2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation

Compatibility Prediction of Eclipse Third-Party Plug-ins in New Eclipse Releases

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*—Incompatibility between applications developed on top of frameworks with new versions of the frameworks is a big nightmare to both developers and users of the applications. Understanding the factors that cause incompatibilities is a step to solving them. One such direction is to analyze and identify parts of the reusable code of the framework that are prone to change. In this study we carried out an empirical investigation on 11 Eclipse SDK releases (1.0 to 3.7) and 288 Eclipse third-party plug-ins (ETPs) with two main goals: First, to determine the relationship between the age of Eclipse non-APIs (internal implementations) used by an ETP and the compatibility of the ETP. We found that third-party plug-in that use only old non-APIs have a high chance of compatibility success in new SDK releases compared to those that use at least one newly introduced non-API. Second, our goal was to build and test a predictive model for the compatibility of an ETP, supported in a given SDK release in a newer SDK release. 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% of precision and recall and up to 98% of accuracy of the predictions. Finally, despite the fact that SDK releases with *API breaking changes*, i.e., 1.0, 2.0 and 3.0, have got nothing to do with non-APIs, our findings reveal that non-APIs introduced in these releases have a significant impact on the compatibility of the ETPs that use them.**

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

I. Introduction

Applications developed on top of frameworks are becoming increasingly popular these days and users of these applications are constantly on the rise [1], [2] (e.g., currently Eclipse marketplace[1] reports over 1,300 Eclipse solutions developed and there are over 1.6m installations of these solutions installed directly from Eclipse). Frameworks are constantly evolving to provide better quality of the functionality they provide to their clients. The applications that want to benefit from the better quality provided by the new release of the framework are subject to compatibility problems [3], [4]. Understanding the factors that cause incompatibilities is a step to solving them. One direction is to identify parts of the reusable code of the framework used by the applications that are prone to change [5].

In [6], [7] the authors state that Eclipse non-APIs (internal implementations) are subject to arbitrary change without notice. The authors discourage the use of these non-APIs and further state that developers can use them at their own risk. However, despite being discouraged, we have observed that the use of non-APIs is not uncommon: 44.2% of the Eclipse third-party plug-ins (ETPs) on SourceForge have at least one version that depends on at least one non-API [8]. Moreover, we found that the use of non-APIs affects the compatibility of the ETPs with new Eclipse SDK releases [9].

Extending the aforementioned work, in this paper we target two main goals. First, we consider the relationship between the *age* of non-APIs (i.e., the Eclipse SDK release in which the non-API was introduced) used by an ETP and the *compatibility* of the ETP in new SDK releases. Based on our findings a framework-based application developer can estimate the maturity of the non-APIs she is using. Second, we build a predictive model for the compatibility of an ETP supported in a given SDK release in forthcoming SDK releases. The predictive model can be used by Eclipse-based application developers to assess the chances of their applications' compatibility in a newer SDK release. Furthermore, it can be used by Eclipse-based application users before they update the SDK release where their application is installed to a newer SDK release. To perform the investigation, we considered 288 ETPs and extracted from the source code of 11 major SDK releases (1.0 to 3.7) metrics related to age of the non-APIs used by these ETPs.

The remainder of the paper is organized as follows: In Section II we discuss preliminary concepts related to Eclipse and our previous work on ETPs. In Section III we discuss compatibility of the ETPs. In Section IV we discuss the relationship between age of non-APIs used by the ETPs and their compatibility. In Section V present model building and validation. In Section VI we discuss threats to validity. Section VII we discuss related work. Finally in Section VIII we discuss conclusions and future work.

II. Preliminaries

A. Eclipse Plug-in Architecture

Eclipse SDK is an extensible platform that provides core of services for controlling a set of tools working together to support programming tasks. The plug-ins in the SDK provide core functionality upon which all plug-in extensions are built. Applications contribute to the SDK by extending its interfaces for specific solutions.

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

978-0-7695-4783-1/12 $26.00 © 2012 IEEE
DOI 10.1109/SCAM.2012.10

In addition to Java access (visibility) levels, Eclipse has another level called internal implementations (*non-APIs*). According to Eclipse naming convention [7], the non-APIs are artifacts found in a package with the segment `internal` in a fully qualified package name. These artifacts may include public Java classes, interfaces, public or protected methods, or fields in such a class or interface. As opposed to non-APIs, APIs are found in packages that do not contain the segment `internal`. For a detailed discussion of the API usage in Eclipse see [8].

In the current study we are interested in analyzing the non-APIs (classes and interfaces) used by the ETPs.

B. Data Collection

ETPs were released on SourceForge between January 1, 2003 to December 31, 2010. Data collection took place on February 16, 2011. We categorized the ETPs according to their first release (first version) dates in the different years (2003 to 2010) on SourceForge. The ETPs were further categorized based on presence of non-APIs dependencies, i.e., ETPs that had all the versions depending solely on APIs (ETP-APIs) and those that have at least one of the versions depending on a non-API (ETP-non-APIs). For further details on data collection see [8].

C. Compatibility of the ETPs

While Eclipse has multiple notions of compatibility we focus on *API source compatibility*: i.e., the source code of the ETP should continue to compile without errors against the revised SDK without changes in the ETP sources.

In [9], for each of the ETP-APIs and ETP-non-APIs, we determined an ETPs' API source compatibility with the different SDK releases (1.0 to 3.7) by compiling the ETP with each of the SDK releases. The `jar` files of the SDK release we want to test for source compatibility with the ETP are included in the `BUILDPATH` of the ETP. Compatibility results into zero compile time errors in the Eclipse console. To determine compatibility of the same ETP with another SDK release, we swapped the `jar` files in the `BUILDPATH` of the ETP of the former SDK release with the latter.

Furthermore, we found that ETP-APIs were always compatible in later SDK releases that do not involve *API breaking changes* [10] and most of the ETP-non-APIs were incompatible. For this reason, in the current study we decided to further our investigation only on compatibility of ETP-non-APIs in new SDK releases.

To study compatibility of ETPs we have to choose the baseline, i.e., the "intended SDK version". However, this version is rarely indicated in the ETP source code. Luckily, the results in [9] revealed that the majority of the ETPs are source compatible with the SDK released in the same year as the ETP. Therefore, in the current study, we have chosen those SDK releases, to be the baselines of the ETPs' source compatibility. A few of the ETPs that were incompatible with the SDK release in the same release year were found to be compatible with the SDK released in the previous year or a year before the previous. For these ETPs we chose the latest SDK release they compile with as their baseline.

			Compatibility of ETPs in new Eclipse SDK releases							
	# ETPs		**3.0**	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**	**3.6**	**3.7**
2.1	29	S	18	9	8	8	8	8	8	8
		F	11	20	21	21	21	21	21	21
3.0	48	S		28	22	22	21	20	19	19
		F		20	26	26	27	28	29	29
3.1	34	S			15	14	14	14	14	14
		F			19	20	20	20	20	20
3.2	40	S				24	19	17	16	16
		F				16	21	23	24	24
3.3	38	S					25	22	21	20
		F					13	16	17	18
3.4	36	S						31	30	28
		F						5	6	8
3.5	33	S							24	23
		F							9	10
3.6	30	S								28
		F								2

Table I: Compatibility of the ETPs developed on top of a given Eclipse release in new Eclipse release(s). F–Compatibility failure and S–Compatibility success.

III. Compatibility of ETP-non-API

Table I shows the results of the compatibility experiments for the current study. The first column indicates the compatibility with respect to SDK release baselines. The ETPs in the column # ETPs are all source compatible with the SDK releases in first column. Column 3.0-3.7 show the number of ETPs that successfully compile (S) and fail to compile (F) with the corresponding SDK releases.

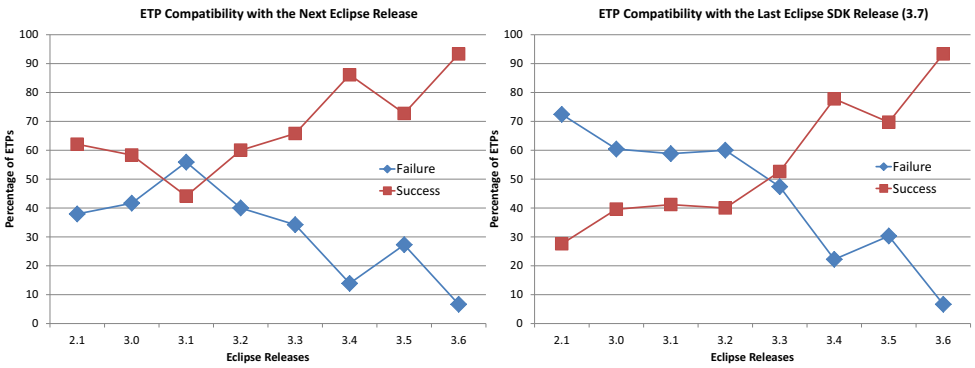
Figure 1 shows the results of compatibility failure and success trends of the ETP supported in the different SDK releases (2.1-3.6) with the next SDK release (left plot) and the most recent SDK release (right plot). The plots are normalized results of corresponding entries in Table I (F and S are a percentage of the number on compatible ETPs–column *# ETPs*). Both figures show a general increase in the compatibility success of the ETPs.

IV. Age of non-APIs vs Compatibility

In this section, we investigate the relationship between the age of the non-APIs used by an ETP and the compatibility of the ETP in a new SDK release. We analyze the compatibility of the ETP by looking at the ETPs' compatibility success or failure in the new SDK release. For age of the non-APIs used by the ETP, we search the SDK release in which the non-API was introduced along the evolution of the SDK.

A. Hypotheses

In [9], we have inspected a small number of ETPs and observed that increase in compatibility success can be attributed to newly released ETPs using "old" non-APIs. This informal observation has lead to formulating the following null and alternative hypotheses.



(a) Next Eclipse release (diagonal Table I) (b) Last Eclipse release (colum 3.7 Table I)

Figure 1: Compatibility trends for ETPs developed on top of the SDK releases (x-axis) with the next SDK release (left) and the most recent SDK release–3.7 (right).

					# non-APIs from Eclipse SDK Releases								
		Compatibility	**# ETPs**	**Aggregation**	**1.0**	**2.0**	**2.1**	**3.0**	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**
ETPs Supported in Eclipse Releases	3.2	Success	16	Median	2	0	0	0	0	0			
				Mean	2.2	0.0	0.1	0.3	0.1	0			
				Max	8	3	1	2	1	0			
				Min	1	0	0	0	0	0			
		Failure	24	Median	2.5	1	0	2	0	0			
				Mean	5.8	4.5	0.5	2.3	1.4	0.5			
				Max	32	20	2	8	11	4			
				Min	0	0	0	0	0	0			
	3.5	Success	23	Median	1	0	0	0	0	0	0	0	0
				Mean	2	2.3	0.1	1.5	0.9	0.1	0.1	0.1	0
				Max	10	15	1	8	6	1	2	2	0
				Min	0	0	0	0	0	0	0	0	0
		Failure	10	Median	1.5	0	0	0	0	0	0	0	0
				Mean	2.3	1.8	0.3	0.9	0.5	0.3	0.3	0.1	0
				Max	16	17	4	12	17	6	5	3	0
				Min	0	0	0	0	0	0	0	0	0

Table II: The descriptive statistics of non-APIs in ETPs supported in Eclipse SDK 3.2 and 3.5. The groups (Failure and Success) are with respect to compatibility of the ETPs with Eclipse 3.7.

- $H1_0$: *Older non-APIs used in an ETP are equally likely to cause compatibility problems of the ETP with new SDK releases as the newer non-APIs.*
- $H1_a$: *The older a non-API used in an ETP, the less likely it will cause compatibility problems of the ETP with new SDK releases.*

B. Fact Extraction

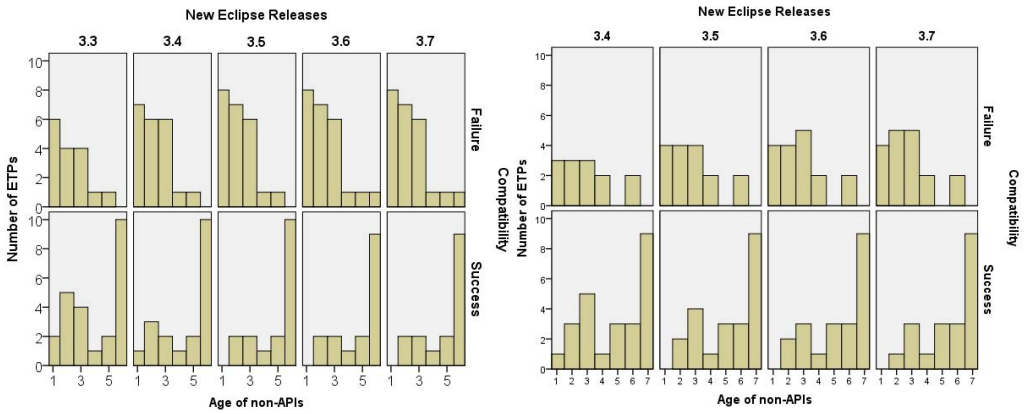
We carried out the following steps in extracting the facts used in our analysis using a number of scripts:

1) We extracted public non-APIs (classes and interfaces) from each of the SDK releases (1.0 to 3.6) using the Abstract Syntax Tree (AST) of Eclipse JDT. We further classified these non-APIs based on the SDK releases that introduce them.
2) For each of the ETPs supported in the different SDK releases, we extracted non-APIs used by the ETP.
3) For each non-API in the non-API collection of each ETP, we searched where in the non-API collections of the SDK (Step 1 above) is it located.

Table II shows the descriptive statistics of the non-APIs in the ETPs supported in two SDK releases, 3.2 and 3.5. Descriptive statistics for other releases is available at [11, p.2].

C. Data Transformation

To perform the formal statistical analysis of the relationship between the age of the non-APIs used in an ETP and the compatibility of the ETP, we chose to use the ordinal scale enabling us to perform measure of association [12]. We transformed the data into a contingency table that is required for an ordinal measure of association between the variables. To generate the contingency table we started by ranking the age of the non-APIs used by the ETP supported in a given SDK release such that *rank 1* is associated with non-APIs introduced in the SDK release where the ETP is supported, *rank 2* with non-APIs introduced in the SDK release preceding the SDK release where the ETP is supported, etc. Next we determine the overall age of the non-APIs used by the ETP as the age of the youngest used non-API. The rationale behind the overall age ranking of the



(a) ETPs supported in SDK 3.2. x-axis, 1=3.2, 2=3.1, etc. (b) ETPs supported in SDK 3.3. x-axis, 1=3.3, 2=3.2, etc.

Figure 2: Histograms showing the source compatibility trends of ETPs supported in SDK 3.2 (left) and SDK 3.3 (right).

	non-API age ranking					
	3.2(1)	3.1(2)	3.0(3)	2.1(4)	2.0(5)	1.0(6)
Failure	6	4	4	1	1	0
Success	2	5	4	1	2	10

Table III: Contingency table for *overall age rank* of the non-APIs of the ETPs supported in SDK 3.2 with respect to the compatibility with SDK 3.3.

	New SDK releases							
	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
2.1	**.247**	.433	.371	.371	.371	.371	.371	.371
3.0		.389	.293	.293	.345	.398	.450	.450
3.1			.547	.647	.647	.647	.647	.647
3.2				.538	.683	.800	.725	.725
3.3					.476	.665	.693	.759
3.4						.269	.376	.336
3.5							.544	.602

Table IV: Kendall's τ_c correlation coefficients for the age of the non-APIs used by the ETPs and the compatibility of the ETPs over new SDK releases. The results of ETPs supported in SDK 3.6 are not included since they exhibit over-fitting as a result of imbalance in the number of failing (2) and successful (28) ETPs.

non-APIs used by the ETP is based on the stated hypothesis, i.e., the youngest used non-API are most likely to cause compatibility problems of the ETP with new SDK releases. Finally, for each of the new SDK releases the results are grouped according to the compatibility success/failure with respect to the different new SDK releases.

D. Results

Table III shows the contingency table for the distribution of the ages of the non-APIs in the ETPs supported in SDK 3.2 with respect to the compatibility with SDK 3.3. The the cells indicate the number of ETPs that succeeded or failed to compile with SDK 3.3 depending on the latest/youngest introduced non-APIs the ETPs use. For example 6 in the cell (Failure, 3.2(1)) indicates that there are six ETPs that failed with SDK 3.3 and all six contained at least one non-API introduced in SDK 3.2 as the youngest non-API. Similar contingency tables can be built for other SDK releases.

Figure 2 shows the plots of the contingency tables of the ages of the non-APIs for the ETPs supported in SDK 3.2 and 3.5 with respect to the compatibility with newer SDK releases. The rest of the plots of the other ETPs can be found in [11, pp.3–10]. For example, the sub-plot–block 3.3 of the left plot in Figure 2 represents Table III.

Table IV presents the results of Kendall's τ_c rank correlation coefficients for the contingency tables akin to Table III. All the correlation coefficients are statistically significant ($p < 0.05$) except one for ETPs originating from SDK 2.1 and compatibility with SDK 3.0 typeset in bold ($p = 0.198$). When interpreting the strength of the correlation, one should keep in mind that $\tau_c \geq 0.9$ is very rare unless in an undesirable situation of a more or less perfect separation in the groups being compared [13]. The perfect separation would mean in our case that the ETPs that succeed depend only on old non-APIs and the ETPs that fail depend on at least one young non-API.

E. Discussion

From the histograms in Figure 2 we can observe that the plots of the ETPs supported in SDK 3.2 and 3.3 that failed in new SDK releases (upper plots), are positively skewed. This indicates that the majority of the ETPs that failed in the new releases use at least one young non-API and very few of the ETPs that fail in new releases use only old non-APIs. The lower plots of Figure 2 are negatively skewed indicating that majority of the ETPs that successfully compiled with new SDK releases mostly use very old non-APIs and very few that succeed use a young non-API. The rest of the plots [11, pp.3–10] show similar trends except for the histogram for

ETPs supported in SDK 2.1. The reason for this histogram to have a different shape is that the non-APIs used by the ETP are all relatively young.

The values Kendall's τ_c in Table IV are all greater than zero and are all but one significant at a level of 0.05. This implies that there is an evidence of the tendency of ETPs that use only old non-APIs to be compatible in new SDK releases (positive relationship of *age* vs *compatibility*).

The relationship of *age* vs *compatibility* between the ETPs supported in SDK 3.1, 3.2, 3.3 and 3.5 and the corresponding new SDK releases is relatively strong. As opposed to this, for the ETPs in SDK 2.1 and 3.0 the relationship is relatively weak, which can be explained by observing that all non-APIs were relatively young at that time. Similarly, for SDK 3.4 the relationship is weak due to presence of many outliers.

On two occasions we furthermore observe a decrease of the τ_c values for ETPs originating from the same SDK from one new SDK release to the next one. This is, for instance, the case for ETPs originating from SDK 3.2: τ_c drops from 0.800 to 0.725. This decrease can be attributed to introduction of outliers further discussed in Section V.

The results also indicate that some non-APIs that were introduced in early SDK releases continue to be used in later SDK releases without causing incompatibility problems. This could possibly mean that these non-APIs have reached maturity but they have not been graduated into APIs.

Summarizing, we reject $H1_0$ and accept $H1_a$ that "a newly introduced non-APIs used by an ETP is likely to cause incompatibilities of the ETP with new SDK releases".

V. Compatibility predictions

In this section we will build and validate prediction models for estimating the compatibility of an ETP, supported in a given SDK release, in new SDK releases. To perform a compatibility prediction of an ETP in a new SDK release ($release_j$) we have to know in which SDK release the ETP is supported ($release_i$, $i < j$) and for which SDK release the prediction is required. As dependent variable we take compilation success or failure of the ETP with respect to $release_j$, as independent variables—the number of non-APIs used in the ETP originating from the SDK $release_k$, $k \leq i$. For instance, Table II shows values of the dependent and independent variables for $release_i = 3.2$ and $release_j = 3.7$ as well as for $release_i = 3.2$ and $release_j = 3.7$. Since we base our predictions on ETPs supported in SDK releases 2.1, 3.0, ..., 3.6 and predict compilation success for releases 3.0, 3.1, ..., 3.7 we need to construct 36 prediction models.

To build and validate the models, we need a data-set for training the models and a separate data-set for testing the built models. Given $release_i$ and $release_j$ as above, as the training set we consider all ETPs in the cell of $(release_i, release_j)$ in Table I, e.g., to predict compatibility of ETPs supported in 3.1 with respect to 3.4 we consider 34 ETPs, 14 of which are compatible with 3.4 (denoted "S" in Table I). The training set consists of all ETPs supported with respect to SDKs released prior to $release_i$ and compatible with it, e.g., to test our model for predicting compatibility of ETPs supported in 3.1 with respect to 3.4 we consider 37 ETPs, 9 that have been originally supported in 2.1 and are compatible with 3.1, and 28 that have been originally supported in 3.0 and are compatible with 3.1.

A. Hypotheses

In this section we state the hypotheses that will assess how good the predictors are.

- $H2_0$: *There is no statistical relation between the number of non-APIs (taking into account the age of these non-APIs), used by an ETP and the ETPs' compatibility in newer SDK releases.*
- $H2_a$: *There is a statistical relation between the number of non-APIs (taking into account the age of these non-APIs), used by an ETP and the ETPs' compatibility in newer SDK releases.*

B. Model training

In this section we will discuss how the trained the models that were built.

1) Methodology: The multivariate logistic regression model (based on equation 1 and 2) below was used to train the models. The data used to train some of the prediction models is presented in the descriptive statistics in Table II, i.e., the prediction models of the compatibility of the ETPs with the last SDK release (3.7). For a given the ETPs supported in a given SDK release, the predictor variables are number and ages of the non-APIs and the predicted variable is the compatibility (failure or success). For example, using the sample of the data we used in Table II, in building the model to predict compatibility of the ETPs supported in SDK 3.2 with SDK 3.7 (upper part of the table), the predictor variables are the number of non-APIs used by the ETP introduced in the SDK releases 1.0, 2.0,...,3.2 and the predicted variable compatibility which is a binary, i.e., 0 for failure and 1 for success. The logistic function used in building the prediction models is 1 [13]:

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

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.

To perform the analysis, we used the software IBM SPSS Statistics 19. We used the *backward-stepwise elimination* method (Backward: LR) to build the models. Backward-stepwise elimination starts with all the variables in the model, then at each step, variables are evaluated for entry

		3.5		3.6		3.7	
		β	Sig	β	Sig	β	Sig
3.4	1.0	**-0.462**	**.064**	**-0.235**	**.106**	-0.119	.336
	2.0	-0.840	.708	-0.109	.459	-0.086	.490
	2.1	-1.240	.382	0.137	.886	0.343	.649
	3.0	**-0.788**	**.065**	**-0.575**	**.035**	**-0.603**	**.008**
	3.1	0.903	.225	0.126	.729	-0.131	.717
	3.2	–	–	-0.009	.983	0.115	.782
	3.3	–	–	–	–	–	–
	3.4	–	–	–	–	–	–
	const	**6.770**	**.023**	**4.091**	**.001**	**2.480**	**.000**

Table V: Prediction variables for ETPs supported in Eclipse 3.4 in new Eclipse releases. Variable 3.2, 3.3 and 3.4 was were omitted in the corresponding models since they had very high S.E (indicates multi-collinearity between the predictors)

		3.6		3.7	
		β	Sig	β	Sig
3.5	1.0	-0.139	.393	**-0.927**	**.067**
	2.0	-0.141	.243	**-0.528**	**.077**
	2.1	**-2.042**	**.068**	0.422	.904
	3.0	-0.050	.930	**0.867**	**.147**
	3.1	0.056	.883	-0.564	.373
	3.2	**-3.018**	**.028**	**-3.583**	**.088**
	3.3	1.274	.370	1.701	.554
	3.4	-0.973	.316	-1.309	.381
	3.5	–	–	–	–
	const	**3.293**	**.005**	**6.025**	**.020**

Table VI: Prediction variables for ETPs supported in Eclipse 3.5 in in new Eclipse releases. None of the non-APIs present in the ETPs was introduced in Eclipse 3.5.

		Predicted					
		New SDK releases					
		3.5		3.6		3.7	
Observed		S	F	S	F	S	F
3.4	S	30	1	29	1	26	2
	F	2	3	3	3	5	3
3.5	S			24	0	23	0
	F			4	5	1	9

Table VII: Classification results form model *training*. S–Compatibility, F–Incompatibility

	3.5			3.6			3.7		
	A	P	R	A	P	R	A	P	R
3.4	92	97	94	89	97	91	81	93	84
3.5				88	100	86	97	100	96

Table VIII: Error analysis for model *training*. A–Accuracy, P–Precision and R–Recall

and removal. For all the models, we use the default value for the p-value for entry and removal of the variables of 0.05 and 0.1, respectively, was used. *Forward-stepwise selection* was also tried and the results were more or less the same as the *backward-stepwise elimination* method and the *Enter* method a little bit worse. Using equation (1) in the models, the predicted value (observed prediction), are in the range of 0 to 1. For all the models, we applied the default threshold of 0.5 on the predicted value, i.e., the model considers an ETP to be incompatible if the predicted value is less than 0.5, and compatible, otherwise.

2) Results: Table V and VI present the results of the linear combinations (equation (2)) of the predictor variables for the ETPs supported in SDK 3.4 and 3.5 with respect to the ETPs' compatibility in the corresponding newer SDK releases. Models for other releases can be found in [11, pp.12–13]. After the stepwise elimination analysis, the predictor variables in bold remained in the model, i.e., having significant impact on the outcome of the prediction. The variables not in bold are not considered to have a significant impact on the outcome of the prediction. Table IX presents the results of *Nagelkerke* R^2 for all the corresponding models. *Nagelkerke* R^2 indicates the strength of the relationship between the predictors and the prediction in the model.

For example, in Table VI, the variables 2.1, 3.2 and $const$ remained in the model as the most significant predictors on the outcome of the prediction in from SDK 3.5 to 3.6. Therefore, prediction model between ETPs supported in SDK and SDK 3.6 is shown in equation (3) below. To make a prediction for an ETP supported in SDK 3.5 for its compatibility in SDK 3.6 one should substitute only the number of non-APIs used by the ETP introduced in SDK 2.1 and 3.2. The corresponding value of *Nagelkerke* R^2 = 0.691, indicating strength of the relationship between the predictors and the prediction in the model from SDK 3.5 to 3.6 is shown in cell (3.5, 3.6) in Table IX. Table V shows the corresponding results of prediction for ETP supported in SDK 3.4 in newer SDK releases.

$$P(Comp) = \frac{1}{1 + e^{-3.293 + 2.042 * SDK_{2.1} + 3.018 * SDK_{3.2}}} \quad (3)$$

The classification and error analysis training for the models we trained for the ETPs supported in SDK 3.4 and 3.5 are presented in Table VII and Table VIII, respectively. Table VII shows the number of correctly and incorrectly classified cases. Table VIII shows the results for *accuracy*, *precision* and *recall* [14] corresponding to the results presented in Table VII. *Accuracy* measures the portion of all decisions that were correct decisions, i.e., computed from the numbers in Table VII, the result of ((S,S) + (F,F))/((S,S)+(S,F)+(F,F)+(F,S)). *Precision* measures the portion of the assigned categories that were correct, i.e., the result of (S,S)/((S,S)+(S,F)). *Recall* measures the portion of the correct categories that were assigned, i.e., the result of (S,S)/((S,S)+(F,S)). Over all the models considered the accuracy values are normally distributed (Shapiro-Wilk's $p \simeq 0.14$) with the mean of 83.4% and standard deviation 6.7%. For precision and recall we do not observe a normal distribution (Shapiro-Wilk's $p < 0.05$). Precision ranges between 63% and 100% with the median of 84%, recall—between 62% and 96% with the median of 80%. Detailed results for the classification and error analysis for all the trained models can be found in [11, p.13].

		Prediction for							
		3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
ETPs supported in	2.1	.462	.471	.386	.386	.386	.386	.386	.386
	3.0		.362	.356	.356	.377	.456	.477	.477
	3.1			.449	**.810**	**.810**	**.810**	**.810**	**.810**
	3.2				**.517**	**.582**	**.571**	**.512**	**.512**
	3.3					**.545**	**.657**	**.629**	**.769**
	3.4						**.703**	**.503**	.383
	3.5							**.691**	**.803**

Table IX: Nagelkerke R^2 values showing the strength of the relationship between the predictors and the prediction for all the models that we generated. Values exceeding 0.5 are typeset in boldface.

3) Discussion: A number of observations can be drawn from the results. First, we observe high values of Nagelkerke's coefficient of determination R^2 in Table IX, indicating that the proportion of variation explained by the predictors. Recall that the values of R^2 for the logistic regression model are typically much smaller than those observed for linear regression model [13]. Models with $R^2 = 0.371$ [15], $R^2 = 0.382$ and $R^2 = 0.409$ [16] and even $R^2 = 0.175$ [17] have been shown to make useful predictions. Similarly to Gyimothy et al. [17] and English et al. [15], we also used the univariate logistic regression but it performed worse than the multivariate regression.

For testing goodness of fit in our models, we performed the Hosmer-Lemeshow test [18, p.147]. This test checks how closely the observed and predicted probabilities match: rejecting the null-hypothesis would mean that there is a significant lack of agreement between the fitted logistic regression model and the observed data, and the model should not be used for statistical inference. All the Hosmer-Lemeshow statistics for our models were not significant at a threshold of 0.05, in fact, they were always higher than 0.25. Non-significance indicates that the model prediction does not significantly differ from the observed.

We also cross checked the correlation between the predictors variables. A few of the variables were highly correlated that caused over-fitting in the models. These variables also had a very high standard error when included in the over-fitted model. We excluded these variables in the models as can be seen in the models we presented in Section V-B2.

C. Model outliers

Like any other empirical study, our data had outliers. As outliers we have considered values with the absolute studentized residuals exceeding 2 [19]. In all the models, the number of outliers ranged between zero and two. The highest number of outliers of four was observed in the model between between SDK 3.2 and 3.3. We observed three types of outliers: 1) ETPs that depend on very only old non-APIs and failed to compile in new releases. The outliers of ETPs with very old non-APIs are of two types: those that graduate from non-API to APIs and those that change their interface and still stay non-APIs. Failure caused by only graduation of non-API was found in only one of the ETPs we analyzed. This indicates that graduation of non-APIs is very limited. Preliminary investigation on the graduation of non-APIs a long the evolution of SDK showed similar results. In a follow-up study we intend to investigate graduation on non-APIs in the evolution of Eclipse. 2) ETPs that had dependency on newly introduced non-APIs but still compiled with new SDK releases. One possible reason for this scenario could be that the non-API is actually old but was just renamed or moved. Our methodology identifies renamed and moved classes as newly introduced. Alternatively, the newly introduced non-API can turn out to be stable.

	Predictor Variables								
	1.0	2.0	2.1	3.0	3.1	3.2	3.3	3.4	3.5
Significant	25	24	2	20	3	4	0	0	0
Total	35	35	35	27	20	14	9	5	2
Percentage	71	69	6	74	15	29	0	0	0

Table X: The frequency of the predictor variables in the models. Significant–number of times a predictor variable was considered significant, Total–number of times a predictor was used to build the model.

D. Predictor frequency

In this section we present and discuss the most frequently selected significant predictors in the built models.

Table X presents the a summary results of frequency of the most significant predictors selected. Detailed results can be found in [11, p.11]. With the exception of predictor 3.5, all other predictors have non-APIs. We observe that the most frequent significant predictors are 1.0, 2.0 and 3.0. SDK releases 1.0, 2.0, and 3.0 are releases considered to have *API breaking changes* [10]. Despite the fact that the *API breaking changes* have got nothing to do with non-APIs, our study reveals that non-APIs from SDK releases 1.0, 2.0 and 3.0 have a significant impact on the compatibility of the ETPs.

E. Model validation

In this section we will test a sample of the models we built in Section V-B2. Below we describe the methodology we used to collect the data for testing the models.

1) Methodology: For a given model under test, we collected cases of data that was used to build models for ETPs supported in earlier SDK releases. For example, testing the models between ETPs supported in SDK 3.1 with new SDK releases, we use cases of ETPs supported in SDK 3.0 and SDK 2.1 that compiled with SDK 3.1. The rationale is that if an ETP compiles an SDK release, then it is supported in that SDK release. Since we also know the compatibility results of these ETPs with newer SDK releases after 3.1, we consider these compatibility results as the *expected results*.

For computing the *observed results*, the collected cases are substituted into the model between SDK 3.1 and the SDK we

Observed		Predicted					
		New SDK releases					
		3.5		3.6		3.7	
		S	F	S	F	S	F
3.4	S	82	0	79	0	79	0
	F	5	0	6	2	6	2
3.5	S			102	6	96	9
	F			4	0	4	3

Table XI: Classification results form model *testing*. S–Compatibility, F–Incompatibility.

	3.5			3.6			3.7		
	A	P	R	A	P	R	A	P	R
3.4	94	100	94	93	100	93	93	100	93
3.5				91	94	96	88	91	96

Table XII: Error analysis for model *testing*. A–Accuracy, P–Precision, R–Recall.

are trying to predict (equation 1). Since the cases imported from ETPs developed for earlier SDKs have no entries (non-APIs) in later releases, we put zeros in those missing entries. For example, the cases imported from SDK 3.0 to 3.1 have no non-APIs introduced in SDK 3.1 as the ETPs existed earlier than SDK 3.1. We put zeros in the entry 3.1 which also implies that each of the non-APIs in the imported cases will be one year older in SDK 3.1 compared to SDK 3.0.

2) Results: Table XI and XII present a sample of classification results and error analysis, respectively, from model validation for the ETPs supported in SDK 3.4 and 3.5. Across all models tested if only significant predictors were considered accuracy newer falls below 84%, precision—below 83%, recall—below 90%. If significance of the predictors is ignored accuracy newer falls below 76%, precision—below 74%, recall—below 90%. Detailed results can be found in [11, p.14].

3) Model Discussion: Like in model training, the results results of model testing reveal high values of accuracy, precision and recall. Furthermore, out of curiosity, we validated the models by including all predictors, ignoring the respective significance of the predictors. We observed that there is no big difference when all the predictor variables are included in the model. The results from model testing for the comparison when only significant predictors and when all the predictors are included in the models can be found in [11, p.14].

From the classification results, we can observe that we have very few data cases in the incompatibility group compared to those in the compatibility group. This due to the fact that most of the ETPs that compiled with the next SDK release after the SDK in which the ETPs are supported also compiled with the rest of the new SDK releases until SDK 3.7. Furthermore, we also observe that the percentage of incorrectly predicted incompatible ETPs is high in some cases. This is due to the fact that some of these incompatible ETPs were observed as outliers during the model training phase. In a follow-up study we plan to collect a new dataset of ETPs to replicate the model testing exercise.

F. Overall discussion

Recall that all possible combinations of the SDK release in which ETP is supported and the SDK release the prediction is made for, should lead to 36 models. However, not all these models are different: if non-APIs that the ETPs supported in $release_i$ did not change between $release_j$ and $release_{j+1}$, then the same ETPs that are compatible with respect to $release_j$ are also compatible with respect to $release_{j+1}$. Hence, predictive models for $release_i$ and $release_j$, on the one hand, and $release_i$ and $release_{j+1}$, on the other hand, are identical. Moreover, since number of ETPs supported in 3.6 and compatible with 3.7 was very high and the number of ETPs supported in 3.6 and compatible with 3.7 was very low, no meaningful logistic prediction model has been constructed.

Out of the 23 models currently we have validated 13. The remaining models will be validated in a follow-up study. From the discussion of the results in Section V-B and V-E, we can reject $H2_0$ and accept $H2_a$ that "the number of non-APIs (taking into account the age of the non-APIs), used by an ETP supported in a given SDK release, when used as predictors, have a significant impact on the outcome of the ETPs' compatibility in newer SDK releases".

Besides the interesting statistics, we observe that our results generate a different model for most of the predictions. Differences between the models can be attributed to interplay of two different factors. First of all, evolution of Eclipse SDK results in differences between releases, so *a priori* there is no reason to expect that the same model can predict compatibility of an ETP with respect to them. For situations where the all the non-APIs used by the ETPs did not change between releases, we observe that we have the same models, e.g., from SDK 3.6 to 3.7 in for ETPs supported in SDK 3.2 [11]. Second, non-APIs originating from the same SDK release are not necessarily the same in all subsequent SDK releases, since different subsequent SDK releases might use different versions of the non-API.

VI. Threats to Validity

As any other empirical study, our findings may subject to validity threats. We categorize the possible threats into construct, internal and external validity.

Construct validity focuses on how accurately the metrics utilized measure the phenomena of interest. The methodology used to measure the age of the non-APIs used by the ETPs, is subject to construct validity. Because we wanted to use a quick way of determining the age of non-APIs, the applied methodology considers renamed or moved non-APIs as newly introduced in the SDK releases. In a follow-up study we want to use code cloning to determine the moved and renamed non-APIs [20].

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 in all the results we have presented.

External validity is the validity of generalizations based on this study. Much as we only consider Open Source ETPs from SourceForge, we could say that our results are generalizable to all the Eclipse solutions since we do not measure ETP specific artifacts. The artifacts we measure are based on the framework. However, our results may not be generalizable beyond the Eclipse-based solutions. 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 in [8], [9], [21]. In [21] 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 Eclipse interfaces in the new releases of the ETPs. In [8] we investigated the Eclipse API usage by 512 ETPs, and proposed the distinction between ETP-APIs and ETP-non-APIs. In [9], we investigated the survival of ETPs in SDK releases by comparing two groups of ETPs: those that had all versions depending solely on APIs (ETP-APIs) and those that have at least one of the versions depending on a non-API (ETP-non-APIs). The current study is one of the follow-up studies proposed in [9].

Class change proneness: Several studies have applied metrics related to internal factors that impact change proneness in a software system. Penta et al. in [5] and [22] investigated the relationship between design pattern roles and class change proneness. Penta et al. in [5] carried out a finer-grain level of design pattern roles. This study reports some classes playing certain design pattern roles (e.g., Adapter, Receiver and Command) are more change prone compared to the classes that do not play these roles. Romano and Pinzger in [23] investigates the relationship between Chidamber and Kemerer metrics and change proneness. The authors reports that cohesion exhibits the strongest correlation with the number of code changes. In comparison to our study, we investigate the impact *age* of non-APIs (which is an external factor), used by the ETPs on change proneness of the non-APIs.

Predictive models: A number of studies have reported predictive models. Most of these models are based on predicting defects in a software system. Mende and Koshke in [24] presented two strategies, probabilistic classifier and regression algorithms, to incorporate the treatment of effort into defect prediction models. A number of studies have built defect prediction models based on C&K metrics for example [15], [16], [17]. In comparison to our study, we report compatibility prediction models of framework-based applications in new versions of the framework.

VIII. Conclusions and Future Work

In this paper we analyzed source code to investigated two main goals: 1) the relationship between the age of non-APIs used by an ETP and the compatibility of the ETP in new Eclipse SDK releases. 2) to build and validate a compatibility predictive model for an Eclipse third-party plug-in (ETP), supported in a given SDK release, in a new SDK release. For first goal, we have observed that newly introduced non-APIs used by an ETP are likely to cause incompatibilities of the ETP with new SDK releases. For the second goal, we have built 23 and currently we have validated 13 of the 23 compatibility predictive models. We have shown that the built models can generate good predictions with high accuracy, precision and recall. Furthermore, despite the fact that SDK releases with *API breaking changes*, i.e., 1.0, 2.0 and 3.0, have got nothing to do with non-APIs, our findings reveal that non-APIs introduced in these releases have a significant impact on the compatibility of the ETPs that use them with new SDK releases.

The investigation of the stability of the non-APIs is based on an external artifact, i.e., the impact the non-APIs has on software that use it. In a follow-up study we plan to carry out an investigation by measuring the internal artifacts of the non-API, e.g., relationship age of the non-API and changes on the signature of the non-API.

In testing the models, based on the methodology we used to collect testing samples, we had very few cases of ETPs that were incompatible. In some models, the percentage of the incorrectly predicted ETPs was high for the few incompatible cases considered. In a follow-up study we will collect more samples to test the models. We will also investigate the possibility of combining predictors to make better predictions (e.g., using a non-linear combination of predictor variables or transformation of the predictor variables prior to model fitting). We will also investigate on how we can combine models to come up with one or few general prediction models (e.g. using voting majority [17]). The models presented can be used to make interpolation compatibility predictions, i.e., predictions can only be made SDK releases that were used to train the models. In a follow-up study we plan to build extrapolation prediction models, i.e., making predictions on an SDK we have not used in building the model. We also intend to develop a domain specific tool based on the models that can be used to make predictions with the help of a tool like RASCAL [25]. One might also consider more refined prediction models based on usage of specific SDK packages as predictor variables. To express the degree of dependence of an ETP on a specific SDK package one can apply econometric inequality indices [26], [27] or the Squale model [28].

Eclipse ETPs are examples of framework-based software. Similarly to the distinction between API and non-API in Eclipse other frameworks publish guidelines pertaining to stability of their interfaces: e.g., NetBeans distinguishes between eight categories of interfaces, including unstable "private" and "friend", more stable "devel" and even more stable "stable" and "official" [29]. Hence, studies similar in spirit to our work can be carried out on further software frameworks.

REFERENCES

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

[2] D. Brugali, G. Broten, A. Cisternino, D. Colombo, J. Fritsch, B. Gerkey, G. Kraetzschmar, R. Vaughan, H. Utz, Trends in robotic software frameworks, in: D. Brugali (Ed.), Software Engineering for Experimental Robotics, Vol. 30 of Tracts in Advanced Robotics, Springer, 2007, pp. 259–266.

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

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

[5] M. Di Penta, L. Cerulo, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the relationships between design pattern roles and class change proneness, in: ICSM, 2008, pp. 217 –226.

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

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

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

[10] J. Arthorne, T. Eicher, M. Keller, D. Williams, Version numbering, http://wiki.eclipse.org/Version_Numbering, consulted on October 05, 2011 (2009).

[11] J. Businge, A. Serebrenik, M. G. J. van den Brand, Complementary material for "Compatibility prediction of Eclipse plug-ins over new Eclipse releases", http://www.win.tue.nl/ ~aserebre/SCAM12Appendix.pdf (2012).

[12] A. Agresti, Categorical Data Analysis, John Wiley & Sons, Inc., 2002.

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

[14] N. C. Barford, Experimental measurements : precision, error and truth, Chichester : Wiley, 1985.

[15] M. English, C. Exton, I. Rigon, B. Cleary, Fault detection and prediction in an open-source software project, in: PROMISE, 2009, pp. 17:1–17:11.

[16] Y. Zhou, H. Leung, Empirical analysis of object-oriented design metrics for predicting high and low severity faults, IEEE Trans. Softw. Eng. 32 (10) (2006) 771–789.

[17] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Trans. Softw. Eng. 31 (10) (2005) 897–910.

[18] D. W. Hosmer, S. Lemeshow, Applied Logistic Regression, Wiley, 2000.

[19] D. Belsley, E. Kuh, R. Welsch, Regression Diagnostics: Identifying Influential Data And Sources Of Collinearity, Wiley, 2004.

[20] T. R. Dean, M. Di Penta, K. Kontogiannis, A. Walenstein, Clone detector use questions: A list of desirable empirical studies, in: Duplication, Redundancy, and Similarity in Software, 2007, pp. 1–5, Dagstuhl Seminar Proceedings 06301.

[21] 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.

[22] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, R. T. Alexander, Design patterns and change proneness: An examination of five evolving systems, in: Software Metrics Symposium, 2003, pp. 40–49.

[23] D. Romano, M. Pinzger, Using source code metrics to predict change-prone Java interfaces, in: ICSM, 2011, pp. 303–312.

[24] T. Mende, R. Koschke, Effort-aware defect prediction models, in: CSMR, 2010, pp. 107–116.

[25] P. Klint, T. van der Storm, J. Vinju, Rascal: A domain specific language for source code analysis and manipulation, in: SCAM, 2009, pp. 168–177.

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

[27] 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.

[28] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, S. Ducasse, Software quality metrics aggregation in industry, Journal of Software: Evolution and Process-doi:10.1002/smr.1558.

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