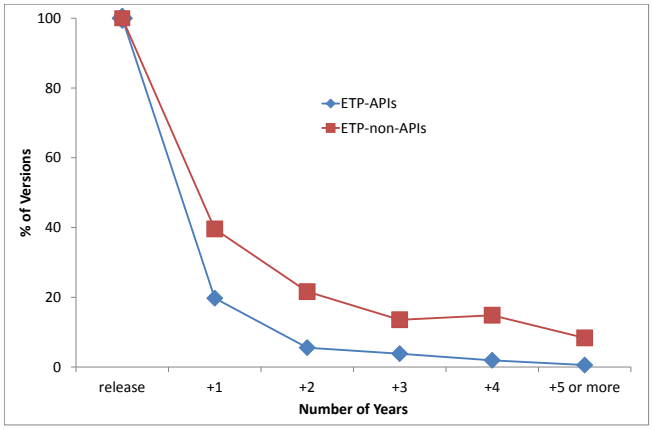
The remaining ETPs were further categorized in [21] into five groups based on presence of non-API dependencies. In the current study we focus on ETP-APIs (Classification I in [21]), i.e., ETPs such that all their versions depended solely on APIs; and ETP-non-APIs (Classification II in [21]), i.e., ETPs that in all their versions have at least one dependency on a non-API. Remaining groups corresponding to ETPs with versions with and without dependencies on non-API, were not considered since the number of ETPs in these classifications is too small for quantitative analysis: it does not exceed 31 as opposed to 286 and 181 in ETP-API and ETP-non-API, respectively. Qualitative analysis of these groups is considered as future work.

Table I shows the data for ETP-APIs (upper part) and ETP-non-APIs (lower part). In the ETP-APIs part, the cell entry ('04,'04), typeset in italics, contains a pair (33 68) indicating that there are a total of 33 ETPs that were first released in the year 2004 on SourceForge altogether having a total of 68 versions. A *version* of an ETP is one of the releases on the SourceForge page of the ETP.

The pair (4 11) in the cell ('04,'05) means that there were 4 ETP-APIs first released in 2004 that had new versions in 2005 with a total of 11 versions in that year. We observe that the trend on the evolution in the number of ETPs is non-monotone, e.g., ('03,'05)=3, ('03,'06)=1, ('03,'07)=1, ('03,'08)=2. This indicates that, while an ETP may have version(s) in a given year, it does not release any version(s)



(a) ETPs. More than 65% of ETP-non-APIs and more than 80% of ETP-APIs did not produce a version in the year following the first release.



(b) Versions. The number of versions released drops in a fashion similar to the number of ETPs, faster for ETP-APIs than for ETP-non-APIs.

Figure 1: Percentage of ETPs that producing new versions (left) and percentage of new versions produced (right).

in the subsequent year(s) but resumes releasing later.

The total number of ETP-APIs is 286 with a total of 687 versions, corresponding to the sum of the lower pair of values in diagonal cells. Similarly, the total number of ETP-non-APIs is 181 having a total of 760 versions.

IV. Release of new ETP Versions

A. Hypothesis Testing

Since non-APIs are subject to change and removal without notice, we expect ETP-non-APIs to fail more often compared to ETP-APIs when ported to new releases of Eclipse SDK. Therefore, it is expected that ETP-non-APIs will have more versions released to fix the incompatibilities compared to ETP-APIs. We, thus, test the following hypotheses:

- H_0^e: *ETP-APIs and ETP-non-APIs represent two populations with equal median values for the number of ETPs releasing new versions*;
- H_a^e: *ETP-non-APIs represents a population with higher median number of ETPs releasing new versions than ETP-APIs*;
- H_0^v: *ETP-APIs and ETP-non-APIs represent two populations with equal median values for the number of new versions released by ETPs*;
- H_a^v: *ETP-non-APIs represents a population with higher median number of new versions released by ETPs than ETP-APIs*.

B. Results and Discussion

Table II shows the numbers of ETPs and their versions released after i years after the initial release. This number can be determined as the sum of the number of ETPs on the ith diagonal in Table I, assuming the main diagonal has number zero. For instance, 53 ETPs for ETP-APIs and $i = 1$ are obtained as the sum of 10, 4, 10, 7, 12, 7 and 3.

		release	+1	+2	+3	+4	+5 or more
ETPs	ETP-API	286	53	18	12	6	3
	ETP-non-API	181	58	29	21	22	12
Versions	ETP-API	522	103	29	20	10	3
	ETP-non-API	384	152	83	52	57	32

Table II: Number of ETP-APIs, ETP-non-APIs and their versions per year after the initial release.

Fig. 1-(a) shows a plot of the percentage of ETPs that release new versions each year since they were first released on SourceForge. For both ETP-APIs and ETP-non-APIs, the number of ETPs with new versions are normalized on the total number of ETPs that were first released on SourceForge. Similarly, Fig. 1-(b) shows the percentage of new versions of ETPs released each year. The number of new versions is normalized with respect to the total number of versions that were first released on SourceForge (sum of the numbers of versions in the main diagonals in Table I).

We observe that both Fig. 1-(a) and Fig. 1-(b) exhibit a clear decreasing trend. The decrease is sharpest from the initial release year to the subsequent year. This indicates that more than 65% of ETPs are not maintained immediately after the first year of their release. The observation supports the earlier findings [15], [22] that SourceForge projects have a low chance of evolving. Study of ETPs hosted at other repositories is considered as a future work.

Furthermore, we observe a sharper decrease in the numbers of ETPs releasing new versions and versions of ETP-API than in ETP-non-API. This observation is supported by the statistical analysis of Table II. By performing a chi-square test we can on any reasonable threshold reject the hypothesis of independence of the number of ETPs releasing new versions (p-value $= 2.036 \times 10^{-7}$) and of the number of versions released (p-value $< 2.2 \times 10^{-16}$) from whether

an ETP is an ETP-API or an ETP-non-API. Moreover, the Wilcoxon testallows us to confidently reject (on any reasonable threshold) both H_0^e (p-value $= 1.094 \times 10^{-9}$) and H_0^v (p-value $< 2.2 \times 10^{-16}$), and, accept H_a^e and H_a^v.

One possible reason for lower values in ETP-APIs may be that developers of ETP-APIs are not forced to release new versions to keep up with changes in Eclipse SDK APIs. As opposed to ETP-APIs, ETP-non-APIs can be expected to fail more often when ported to new releases of Eclipse SDK compared to ETP-APIs due to the use of unsupported and unstable non-APIs. We therefore conjecture that ETP-APIs have a higher *forward source compatibility success rate* when ported to newer releases of Eclipse SDK compared to ETP-non-APIs. We verify this conjecture in Section V.

V. Incompatibilities between ETPs and Eclipse

In this section, we discuss the survival related to incompatibilities between the ETPs and Eclipse SDK. To measure survival of an ETP developed on top of a given major Eclipse SDK release we determine the number of subsequent major Eclipse SDK releases that the ETP is compatible with (forward compatibility). Eclipse distinguishes between a number of compatibility notions [19], [23]. *API Binary Compatibility* requires that pre-existing binaries of the ETP link and run with new releases of the Eclipse SDK without recompiling. *API Source Compatibility* requires that 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. Furthermore, the ETP may also be subject to runtime incompatibilities [7], [24]. 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.

A. Dependency Structure of an ETP

ETPs commonly depend on multiple software components such as ECPs, EEPs, external libraries and another ETPs (cf. Fig. 2). 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 Section III we have excluded from consideration ETPs that do not depend on ECPs. 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.

B. Source Compatibility Check

To study source compatibility, we need to compile ETPs with different releases of Eclipse SDK. Initially, we were interested in comparing the success rate of ETP-APIs and

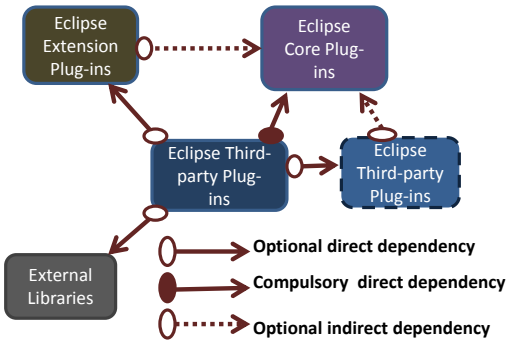


Figure 2: Dependency Structure of an ETP

ETP-non-APIs when compiled with new Eclipse SDK releases, i.e., Eclipse SDK releases following the Eclipse SDK release on top of which the ETP has been built. This would require determining on top of which Eclipse SDK release a given ETP has been built. Unfortunately, less than 5% of the collected ETPs explicitly stated this information either by mentioning it in the SourceForge description or by recording it in the meta-data section of the manifest file of the ETP. The release year of the ETP can be used to determine *terminus ante quem*[4] of the Eclipse SDK release, but cannot be seen as the exact date. Indeed, there may be a time gap between beginning of the development process phase and choice of the SDK, and the end of the development process phase and publication of the ETP at SourceForge. Moreover, we have observed that some programmers prefer to develop plug-ins on top of earlier releases rather than on top of the most recent one, e.g., if those earlier releases are being perceived as being more stable.

Hence, we decided to check the source compatibility of the ETPs with *all* Eclipse SDK major releases.

To check for source compatibility of a version of an ETP with a given ECP release, we employ the methodology described in [25]. 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*. The information about the components *required by* the ETP can be found in the ETP's manifest file. However, information about the appropriate versions of the components is usually missing.

A promising approach to automatic identification of the component versions is *software bertillionage* [26]. To perform the identification bertillionage requires the jar used in compiling the ETP and corpus of all versions of the jar. Unfortunately the ETPs we collected were not bundled with the Eclipse SDK jar files used to compile the ETP, rendering

[4](Lat.) the latest possible date of an event or an object.

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.

Figure 3: Manual version identification: tracing errors to their source (Steps 2b and 4c) is a manual process.

bertillionage inapplicable. Thus, version identification is essentially a manual process.

Version identification would require a prohibitive effort if all versions of an ETP have been considered: indeed, each version of the ETP might require a completely different set of versions of the components. Thus, we focus only on one version of the ETP in each release year. Given the ETP p and a release year y, the manual version identification process follows the steps presented in Fig. 3.

After initialization, we identify 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 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.

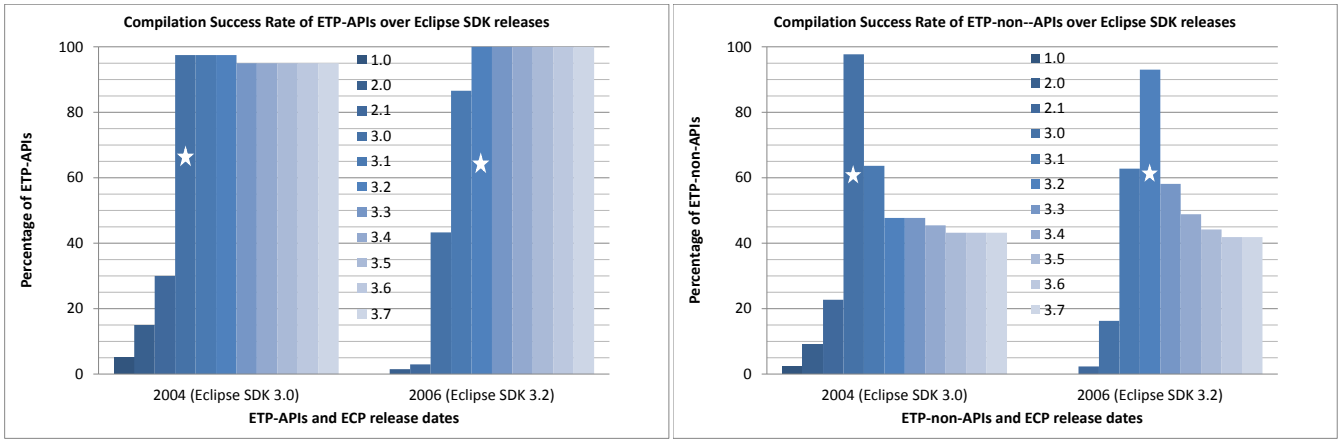
We illustrate the version identification process with an example of an ETP, googlipse 0.5.4. In addition to the ECP dependencies, googlipse depends on two EEPs, J2EE Standard Tools (JST) and Web Standard Tools (WST). Since googlipse 0.5.4 was released in 2007, the versions of the jar files added to the *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 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 addition to swapping the components an ETP depends on, we also had to select the appropriate Java JDK that is compatible with the ETP. Since the ETPs' development dates back as far as the year 2003, the Java JDKs that were used on the different ETPs range from 1.3 to 1.6.

We call an ETP source compatible with a given Eclipse SDK release if no compilation errors are reported, and source incompatible if at least one error related to the *compulsory dependencies* is reported.

C. Results

Table III and Fig. 4 present the results of the source compatibility experiments. The numbers of ETP-APIs and ETP-non-APIs considered in the experiments are those in the *Total* rows in the upper and lower part of Table I. For example, consider the data from Table III for ETP-APIs released in 2003. 32 ETP-APIs are source compatible (**SC**) at least once with all the components they depend on, 4 of the 32 are source compatible with Eclipse SDK 1.0, 19 of



(a) ETP-APIs

(b) ETP-non-APIs

Figure 4: Backward and forward source compatibility success rate of the ETPs released in 2004 and 2006 with all the releases of Eclipse SDK considered.

		SDK releases											SC	SIC
		1.0	2.0	2.1	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7		
		'01	'02	'03	'04	'05	'06	'07	'08	'09	'10	'11		
ETP-APIs released in	'03	4	19	*31*	27	27	27	27	27	27	27	27	**32**	3
	'04	2	6	12	*39*	39	39	38	38	38	38	38	**40**	3
	'05	0	1	2	34	*41*	41	41	41	41	41	41	**41**	7
	'06	0	1	2	29	58	*67*	67	67	67	67	67	**67**	9
	'07	0	0	1	13	22	41	*46*	46	46	46	46	**47**	4
	'08	1	1	2	10	16	34	50	*55*	55	55	55	**55**	3
	'09	0	0	0	3	5	12	30	34	*35*	35	35	**35**	2
	'10	0	0	0	2	6	11	22	27	28	*28*	28	**28**	2
ETP-non-APIs released in	'03	2	9	*27*	15	8	7	6	6	6	6	6	**28**	5
	'04	1	4	10	*43*	28	21	21	20	19	19	19	**44**	9
	'05	1	1	2	21	*44*	23	22	22	22	22	21	**45**	3
	'06	0	0	1	7	27	*40*	25	21	19	18	18	**43**	4
	'07	0	0	1	6	16	28	*32*	24	21	20	20	**38**	7
	'08	0	0	0	1	7	19	27	*33*	30	30	28	**36**	4
	'09	0	0	0	0	1	7	19	23	*25*	23	23	**25**	5
	'10	0	0	0	1	5	7	10	18	21	*19*	19	**23**	1

Table III: The number of ETP-APIs (above) and ETP-non-APIs (below), source compatible with the Eclipse SDK releases. The diagonal cells (shaded gray) show the number of ETPs that are source compatible with the Eclipse SDK released in the same year as the ETPs. The upper and lower triangles correspond to forward and backward source compatibility, respectively.

the 32 are source compatible with Eclipse SDK 2.0 and 31 of 32 are source compatible with Eclipse SDK 2.1.

For each ETP release year, the bars in Fig. 4 corresponding to the Eclipse SDK releases (shown in the *legend* of the graphs) show the percentage of source compatible ETPs with respect to the total number of source compatible ETPs (column SC in Table III). For the sake of illustration we choose two ETP release dates, 2004 and 2006; bar charts corresponding to other release years have similar shapes.

Starting with the Eclipse SDK release corresponding to the release year of the ETPs (starred bar), for each year, we notice that most ETP-APIs are forward source compatible but not backward source compatible. In contrast, ETP-non-APIs are neither forward source compatible, nor backward source compatible with the Eclipse SDK.

D. Discussion

1) ETP-APIs: According to the *Provisional API guidelines* [17], ETPs that follow the guidelines are not supposed to fail in the subsequent Eclipse SDK releases. In addition, the *Version Numbering Document* [27], describing the guidelines of how to evolve Eclipse SDK versions in the subsequent releases states that Eclipse SDK version numbers are composed of two integers named major.minor: the major segment indicates *breaking change* in the API, the minor segment indicates *non-breaking change* in the API ("externally visible" changes). A breaking change violates both the ETP's binary and source compatibility (cf. Section V). For our study this means that ETP-APIs should not fail in the subsequent Eclipse SDK releases that do not involve breaking changes. To check whether this guideline is indeed being adhered to we take a closer look at Table III.

By inspecting Table III, we observe that four ETP-APIs released in 2003 failed from Eclipse release-3.0 to release-3.7 (i.e., there were 31 source compatible ETPs in cell ('03,21) and 27 source compatible ETPs in cell ('10,3.7)). This observation does not contradict the guideline. Indeed, Eclipse SDK 2.1 was released in 2003, i.e., the ETPs released in 2003 were based on an Eclipse SDK release not later than 2.1. According to the version numbering, there were breaking changes between Eclipse SDK 2.1 and Eclipse SDK 3.0. Hence following the guideline some ETP-

APIs might have become source incompatible, and this indeed happened for four ETP-APIs.

One ETP-API, OracleExplorer, released in 2004 failed from Eclipse SDK 3.3 throughout to 3.7. From Eclipse SDK 3.2 to Eclipse SDK 3.3, the Eclipse version numbering does not indicate breaking changes and, therefore, the failure of OracleExplorer indicates a guideline violation. Since Eclipse SDK 3.3, org.eclipse.ui.part.MultiPageEditorPart, extended by one of the OracleExplorer classes, contains a final method setActiveEditor(IEditorPart targetEditor). This method has the same signature as setActiveEditor(IEditorPart editorPart), one of the methods of OracleExplorer itself. Therefore a conflict is introduced as a final method cannot be overridden.

2) ETP-non-APIs: In Table III, looking at the source compatibility success rate of the ETP-non-APIs, we observe that the forward source compatibility improves every year for ETPs released in 2007 to 2010. On further investigation of the possible cause of this phenomenon, we discovered that the majority of non-APIs used by these ETPs were introduced in the early Eclipse SDK releases. For example, we found out that the 19 ETPs classified in release year 2010 altogether used a total of 127 non-APIs of which 28% were introduced in Eclipse SDK release-1.0, 30% in Eclipse SDK 2.0 and none of the newly introduced non-APIs in Eclipse SDK 3.5 and 3.6 was used by any of the ETPs. This could mean that older non-APIs are relatively stable, and this is why the ETPs that use them do not fail. This conjecture has been formally studied in a follow-up paper [28].

3) ETP-APIs vs ETP-non-APIs: Comparing Figs. 4-(a) and 4-(b) it becomes apparent that ETP-APIs have a very high source compatibility success rate compared to the ETP-non-APIs. This can be expected, since ETP-non-APIs use unstable non-APIs and hence are more likely to be affected by changes in Eclipse. For *backward* source compatibility we look at the source compatibility success rates from the Eclipse SDK released in the same year as the ETPs back to Eclipse SDK 1.0. We observe that for both ETP-APIs and ETP-non-APIs, the percentage of source compatible ETPs rapidly decreases. We also observe that the decrease rate is much sharper for ETP-non-APIs compared to ETP-APIs. For *forward* source compatibility we look at the source compatibility success rates from the Eclipse SDK released in the same year as the ETP up to the most recent Eclipse SDK release considered (3.7). ETP-non-APIs have a very high failure rate compared to ETP-APIs.

E. Quantifying the Survival of ETPs

While Section V-D informally discussed differences in the forward source compatibility success rate between ETP-APIs and ETP-non-APIs, in this section we augment the preceding discussion with a formal statistical study. Recall that at the beginning of Section V we stated a general definition of survival of an ETP developed on top of a given *major Eclipse SDK release* as the number of subsequent major Eclipse SDK releases that it can successfully compile with (forward source compatibility).

	Compilation Response		
	Failure (1)	**Survival (2)**	**Total**
ETP-non-APIs	672	678	1,350
	49.8%	50.2%	100.0%
ETP-APIs	52	1,512	1,564
	3.3%	96.7%	100.0%
Total	724	2,190	2,914
	24.8%	75.2%	100.0%

Table IV: ETPs forward-source-compatibility contingency table based on the total compilation responses

Because we lack the information about the specific Eclipse SDK releases that were used when developing the ETPs, in this section we redefine *survival* of an ETP. Since the release year of the ETP can be used to determine *terminus ante quem* of the Eclipse SDK release, we define survival of an ETP as the number of major Eclipse SDK releases that the ETP can successfully compile with, starting with the Eclipse SDK release in the year after the release of the ETP (i.e., the count of Eclipse SDK releases after the release corresponding to the cell in the shaded diagonal in Table III). *Failure* of an ETP is the opposite of *survival*, i.e., the number of major Eclipse SDK releases that an ETP fails to compile with, starting with the Eclipse SDK release in the year after the release of the ETP. In the data-set all ETPs that were source compatible with one Eclipse SDK release and failed in the next Eclipse SDK release also failed with all the Eclipse SDK releases released later.

Table IV is the contingency table of survivals and failures for ETP-APIs and ETP-non-APIs based on Table III. Observe that although the number of ETPs that failed are not shown in Table III it can easily be computed. For example, of the ETP-non-APIs released in 2003 in Table III, a total of 28 ETPs were source compatible. With respect to Eclipse SDK 3.0 we have a survival of 15 ETPs in cell ('03,3.0). The failure with respect to Eclipse SDK 3.0 is of 13 ETPs (difference between the total, 28, and the survival, 15).

We compare the *overall* survival between ETP-APIs and ETP-non-APIs. First, we test the independency of whether an ETP is source compatible of it being an ETP-API or an ETP-non-API. Second, we test the impact of non-APIs in ETPs on the forward compatibility success rate. All statistical calculations have been carried using popular statistical software R [29].

Independence. We test the following hypotheses:

- H_0^i: *The compatibility success rate is independent on whether an ETP is an ETP-API or ETP-non-API*;
- H_a^i: *The compatibility success is dependent on whether an ETP is an ETP-API or ETP-non-API.*

For the independency test, the results of both the chi-square test and Fisher's exact testlead to the p-value $< 2.2 \times 10^{-16}$. Hence, we can guarantee statistical significance on any

reasonable threshold, confidently reject H_0^i and claim that the compatibility success is dependent on whether an ETP is an ETP-API or ETP-non-API.

Impact. Based on the Eclipse guideline [19] we expect ETP-APIs have higher forward compatibility success rate:

- H_0^c: *ETP-APIs and ETP-non-APIs represent two populations with equal median values for forward compatibility success*;
- H_a^c: *ETP-APIs represents a population with higher forward compatibility success median value than ETP-non-APIs*.

To test the hypotheses, we imposed an ordinal scale on compatibility failure (1) and success (2) as suggested in [30], and performed the Wilcoxon test. The test statistic equals 1546104 and the p-value $< 2.2 \times 10^{-16}$. As above, we can guarantee statistical significance on any reasonable threshold, confidently reject H_0^c and claim that ETP-APIs have higher forward compatibility success than ETP-non-APIs.

VI. Threats to Validity

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

Construct validity seeks agreement between a theoretical concept and a specific measuring device or procedure. In our analysis, 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 [22]). 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. Furthermore, construct validity may be threatened by the grouping of the ETPs into ETP-APIs and ETP-non-APIs. During the grouping, we relied on the Eclipse naming convention [18] rather than the API guidelines [19]. Noise in our study might have been introduced if software systems deviate from the convention but stick to the guideline. When checking for source compatibility we have considered only one version of an ETP per year and as such our notion of compatibility success might have been too restrictive. Furthermore, for the decrease in the release of new versions of ETPs, it is possible that developers migrated to repositories that have become more popular, e.g., github.

Internal validity is related to validity of the conclusion within the experimental context of the ETP collection considered above. We have paid special attention to the appropriate use of statistical machinery, e.g. in Section V-E.

External validity is the validity of generalizations based on this study. Our results may not be generalizable beyond the specific collection of ETPs since we only considered ETPs from SourceForge, and, therefore, necessarily only open-source ETPs. To ensure that our results can be generalized one has to replicate the study above with respect to ETPs from other open-source repositories as well as commercial ETPs. Going beyond Eclipse we realize that the same study needs to be carried out on a different plug-in framework.

VII. Related Work

This work complements our previous work on Eclipse API usage [21] and evolution [31]. In [31] 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 distinguishing between ETP-APIs and ETP-non-APIs. In [21] we investigated the Eclipse API usage by 512 ETPs, and proposed the distinction between ETP-APIs and ETP-non-APIs. The current study is one of the follow-up studies proposed in [21]. In a follow-up paper [28] we formally verified the conjecture of the relation between the age of the non-APIs used and compilation success of an ETP, and building on the insights of the current work we have developed statistical models for predicting compatibility of ETPs in new releases of Eclipse.

Besides our own related work, the current study is related to two categories of existing studies: survival due to release of new versions of applications and effect of API changes on survivability of framework-based applications.

A. Survival due to release of new versions of applications

Rainer and Gale [22] studied Sourceforge.net and concluded that the majority of the projects hosted there should be considered as failures, which, according to them, is the absence of activity in code development, mailing lists and bug reporting. Weiss [32] defined success of a project in correlation with its popularity on the web. English and Schweik [33] classifies project into success categories according to the number of its releases and the time between these releases. Majority of projects hosted at SourceForge have been observed to be abandoned or in an early stage [32], [33]. As opposed to [22], [32], [33], we studied projects from a specific domain, Eclipse plug-ins. Similarly to [32], [33] we observe that majority of projects get abandoned on the early stages and refine this observation by relating abandonment to presence or absence of non-APIs.

In [15] authors examined large repositories of open source projects: Sourceforge.net, KDE, GNOME, Rubyforge and Savannah. They have observed that repositories like SourceForge, Savannah and Rubyforge with more "relaxed" preconditions in hosting a project, contain projects with low activity. The authors showed that when a project enters a more "controlled" repository, like the KDE or GNOME, it has better chances to become successful, with success defined as growth in size and attention gained from developers.

B. Effect of API changes on survivability of framework-based applications

Dig and Johnson [24] 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. [5] 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 [6] 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. [7] 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 currently investigate the effects of API changes on survival of framework-based applications. Developing a tool based on the current findings is considered as future work.

VIII. Conclusion and Future Work

In this study, we investigated two notions of survival of Eclipse third-party plug-ins (ETPs): survival related to release of new versions and survival related to source compatibility of the plug-ins with Eclipse SDK releases. While understanding both dimensions of survival is essential both for the users of the ETPs and for their maintainers, it is complicated by the fact that while some SDK interfaces, *APIs*, are stable and supported, other SDK interfaces, *non-APIs*, are subject to arbitrary change or removal without notice. Therefore, for both notions of survival, we compared the trend followed by ETPs that depend on *only* stable and supported Eclipse APIs (ETP-APIs) and that followed by ETPs that depend on at least one of the unstable, discouraged and unsupported Eclipse non-APIs (ETP-non-APIs).

For the survival related to release of new versions, we observed that the majority of ETPs do not survive beyond the first year of release. We also observed that the rate at which new versions are released for ETP-non-APIs is higher than that of ETP-APIs. For the survival related to source compatibility of the ETPs we made a number of observations: First, we observed that ETP-APIs almost never fail in the subsequent Eclipse SDK releases unless these releases involve API-breaking changes. Second, we observed that recently released ETP-non-APIs depend more on old non-APIs and less on newly introduced non-APIs. These ETP-non-APIs have a very high forward source compatibility success rate. Third, we observed that ETP-APIs have a high source compatibility compared to ETP-non-APIs. Moreover, ETP-APIs have a relatively strong tendency to be forward source compatible compared to ETP-non-APIs.

Our results show that developers who commit their ETPs on SourceForge are not committed to updating their plug-ins. Next, we confirm that as stated by Eclipse, APIs are stable over subsequent Eclipse releases that do not involve API-breaking changes. We further confirm that non-APIs are indeed unstable. ETP developers should avoid the use of non-APIs as much as possible.

In our previous study [21] we discovered that a large number of developers use non-APIs and in the current study we observed that non-APIs influence survival of ETPs. Hence, it is equally important for the framework developers to document both the non-API changes and the API changes.

The results presented so far are interesting and have given us a number of directions for the follow-up work: First, we plan to carry out a survey so as to get first hand information from ETP developers on the factors that impact survival of the ETPs. Second, we plan to build a refactoring tool that will aid developers who use non-APIs. We also intend to complement the current analysis of ETP-APIs and ETP-non-APIs by a qualitative analysis of a smaller group ETPs identified in [21], i.e., plug-ins that introduced and/or eliminated dependency on a non-API during their evolution. One might further refine our distinction between APIs and non-APIs based on usage of specific ECP interfaces. To express the degree of dependence of an ETP on an interface one can apply econometric inequality indices [34], [35] or the Squale model [36]. Finally, we plan to replicate the work using ETPs from other repositories such as github.

Acknowledgment

We are thankful to dr. J.J.M. (Koo) Rijpkema for guiding us on the statistical analysis, and to ir. Bogdan Vasilescu for his comments on the preliminary versions of this paper.

References

[1] T. Tourwé, T. Mens, Automated support for framework-based software evolution, in: ICSM, 2003, pp. 148–157.

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

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

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

[5] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, T. N. Nguyen, A graph-based approach to API usage adaptation, in: OOPSLA, 2010, pp. 302–321.

[6] B. Dagenais, M. P. Robillard, Recommending adaptive changes for framework evolution, ACM Trans. Softw. Eng. Methodol. 20 (2011) 19:1–19:35.

[7] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, M. Kim, AURA: A hybrid approach to identify framework evolution, in: ICSE, 2010, pp. 325–334.

[8] R. C. Elandt-Johnson, N. L. Johnson, Survival models and data analysis, Wiley, 1999.

[9] I. Samoladas, L. Angelis, I. Stamelos, Survival analysis on the duration of open source projects, Inf. Softw. Technol. 52 (2010) 902–922.

[10] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, Y. Zou, Studying the impact of clones on software defects, in: WCRE, 2010, pp. 13–21.

[11] K. Crowston, H. Annabi, Information systems success in free and open source software development: Theory and measures, in: Software Process: Improvement and Practice, 2006, pp. 123–148.

[12] K. Crowston, H. Annabi, J. Howison, Defining open source software project success, in: ICIS, 2003, pp. 327–340.

[13] C. Subramaniam, R. Sen, M. L. Nelson, Determinants of open source software project success: A longitudinal study, Decis. Support Syst. 46 (2009) 576–585.

[14] C. M. Schweik, R. C. English, M. Kitsing, S. Haire, Brooks' versus Linus' law: an empirical test of open source projects, in: ICDGR, 2008, pp. 423–424.

[15] K. Beecher, A. Capiluppi, C. Boldyreff, Identifying exogenous drivers and evolutionary stages in FLOSS projects, J. Syst. Softw. 82 (2009) 739–750.

[16] M. M. Lehman, J. F. Ramil, Rules and tools for software evolution planning and management, Ann. Softw. Eng. 11 (2001) 15–44.

[17] J. Arthorne, M. Milinkovich, J. McAffer, Provisional API guidelines, http://wiki.eclipse.org/Provisional_API_Guidelines, consulted January, 2011.

[18] J. des Rivières, Evolving Java-based APIs, http://wiki.eclipse.org/Evolving_Java-based_APIs, consulted January, 2011.

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

[20] A. Schröter, T. Zimmermann, A. Zeller, Predicting component failures at design time, in: ISESE, ACM, 2006, pp. 18–27.

[21] J. Businge, A. Serebrenik, M. G. J. van den Brand, Eclipse API usage: the good and the bad, in: SQM, CEUR WS, 2012, pp. 55–63.

[22] A. Rainer, S. Gale, Evaluating the quality and quantity of data on open source software projects, in: ICOSS, 2005, pp. 11–15.

[23] M. Dmitriev, Language-specific make technology for the Java programming language, in: OOPSLA, 2002, pp. 373–385.

[24] D. Dig, R. Johnson, How do APIs evolve? A story of refactoring, J. Softw. Maint. Evol. 18 (2006) 83–107.

[25] A. Bolour, Notes on the Eclipse plug-in architecture, http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, consulted January, 2011.

[26] J. Davies, D. M. German, M. W. Godfrey, A. Hindle, Software bertillonage: finding the provenance of an entity, in: MSR'11, 2011, pp. 183–192.

[27] J. Arthorne, T. Eicher, M. Keller, D. Williams, Version numbering, http://wiki.eclipse.org/Version_Numbering, consulted October, 2011 (2009).

[28] J. Businge, A. Serebrenik, M. G. J. van den Brand, Compatibility prediction of Eclipse third-party plug-ins in new Eclipse releases, in: SCAM, IEEE, 2012, accepted.

[29] R Development Core Team, R: A Language and Environment for Statistical Computing, Vienna, Austria (2010).

[30] A. Agresti, Categorical Data Analysis, Wiley, 2002.

[31] 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.

[32] D. Weiss, Measuring success of open source projects using web search engines, in: ICOSS, 2005, pp. 93–99.

[33] C. M. Schweik, Identifying success and abandonment of FLOSS commons: A classification of Sourceforge.net projects, Upgrade: The Euro. J. Infor. Prof. 2.

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

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

[36] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, S. Ducasse, Software quality metrics aggregation in industry, Journal of Software: Evolution and Process.