2013 17th European Conference on Software Maintenance and Reengineering

Co-evolution of the Eclipse SDK Framework and its Third-party Plug-ins

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

***Abstract*—Today, when constructing a new software system, many developers build their systems on top of frameworks. Eclipse framework is one such popular and widely adopted framework that has been evolving for over a decade. Like many other evolving software systems, the Eclipse SDK framework has both stable and supported APIs (good interfaces) and unstable, discouraged and unsupported non-APIs (bad interfaces). However, despite being discouraged by Eclipse, in our experience, the usage of bad interfaces is not uncommon. In this thesis, by means of a series of empirical studies, we quantify/qualify some the challenges faced by Eclipse third-party plug-in developers in using the interfaces provided by the Eclipse SDK framework. Furthermore, we propose solutions to the identified challenges, like changes in development strategy to both interface providers and interface users. In particular, the lessons learned from this study can provide valuable information in particular to the interface providers, i.e., Eclipse SDK developers, and the interface uses, i.e., ETP developers, in co-evolving the Eclipse framework. In general, the lessons learned can be transferable to other framework ecosystem.**

***Keywords*-Eclipse; APIs; non-APIs; Plug-ins;**

I. Introduction

Today, many software developers build systems on top of frameworks (e.g., currently Eclipse marketplace[1] reports over 1,478 Eclipse solutions developed and three million Eclipse solutions directly installed from Eclipse). This approach has many advantages, such as reuse of the functionality provided [1] and increasing productivity [2]. However, these benefits are accompanied by co-evolutionary challenges that come along with using frameworks [3]: as the framework evolves, it makes changes to its interfaces and these changes may cause the applications that use them to fail [4], [5]. Framework-based applications are, therefore, subject to two independent and potentially conflicting evolution processes [6]: 1) evolution to keep up with the changes introduced in the framework (framework-based evolution); 2) evolution in response to the specific requirements of the stakeholders of the systems itself (general evolution).

Based on the two evolution processes of the framework-based applications and the respective challenges, we felt the need to study framework interface usage by the applications so as to quantify/qualify the challenges faced by the developers of the applications. Our case study is the Eclipse SDK framework and the Eclipse third-party plug-ins (ETPs) that use the interfaces from the framework. We chose to study the Eclipse SDK since it has been constantly evolving for over a decade and as we stated above it is also widely adopted. In particular, the lessons learned from this study can provide valuable information in particular to the interface providers, i.e., Eclipse SDK developers, and the interface uses, i.e., ETP developers, in co-evolving the Eclipse framework. In general, the lessons learned can be transferable to other framework ecosystem.

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

II. Research Questions

Eclipse SDK framework provides two types of interfaces it provides to the ETPs:

- ***Eclipse non-APIs ("bad"):*** The non-APIs, which we also term as *bad interfaces*, are internal implementation artifacts that are found in a package with the substring "internal" in the fully qualified package name according to Eclipse naming convention [7]. Eclipse discourages the use of these bad interfaces since they are unsupported, undocumented and are subject to arbitrary change or removal without notice [8].
- ***Eclipse APIs ("good"):*** The APIs, which we also term as *good interfaces*, are found in packages that do not contain the segment "internal" in the fully qualified package name [7]. The APIs are supported by Eclipse and are considered to be stable in new SDK releases [8].

Based on the information stated by Eclipse, we formulated a number of research questions:

RQ1 How does the evolution of ETPs with respect to the interfaces they use from the framework compare with the evolution of traditional applications?
RQ2 To what extent do ETPs depend on the two types of interfaces the Eclipse SDK framework provides?
RQ3 How does the compatibility of ETPs that depend solely on Eclipse APIs compare with that of ETPs that depend on at least one Eclipse non-API in new Eclipse SDK releases?
RQ4 Can prediction models based on the usage of the Eclipse interfaces in the ETPs help in predicting compatibility of an ETP in a new Eclipse SDK release?
RQ5 What is the state-of-practice of Eclipse interface usage by the developers of ETPs?

III. Methodology and Results

To answer the research questions stated above, we carried out a series of sequential empirical studies according to the order of the questions. The empirical study of RQ1 was based on source code analysis of 21 carefully selected open-source ETPs that we downloaded from SourceForge. The empirical studies of RQ2—RQ4 was based on source code analysis of 513 open-source ETPs all together having a total of 1,873 versions that we downloaded from SourceForge. The empirical study of RQ5 was based on a survey that we conducted on Eclipse ETP developers to investigate the state-of-practice of the usage of Eclipse interfaces.

A. RQ1: Evolution of ETPs

In answering RQ1, we employed Lehman's laws of software evolution [9] that describe the evolution of E-Type systems. An E-Type system can be defined as a real world system, i.e., components of real world processes [9]. We investigate whether the constrained evolution of the ETPs follows the laws, i.e., we study the evolution of the ETPs based on the interfaces they reuse from the Eclipse SDK framework. For each of the laws, we use metrics to study the evolutionary trends followed by the ETPs over time. The metrics used include total number of Eclipse interfaces used by an ETP, unique Eclipse interfaces used by the ETP, handles interfaces (added, deleted) in new versions of an ETP.

Our findings confirm the laws of *continuing change*, *self regulation* and *continuing growth* when metrics related to Eclipse SDK interfaces used by the ETPs are used to study the laws. Unlike this, the *conservation of familiarity* law was not confirmed and the results for the *declining quality* law were inconclusive. The *declining quality* was inconclusive since methodology we used to test the law could only be applied on eight of the 21 ETPs (not statistically significant). The findings of this research question confirm that the constrained evolution of ETPs is similar to general evolution of traditional software system reported [10], [11]. The detailed results of this study can be found in Businge et al. [12].

B. RQ2: Eclipse API Usage

In this research question, we study the Eclipse SDK interface usage by the ETPs. First, we grouped the downloaded 513 ETPs into five different classifications as follows:

I ETPs with all versions dependent solely on APIs—*good ETPs*;
II ETPs with all versions depending on a non-API—*bad ETPs*;
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*.

After classifying the ETPs, the number of ETPs in each classification were as follows: *good ETPs* had 286, *bad ETPs* had 181, *good–bad ETPs* had 32, *bad–good ETPs* had 6, and *oscillating ETPs* had 8.

Second, during the investigation of RQ1, we informally observed that ETPs that depended on at least one non-API are larger systems compared to those that depended on APIs only. In investigating RQ2, we decided to formally verify the observation on a larger scale of ETPs. Therefore, we considered 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 Eclipse SDK classes or interfaces. Furthermore, we want to understand whether *good ETPs* and *bad ETPs* differ in the amount of Eclipse SDK functionality used. We consider, therefore, two metrics related to the amount of Eclipse SDK functionality imported by an ETP: *D-Tot*, the number of `import` statements related to Eclipse SDK classes and interfaces, and *D-Uniq*, the number of unique `import` statements related to Eclipse SDK classes and interfaces.

We discovered that 44% of the 513 analyzed ETPs depend on "bad" interfaces and that developers continue to use "bad" interfaces. The empirical study also shows that ETPs 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 (we formally verified the observation using statistics). Furthermore, the findings show that ETPs use a diverse set of "bad" interfaces. In [13] we observed that the reason why developers use or start to use "bad" interfaces is because the required functionality is absent in the "good" interfaces. The detailed analysis and results of this research question can be found in Businge et al. [14].

C. RQ3: Survival of the ETPs

In this research question, we studied the survival of the ETPs in two of the largest classifications discussed, i.e., *good ETPs* and *bad ETPs*, discussed in RQ2. We define survival for a given version of an ETP developed on top of a given Eclipse SDK as the count of the new SDK releases the version of the ETP compiles with.

To determine if a version of an ETP is source compatible with a given release of an SDK, we include in the `BUILDPATH` of the ETP the `jar` files of the SDK in the Eclipse IDE. The ETP is compatible if there are no compile time errors in the Eclipse console. To determine source compatibility of the same version of the ETP with another SDK release, we swap the `jar` files of the former with the latter.

Comparing the two biggest categories of ETPs, i.e., *good ETPs* and *bad ETPs*, we observed that the *good ETPs* have a very high source compatibility success rate compared to the *bad ETPs*. However, we have also observed that recently released *bad ETPs* also have a very high forward source compatibility success rate. This high source compatibility success rate is due to the dependency structure of these ETPs: recently released *bad ETPs* predominantly depend on old Eclipse non-APIs rather than on newly introduced ones. Compatibility results of the ETPs may be may be affected by other confounding factors that we did not test for example, the domain of the ETPs. Finally, we showed that the majority of plug-ins hosted on SourceForge do not evolve beyond the first year of release. Detailed analysis and the results for this research question can be found in Businge et al. [15].

D. RQ4: Compatibility Prediction of ETPs

The investigation of this research question builds upon the findings in RQ3. In RQ3 we discovered that the *good ETPs* are always compatible in new SDK releases and the *bad ETPs* have a high source compatibility failure rate in new SDK releases. We also found that the *bad ETPs* that use more of the old non-APIs and less of the newly introduced non-APIs have a very high source compatibility success rate in new SDK releases.

Based on the compatibility findings from RQ3, in this research question we built and tested prediction models. The models predict the source compatibility of a given version of an ETP developed on top of a given SDK release in a new SDK release. The prediction models were built using statistics, specifically, using the *binary logistic regression model*. The models are built based on the interfaces they use from Eclipse SDK. Since we are aware that APIs used by the ETPs do not cause incompatibility failures in new SDK releases, the models are built based on the non-API usage. For training the model to predict compatibility of ETPs in a given SDK release we perform the following steps: 1) For a group of ETPs built on top of a given SDK release, we extract the number of non-APIs from each of the ETPs and the corresponding age of these non-APIs, i.e., at what stage during the evolution of the Eclipse SDK they were introduced. 2) We obtain source compatibility results of the ETPs with the new SDK release with which we want to build the model, i.e., the results are binary where 0 means incompatibility and 1 means compatibility. 3) For each ETP, the dependent variable is the compatibility result and the independent variables are the number of non-APIs used by the ETP taking into account the age of the non-APIs. 4) We subject the dependent variable and the independent variables to the *binary logistic regression model* in equation 1.

$$P(Comp) = \frac{1}{1+e^{-Z}} \tag{1}$$

where Z is the linear combination of predictor variables and $P(Comp)$ is the probability of compatibility.

$$Z = \beta_0 + \beta_1 X_1 + \beta_p X_p + ... + \beta_p X_p \tag{2}$$

where Xs are the predictor variables and p is the number of predictor variables.

The results for building and testing a predictive model that, given the compatibility of an ETP in a certain releases, predicts the compatibility with a subsequent SDK release, are as follows: Our findings produced 23 statistically significant prediction models having high values of the strength of the relationship between the predictors and the prediction (logistic regression R^2 of up to 0.810). In addition, the results from model testing indicate high values of up to 100% *precision* and *recall* and up to 98% *accuracy* of the predictions. Detailed analysis and the results for this research question can be found in Businge et al. [16].

E. RQ5: Putting the Developer in the Loop

To answer RQ5, we carried out a survey on Eclipse SDK interface usage by the ETP developers. These are the survey goals:

1) Identify if application developers are aware of the guidelines and the different types of interfaces Eclipse provides, and the characteristics of these developers.
2) Understanding the differences in characteristics of software applications that use the two types of Eclipse SDK interfaces (APIs and non-APIs).

To this end, we asked questions related to the following categories: Education and experience of the developers, general Eclipse product development, awareness of the Eclipse interfaces and their guidelines, use of non-APIs, and testing of Eclipse products.

The survey obtained 30 completed responses from the ETP developers that we analysed using statistics. The the questionnaire and the data used in the analysis can be found here[2].

Our findings for research question RQ5 are as follows: First, we observed that the experience of an ETP developer plays a major role in the development and maintenance of the ETPs. We have observed that developers with a level of education of up to master degree have a tendency not to read product manuals/guidelines. Second, while for less experienced developers instability of the non-APIs overshadows their benefits, more experienced developers prefer to enjoy the benefits of non-APIs despite the instability problem. Finally, we have observed that there are no significant differences between Open Source and commercial Eclipse products in terms of awareness of Eclipse guidelines and interfaces, Eclipse product size and updating of Eclipse products in the new SDK releases. Detailed analysis and the results for this research question can be found in Businge et al. [13].

IV. Conclusions and Future Work

In this thesis, we carried out a series of empirical analyses to investigate how the evolution of the Eclipse SDK

[2]http://www.win.tue.nl/∼jbusinge/CSMR13

framework affects the applications that use the framework interfaces. The Eclipse SDK framework has two types of interfaces it provides to applications developers: the stable and supported APIs (good interfaces) and unstable, discouraged and unsupported non-APIs (bad interfaces).

Our findings are as follows:

1) We observed that, the evolution of framework-based applications, with respect to the framework interfaces they use, conforms to Lehmans' laws of software Evolution [9]. This finding confirms that evolution of framework-based application is similar to the evolution of traditional software systems.
2) We discovered that 44% of the 513 analyzed Eclipse third-party plug-ins depend on "bad" interfaces and that developers continue to use "bad" interfaces.
3) We also discovered that as Eclipse states, good interfaces are indeed stable as they do not cause compatibility failures to applications that solely depend on them. We also observed that indeed non-APIs cause compatibility failures to applications that use them in new framework releases. Furthermore, we observed that old non-APIs are relatively stable.
4) We have proposed a prediction model to predict the compatibility of a framework-based applications developed on a given Eclipse SDK in a new SDK release.
5) We also observed that, while for less experienced developers instability of the non-APIs overshadows their benefits, more experienced developers prefer to enjoy the benefits of non-APIs despite the instability problem.

The work in this thesis can be extended in a number of ways: First, in this thesis we studied stability of the Eclipse SDK interfaces based on how they affect the applications that use them in new SDK releases. In a follow up study, we hope to study the stability of the interfaces based on the changes on the interface signatures as Eclipse SDK evolves. Second, we hope to come up with a tool based on the prediction models we built that can be used by framework-based application developers or users. Users can use the tool to predict compatibility of the application they are using in case they want to get updates of a newer SDK release. Developers can use the tool to give them pointers in their source code where very unstable interfaces are used. Furthermore, the work presented in this thesis could be extended in other framework ecosystem. One would first study naming convention the interface during the evolution of these frameworks. For example, in NetBeans [17] the naming convention of the interfaces include: `friend` that is used for features accessible to specific components in the framework, `devel` (under development) is a name for a contract that is expected to become a stable API, `stable` interfaces are those that have received a final state and the maintainers are ready to support it forever and never change them incompatibly, and `official` are stable ones and also packaged into one of NetBeans official namespaces.

REFERENCES

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

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

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

[4] B. Dagenais, M. P. Robillard, Recommending adaptive changes for framework evolution, J. Tran. on Soft. Eng and Meth. 20 (2011) 19:1–19:35.

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

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

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

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

[9] M. M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68 (9) (1980) 1060–1076.

[10] E. J. Barry, C. F. Kemerer, S. Slaughter:, How software process automation affects software evolution: a longitudinal empirical, J. Softw. Maint. and Evol.: Res. and Pract. 19 (1) (2007) 1–31.

[11] G. Xie, J. Chen, I. Neamtiu, Towards a better understanding of software evolution: An empirical study on open source software, in: Proc. 25th IEEE Int'l Conf. on Soft. Maint. (ICSM2009), 2007, pp. 51–60.

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

[13] J. Businge, A. Serebrenik, M. G. J. van den Brand, Analyzing the Eclipse API usage: Putting the developer in the loop, in: CSMR, 2013, accepted.

[14] J. Businge, A. Serebrenik, M. G. J. van den Brand, Eclipse API usage: the good and the bad, in: SQM, 2012, pp. 54–62.

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

[16] J. Businge, A. Serebrenik, M. G. J. van den Brand, Compatibility prediction of Eclipse third-party plug-ins in new Eclipse releases, in: SCAM, 2012, pp. 164–173.

[17] J. Tulach, API stability, http://wiki.netbeans.org/API_Stability, consulted on June 19, 2012 (January 7 2012).