The Evolution of Eclipse

Tom Mens[1], Juan Fernández-Ramil[2,1] and Sylvain Degrandsart[1]
[1]Institut d'Informatique, Université de Mons-Hainaut
Avenue du Champ de Mars 6, 7000 Mons, Belgium
{ tom.mens | j.f.ramil}@umh.ac.be

[2]Computing Department, The Open University
Walton Hall, Milton Keynes, MK7 6AA, U.K.
j.f.ramil@open.ac.uk

Abstract

We present a metrics-based study of the evolution of Eclipse*, an open source integrated development environment, based on data from seven major releases, from releases 1.0 to 3.3. We investigated whether three of the laws of software evolution were supported by the data. We found that Eclipse displayed continual change and growth, hence supporting laws 1 and 6. Six size indicators, out of eight, closely followed trend models. Four were linear and two superlinear. We found evidence of increasing complexity (law 2) in only two indicators, out of five. At subproject level, size and complexity are not distributed uniformly, and subproject size can be modelled as a negative exponential function of the rank position. We encountered a range of different size and complexity trends across subprojects. Our approach and results can help in evaluating the future evolution of Eclipse, the evolution of other systems and in performing comparisons.*

1. Introduction

Real-world software systems, either proprietary or open source software (OSS), require fixes and enhancements, that is, *evolution*, since their first release and as long as they are used. The so-called *laws of software evolution* were proposed by Lehman [7] to provide a better understanding of this phenomenon over a software system's lifetime, which often spans over several years [10]. The laws originated from studies of proprietary software in the 70s [7]. They have been a topic of research ever since. The laws offer a starting point for a theory of software evolution [8], a basis for justification of software maintenance good practice [9] and have been cited in textbooks (e.g., [13]).

Many improvements in software processes and technology have occurred since the 70s, when the laws of software evolution were first proposed. Such advances include object-orientation, iterative and evolutionary processes and OSS. For example, a study of Linux, a popular OSS, by Godfrey and Tu [5] published eight years ago found superlinear growth, in apparent contradiction to some of the laws. It is not sufficiently well known to what extent contemporary OSS evolution follows the laws [4] and further empirical research is needed. In this paper, we present the results of our measurement-based study of the laws of software evolution for the popular *Eclipse*[1] open source project. Eclipse is an interesting case because it is a large software system of approximately 2 million lines of code (LOC) at release 3.3 and with a huge user community. At the moment of performing this study, data on its evolution history were available for approximately a six-year time period. The focus of this research was to examine data in order to determine whether Eclipse supports three of the laws of software evolution. The results offer a basis to study future evolutions of Eclipse, and to compare its evolution to other OSS.

This article is structured as follows: In Section 2 we briefly present three laws of software evolution for which we study the empirical evidence in Eclipse. Section 3 provides details about the Eclipse system and the data that we have extracted. Section 4 discusses the tools and measurements we used. Section 5 presents the analysis of the results at the global level, considering the Eclipse as a single evolutionary entity. Section 6 covers the analysis of empirical evidence at the level of "namespaces" (subprojects). Section 7 discusses the threats to validity. We compare our results with some earlier studies of other OSS in section 8. Section 9 presents topics of future research in this area.

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

2. The laws of software evolution

Eclipse seems to match well E-type software's [7] type. An E-type system solves a problem or addresses an application in a real-world domain. The laws were proposed as a description of the evolution of this type of software. In this paper, we use measurements to characterise the evolution of Eclipse and explore the empirical support for three of the eight laws. Barry *et al.* [1] classified the laws into three broad groups. Laws 1, 2, 6 and 7 are seen as related to the evolution characteristics of the software. Laws 4 and 5 are linked to organisational and economic constraints. Laws 3 and 8 are seen as *meta-laws*. Due to the limited effort available for data extraction and analysis, we initially focused on the first subset of laws (1, 2, 6 and 7), the ones related to characteristics of the evolving software. Later in our study, we excluded law 7 because of challenges in data collection, such as the need for an appropriate measurement of the evolving quality of the system as perceived by the users. A recent statement of the three laws that we studied is given below [9]:

Law 1: Continuing Change. *An E-type system must be continually adapted else it becomes progressively less satisfactory in use.*
Law 2: Increasing Complexity. *As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it.*
Law 6: Continuing Growth. *The functional capability of E-type systems must be continually incrased to maintain user satisfaction over the system lifetime.*

As in other OSS, Eclipse can be studied using various data such as the source code of each release, the *bytecode* for each supported platform for each release, the defect database Bugzilla[2], version repositories, and mailing lists. In our study, due to effort and time constraints, we focused on the Eclipse source code and the bytecode for Windows. We also considered data from the Bugzilla defect database. Our research question is the following: *Having Eclipse's code repository (source and bytecode) and its Bugzilla reports as data sources, what is the empirical support for laws 1, 2 and 6, by considering Eclipse as a single entity and by looking at its major components?*

3. About Eclipse

Eclipse is an open source, extensible, integrated development environment (IDE) and also an application framework (i.e., it can be used as a basis for other software systems). Eclipse is written in the popular object-oriented programming language Java. In addition, a small part of the Eclipse implementation requires specific code for each platform (e.g., Windows, Mac OS X, Linux) to improve the interoperability of Eclipse and its performance.

We focused on the Eclipse SDK (i.e., Software Development Kit), which includes the Eclipse Platform, Java Development Tools (JDT), and the Plug-in Development Environment (PDE). Based on the data obtained from the Eclipse project website[3] there are about 60 active "committers". The majority of these contributors seem to be from IBM, the company that initially created the system.

At the time of conducting this research, the release history data is available over a period close to six years, from release 1.0 in November 2001 to release 3.3.1 in September 2007. There are several types of releases available[4]. In this study we only studied the major releases of Eclipse. They are indicated in boldface in Table 1, together with their release date and the size difference of the downloadable `.zip` file (in megabytes) with respect to the previous release. We excluded the minor releases from our study since their small size increments suggest only a marginal contribution to the overall Eclipse functionality. A similar conclusion was reached when computing the incremental growth in terms of number of `.java` files, number of `.jar` files, number of compiled classes and LOC.

Table 1. Eclipse public releases

release	type of release	date	Δ size
3.3.1	minor	Fri, 21 Sep 2007	0.2
3.3	**major**	Mon, 25 Jun 2007	**19.0**
3.2.2	minor	Mon, 12 Feb 2007	1.0
3.2.1	minor	Thu, 21 Sep 2006	0.3
3.2	**major**	Thu, 29 Jun 2006	**17.1**
3.1.2	minor	Wed, 18 Jan 2006	0.0
3.1.1	minor	Thu, 29 Sep 2005	0.4
3.1	**major**	Mon, 27 Jun 2005	**17.6**
3.0.2	minor	Fri, 11 Mar 2005	0.1
3.0.1	minor	Thu, 16 Sep 2004	0.3
3.0	**major**	Fri, 25 Jun 2004	**22.5**
2.1.3	minor	Wed, 10 Mar 2004	0.0
2.1.2	minor	Mon, 3 Nov 2003	0.0
2.1.1	minor	Fri, 27 Jun 2003	0.9
2.1	**major**	Thu, 27 Mar 2003	**7.4**
2.0.2	minor	Thu, 7 Nov 2002	0.4
2.0.1	minor	Thu, 29 Aug 2002	0.0
2.0	**major**	Thu, 27 Jun 2002	**17.4**
1.0	**major**	Wed, 7 Nov 2001	**36.6**

The Eclipse SDK represents nearly 2 million (more precisely, 1,988,767) LOC at the most recent release (3.3). At

[2] `https://bugs.eclipse.org/bugs/`

[3] `http://www.eclipse.org/projects/project_summary.php`, consulted on 26 May 2008 to obtain an up-to-date list of active committers to the Eclipse Platform, PDE and JDT, respectively.

[4] `http://download.eclipse.org/eclipse/downloads/build_types.html`

release 1.0, Eclipse counted only half a million LOC (to be precise, 506,252), so the size of Eclipse in LOC has increased four times over the considered period.

Each release can be downloaded as a single `.zip` file[5] (e.g., `eclipse-SDK-2.1.2-win32.zip`) containing code and both user and programmer documentation. For our study we concentrate on the Java source code (in `.java` files) and the compiled code (in `.class` files that are bundled in `.jar` files). For studying the compiled versions of the Eclipse SDK, we used the ones for the Windows 98/ME/2000/XP platform. While the code is an important part of the `.zip` file, many other files are included for configuration, data storage and documentation. To get an idea, for release 3.0 about 63% of the files were Java code (10,635 `.java` files out of a total of 16,816 files).

4. Measurements and tools

To quantify changes (law 1), we compared the code at consecutive pairs of releases. We used our own Python scripts to count files and file sizes, and to compute differences (additions, deletions, modifications) between releases using various Unix commands (`diff`, `find`, `grep`, `sed`). We considered five complementary "types" of complexity (law 2) and looked at six of its indicators. These included measurements of *code quality* provided by STAN, a commercial static code analysis tool [11], and defect data from Eclipse's defect reports stored in the Bugzilla bug tracking system. Size (law 6) was assessed through eight size measurements at different levels of abstraction and granularity. Measurements of LOC were extracted by STAN. We derived the other size measurements manually by inspecting the Eclipse downloadable files. We used Microsoft Excel for plotting and trend analysis.

STAN performs structural analysis on compiled Java code, taking a set of `.jar` files as input. STAN computes program dependencies, calculates various types of measurements (e.g., size, cyclomatic complexity, coupling), and produces visualisations of dependency graphs at various levels of abstraction. The tool classifies all the measurements into three categories: green (acceptable), yellow (referred to as "warnings") and red (referred to as "errors"), comparing them to a set of thresholds. These thresholds might be fine-tuned to specific domains or applications. In our study we used the default threshold values. For cyclomatic complexity, these threshold values are '> 15' for yellow and '> 20' for red. One of the complexity indicators that we considered, called *quality issues* by us, was calculated by adding the number of entities with yellow and red measurements.

[5] `http://archive.eclipse.org/eclipse/downloads`

5. Results - global view

This section presents the results of the analysis of data extracted from the Eclipse SDK when looking at Eclipse globally, that is, as a single evolutionary entity. Unless stated otherwise, from now on in this paper, wherever we mention Eclipse, we refer to the Eclipse SDK.

5.1. Growth - global

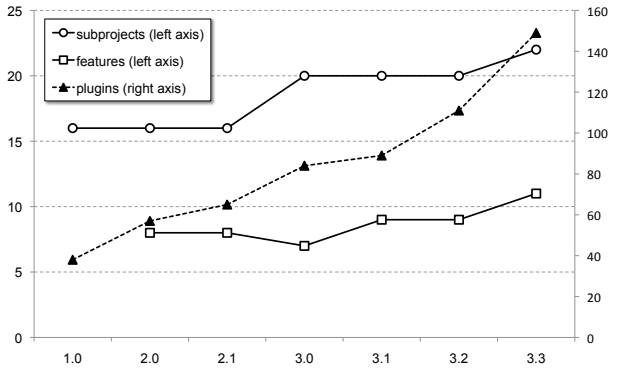

Figure 1. Size of Eclipse measured in number of subprojects, features and plug-ins.

Since Eclipse follows a well-defined *plug-in* architecture [3, 12], we started our study by looking at the growth of Eclipse in terms of architectural entities. In particular, we considered three different entities: *plug-ins* (the components, which are the basic unit of functionality in Eclipse [12], *features* (a grouping mechanism for plug-ins), and *subprojects*. Subprojects are "namespaces" that store code based on a 'prefix' naming convention, e.g., the `jdt` subproject is built up from everything recursively contained in `org.eclipse.jdt`.

Fig. 1 shows the growth trends of these architectural units over releases. We observe that the number of plug-ins has increased from 38 to 149 plug-ins (292%). The *features* and *subprojects* have increased more slowly than plug-ins, in relative terms (only by 38%). Because the notion of feature was introduced in Eclipse at release 2.0, its trend starts at that release and not at 1.0.

At a finer granularity (bytes), Fig. 2 shows the growth trend of Eclipse measured by the size of the `.zip` files that contain the compiled (byte-code) and source code as released. Fig. 1 displayed the release numbers on the x-axis, whilst Fig. 2 shows the release dates in format `mm/dd/yy`. Since the major releases of Eclipse have been made available at more or less equally spaced time intervals, both release sequence numbers and real time dates show roughly the same information. Therefore, the remaining plots in this

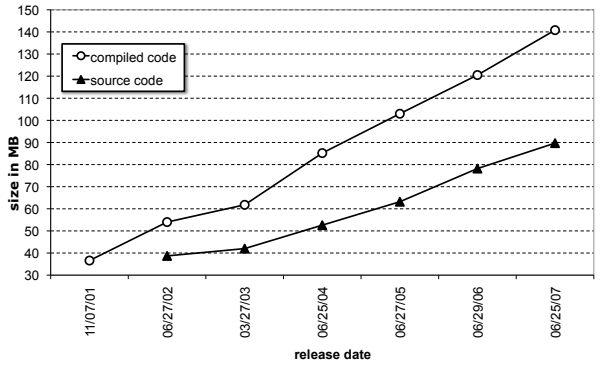

Figure 2. Size of downloadable `.zip` files (in megabytes) as a function of release dates.

paper use the release sequence on the x-axis. It is worth noting that the regularity in the timing of the major releases suggests that the Eclipse development team followed a well-planned release process.

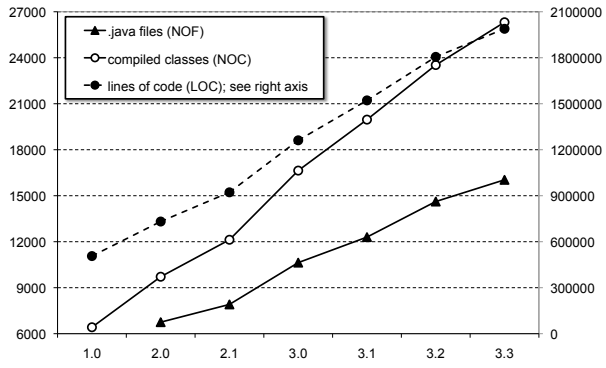

Figure 3. Growth in number of files (NOF), of classes (NOC) and lines of code (LOC).

Fig. 3 shows similar growth trends to those of Fig. 2, this time measured as the number of `.java` files (NOF), number of compiled classes (NOC) and LOC. The growth in estimated LOC was computed by the STAN tool. The average growth has been 260 KLOC per each major release. The relative growth in LOC between releases 1.0 (506 KLOC) and 3.3 (1,988 KLOC) is about 293% and very close to the relative growth in number of plug-ins (292%).

Visual inspection of Figures 1 to 3 suggests that continuing growth is a dominant characteristic. In order to confirm this, we calculated the incremental growth between releases and checked whether it was positive, zero or negative. A large portion of positive increments would support the hypothesis that there is increasing growth. In total, out of 45 increments, one was negative (in features, from releases 2.1 to 3.0), six were zero (4 in subprojects and 2 in features) and the rest (38 out of 45, or 84%) were positive, providing support that Eclipse's evolution has followed law 6 of "continuing growth".

A further question that we explored is what type of trend predominates. In order to do this, we examined the growth trends of the eight size measures presented in figures 1 to 3. Using Excel's 'trendline' function, we fitted its five different models (2nd order polynomial, linear, exponential, logarithmic and power models) to each measurement. Second order polynomials have the property that the sign (positive or negative) of the second order term can indicate whether a trend is superlinear or sublinear [6]. This can give us insights about the evolving complexity as is explained in section 5.3. We used the coefficient of determination R^2 as the goodness-of-fit criterion. We fitted the models using the release sequence numbers on the x-axis. When observing the trend line or curve superimposed on the actual data, for R^2 values lower than 0.9 the trendline did not appear to be a good fit. For this reason, we only recognised a trend model as a good fit when R^2 was equal or greater than 0.9. For two competing models that were a good fit, if the difference in R^2 between a linear model and the best model was less than 0.01, we report the trend as linear. This makes sense because, generally, if the R^2 difference is so low, the departure from linearity is marginal. Based on this, a trend model was identified in six out of eight indicators. For subprojects and features, the R^2 values were lower than 0.9 and, hence, no trend is reported. Within the six identified trends, four of them (NOF, NOC, LOC and size of the compiled code in MBytes) were linear, with R^2 ranging from 0.992 to 0.997. For the other two (plug-ins and downloadable source code in MBytes) the best fit were quadratic polynomials (superlinear), with R^2 values of 0.9712 and 0.9944, respectively. For these two indicators, exponential models provided slightly lower, but similarly good fits (R^2 values of 0.9706 and 0.9895, respectively). Our trend analysis suggests that Eclipse's evolution not only conforms to law 6 ("continuing growth"), but also that, according to six out of eight growth indicators, the trends were sufficiently disciplined as to follow closely recognisable trend models.

5.2. Change - global

Fig. 4 presents the number of changes (i.e., additions, modifications and deletions) in `.java` files between five pairs of releases (2.0-2.1, 2.1-3.0, 3.0-3.1 and 3.1-3.2). It is difficult to apply `diff` when the same filename occurs more than twice in a pair of releases. Moreover, we could not use the relative path as part of the filename because files can be moved across folders between releases. Therefore, for measurement of the number of modified files, we excluded all files with filenames that appeared more than once

in the same release. The percentage of excluded files varies between 18% and 30% of all files, so we make an underestimation with maximum error of 30%. Fig. 4 shows the number of files with modifications to code only, as well as those files where the comments are also modified. Files with modifications to code only ('code modified files' in Fig. 4) represent between 75% and 90% of all modified files. In the figure we observe that release 3.0 represents the largest increment (number of additions) so far in the history of Eclipse. This was accompanied by the largest number of deletions, suggesting that 3.0 was also a restructuring release. Subsequently, the number of modified files reaches its maximum in release 3.1, reflecting fixes and other rework necessary after release 3.0. Values of modifications after release 3.1 decreased again.

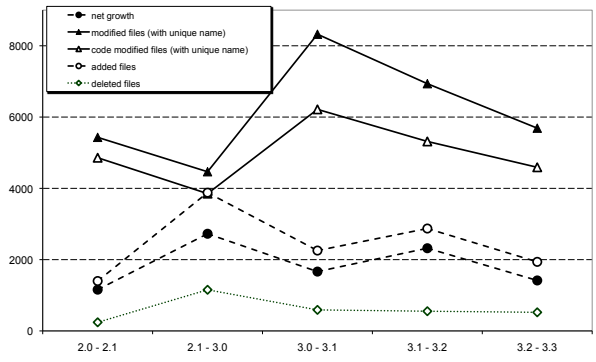

Figure 4. Added, changed and deleted files.

As a summary, we conclude that Eclipse also seems to follow law 1 of "continuing change". The highest number of added files occurred at release 3.0 (see Fig. 4). The number of modified files is always higher than the number of added files, with a peak of modified files at release 3.1.

5.3. Complexity - global

Complexity has possibly many dimensions and it is unlikely to find a single generally accepted definition for each of them. In this study, we explored six different ways to assess complexity which correspond to five different "types" of complexity. Some of the presented types may be overlapping and other types may be defined. The considered types seemed compatible with the data that we were able to extract. Due to lack of space, the statement of each type below is short and we cannot explain their assumptions. There is no meaning attached to the numerical order in the listing:

1. Complexity as size: given any program, a larger program is likely to be more difficult to understand or modify (assuming everything else constant). This type of complexity can be described as related to size through a monotonic function (e.g., $Complexity = a \times Size$, where a is a positive number). Within this view, the observation of increasing size (e.g., in LOC) will support the hypothesis of increasing complexity of type 1.

2. Complexity as the inverse of productivity: as the software gets more complex, the implementation of any given enhancement is likely to be more difficult. If effort is constant, growth rate will decrease. One hypothetical example of this would be a software system that follows the relationship $\Delta Size = b/\Delta Complexity$ where b is a positive number, related to the level of effort. Under this view, under constant effort, a sublinearly increasing or stagnating ($\Delta Size = 0$) size will support the hypothesis of increasing complexity of type 2.

3. Complexity as the number of possible interconnections: increasing complexity of this type is likely to lead to an increasing impact of any additions on existing code. Such impact can be measured by the ratio between number of additions and modifications. If type 3 complexity is increasing, this ratio will increase, that is, adding every new entity will trigger the change of an increasing number of existing entities [7].

4. Complexity as likelihood to introduce defects: increasing complexity of this type will manifest itself as an increasing number of defects introduced during its evolution. The observation of an increasing number of defects found, if other factors such as programming and testing effort remain constant, will suggest increasing complexity of type 4.

5. Complexity as code quality: an increasing number of quality issues (or their *density*) identified by static program analysis will suggest increasing complexity of type 5. Quality issues are identified when a set of complexity-related measurements, including cyclomatic complexity, exceed some threshold.

As seen in section 5.1, overall, the size of Eclipse has increased over the six years considered. Hence, we can conclude that Eclipse's type 1 complexity has been increasing. With regards to complexity of type 2, either a sublinearly growing size or stagnated size will suggest that it is increasing. As also seen in section 5.1, six out of eight measurements display either linear or superlinear growth with the other two not displaying a consistent trend. Therefore, there is no sufficient evidence that complexity of type 2 has been increasing in Eclipse.

In order to assess complexity of type 3, one possible measurement is the *portion of system handled* (PSH) [7].

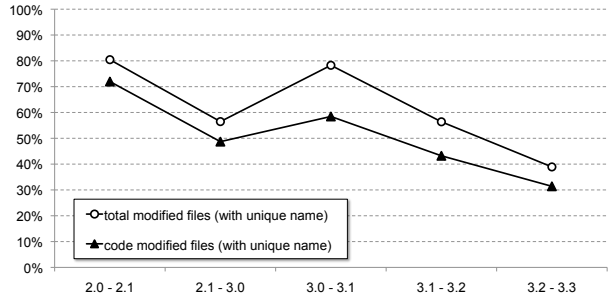

Figure 5. Portion of system handled (PSH).

This metric represents the ratio between the number of modified files and the total number of files (size). Fig. 5 presents this data for Eclipse. PSH decreases its value in 3 out of the 4 release pairs for which modifications were measured. Its overall trend is predominantly decreasing and for this reason PSH provides no consistent support for increasing complexity of type 3.

To explore the evidence for increasing complexity of type 4, we extracted the number of defects reported by the Bugzilla defect repository for each of the Eclipse releases. The overall trend is given in Fig. 6. Out of seven releases, only 2 (releases 2.0 and 3.0) show an increase in the number of defects with respect to the previous release. There isn't a consistent increasing trend in the number of defects and, hence, no indication of increasing complexity of type 4.

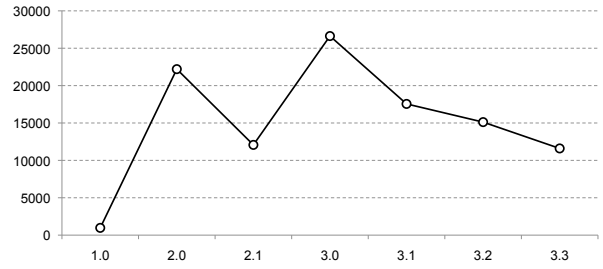

Figure 6. Total number of defect reports.

In search for further evidence related to law 2, we looked at the number of issues identified in Eclipse by the STAN tool, as an indicator of complexity of type 5. Fig. 7 displays the increase in number of issues computed by STAN (at class level). We observed that the total number of issues grows over time, closely following a linear trend ($R^2 = 0.996$). If we consider the total number of issues as a measure of complexity, then we conclude that complexity has been increasing.

In summary, only two (complexity as total size and as number of quality issues) out of five complexity indicators provided evidence of increasing complexity in Eclipse.

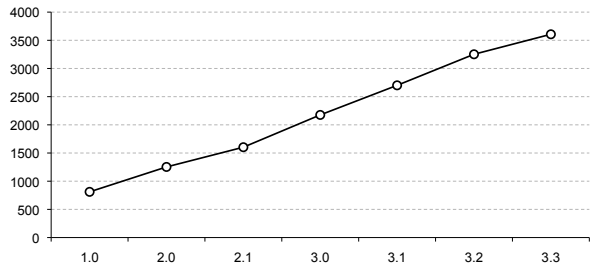

Figure 7. Total number of quality issues.

6. Results - subproject view

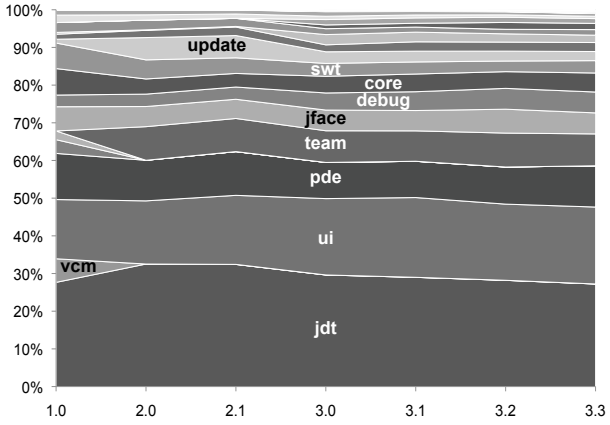

Figure 8. Relative subproject size (based on NOC) over releases.

In this section we analyse the evolution of Eclipse at the level of subprojects. Fig. 8 shows the relative contribution of each subproject to the total size of the system over time. By large, subprojects `jdt` and `ui` make up the largest part of Eclipse, accounting for about half of the total size of Eclipse. The figure also reveals that the relative size of the different Eclipse subprojects does not change much over different releases (the lines are roughly parallel).

Another observation that can be made in Fig. 8 is that the size distribution of subprojects is not uniform. Some subprojects are much larger than others. For example, in all studied releases the seven largest subprojects (32% of all the subprojects) account for more than 80% of the classes.

Fig. 9 displays the subprojects size (release 3.3), ranked in descending order, and using the logarithm of the size in NOC for the y-axis. With regards to law 6 of "continuing growth", the size of a few subprojects dominates the size trends that we presented in section 5.1 for Eclipse as a single entity. It is interesting to notice that the data in Fig. 9 closely follows a negative exponential model ($R^2 = 0.962$), which

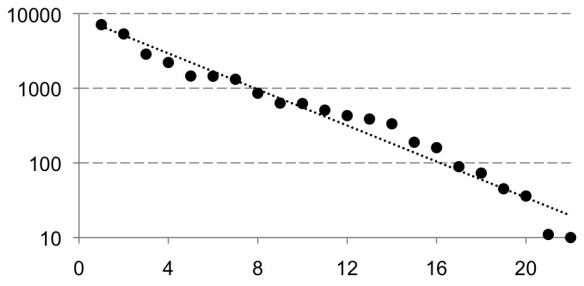

Figure 9. Ranked subproject size (in NOC) at release 3.3.

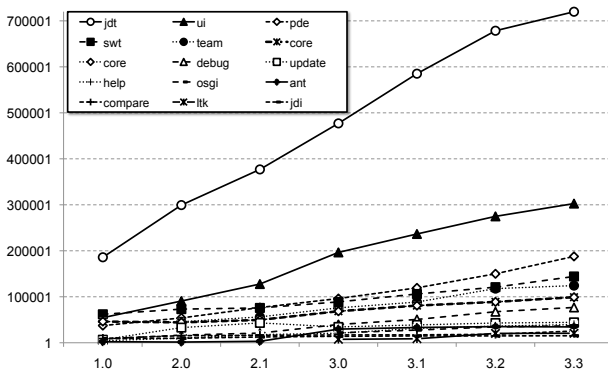

Figure 10. Size of 15 largest subprojects over releases (LOC).

corresponds to the linear trendline added to the figure. For the other releases, we fitted a negative exponential models, resulting in R^2 values of $0.968, 0.845, 0.856, 0.888, 0.911$ and 0.928, for 1.0 to 3.2 respectively. R^2 was higher than 0.9 and a reasonably good in three of the cases. It remains an open question to explain why Eclipse's subproject size behaved as described and, in particular, why a negative exponential model is a good fit in 4 of the 7 studied releases.

6.1. Growth - subprojects

As can be seen from Fig. 1, the number of subprojects increased approximately by 38 % from 16 subprojects at release 1.0 to 22 in release 3.3. Changes at subproject level are observed in releases 2.0, 3.0 and 3.1 only. In release 2.0, two subprojects were deleted (`scripting` and `webdav`), two were newly introduced (`platform` and `tomcat`), and one was renamed and reworked significantly (`vcm` became `team`). In release 3.0, four new subprojects were introduced (`ltk`, `osgi`, `search2`, `text`), providing additional evidence that this release was a major restructuring. Finally, release 3.3 added two new subprojects (`equinox` and `jsch`).

Fig. 10 shows the size of the 15 largest subprojects over releases, measured in LOC. We calculated the relative growth in LOC from release 1.0 to release 3.3. The value was positive in 20 instances, providing evidence for law 6 of "continuing growth" at subproject level. The relative growth could not be calculated for two subprojects that have been introduced at release 3.3 and have a single size measurement. Three other subprojects were present only at release 1.0 and then either removed or became another subproject.

We fitted five different trend models (linear, quadratic, exponential, power and logarithmic) to the growth data from 20 subprojects. Each of the other 5 subprojects had one data point only and were excluded from this analysis. We followed the same rules as we did for fitting models to the global growth (section 5.1). We found that, within these 20 subprojects, the growth of 15 subprojects can be modelled with a good fit with R^2 values in the range 0.998 to 0.923. We identified 7 linear, 4 superlinear and 4 sublinear trends. This group of 15 subprojects includes the eight larger subprojects, with size between 76 and 720 KLOC. Five subprojects (all below 45 KLOC) displayed growth patterns that do not provide a good fit to any of the models we fitted.

Unfortunately we were not able to extract data on the number of modifications for each subproject. This is an item for further work. However, additions to existing code often lead to modifications (e.g.. in order to link the new functionality to the existing code in some suitable way). For this reason, the evidence in support of law 6 of "continuing growth" can also be seen as providing indirect support for law 1 of "continuing change".

6.2. Complexity - subprojects

Due to lack of effort for data extraction we could only examine complexity types 1, 2, 4 and 5 (see section 5.3) at the subproject level. The evidence of increasing size in at least 20 out of 25 subprojects (positive relative growth) given in section 6.1 indicates that type 1 complexity has increased in all the 20 subprojects that have been evolved (i.e., present during more than one release). There is evidence that complexity of type 2 has increased for 4 subprojects only (the ones with a sublinear growth trend model).

To find out whether defect reports are an indicator of subproject complexity (type 4), we extracted the similar data of Fig. 6 for each subproject. We encountered an additional difficulty that the defect reports produced by Bugzilla cannot be mapped in a straightforward and one-to-one way to our notion of subprojects. In Bugzilla, some of the subprojects are classified as *products* (in particular `equinox`, `jdt` and `pde`), while most of the other subprojects are classified as *components* of the `Platform` product. Although

this may affect our results, Fig. 11 seems to indicate the same kind of behaviour for each of the subprojects as we encountered for Eclipse as a whole. Although clearly some subprojects have had more defects than others, we again observe a peak in releases 2.0 and 3.0. For the same reasons given when discussing Fig. 6 in section 5.3, no evidence can be found in Fig. 11 that complexity of type 4 has been increasing.

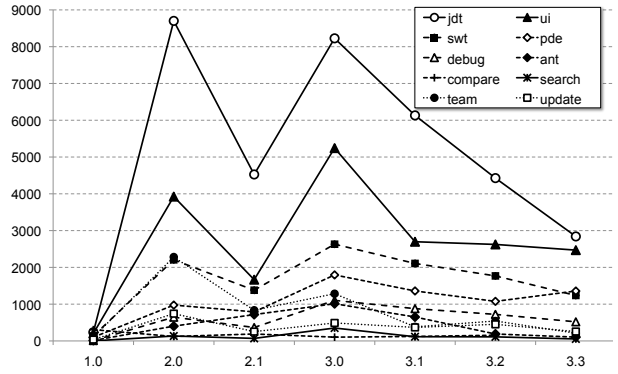

Figure 11. Number of defect reports in the 15 largest subprojects.

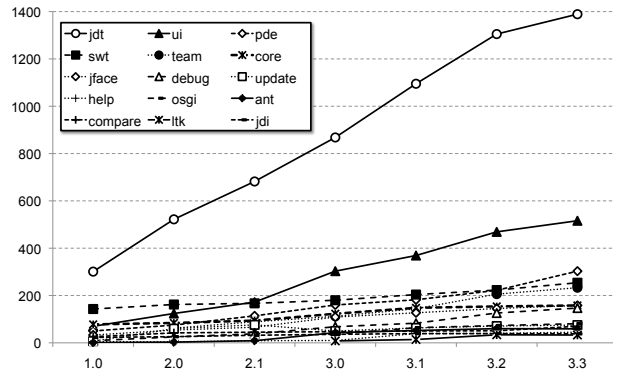

Figure 12. Number of STAN issues reported for the 15 largest subprojects.

In order to evaluate complexity of type 5, we computed the equivalent of Fig. 7 at subproject level, by counting the number of issues reported by STAN per release for each subproject. It is shown in Fig. 12. For 11 subprojects, the number of quality issues is always increasing over releases. Five subprojects have only one release interval for which the number of quality issues is constant or decreasing, 3 of which occur between releases 3.2 and 3.3, the most recent studied. We fitted models as we did in section 6.1. Increasing trends were identified for 14 subprojects with R^2 values in the range from 0.996 to 0.928 (6 were superlinear, 4 linear, and 4 sublinear). In 6 subprojects no trends were identified according to our criterion (R^2 was lower than 0.9). The remaining 5 subprojects had only one data point, so no trend could be found.

In Fig. 13 we show the distribution of STAN issues across subprojects. As was the case for the size distribution (cf. Fig. 8 and Fig. 9), a negative exponential model best explains the distribution (when compared to linear, quadratic polynomial, power, and logarithmic models).

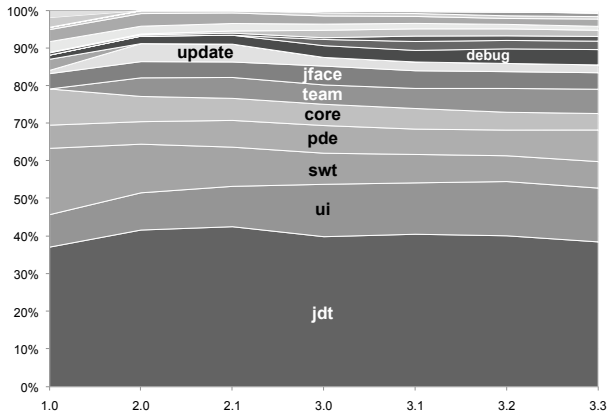

Figure 13. Distribution of number of quality issues for subprojects.

In order to check whether subproject size, class size and the number of quality issues were related across subprojects, we did the following: (i) first, we computed the ratio of LOC against number of classes for each subproject, assuming that a higher average number of LOC per class indicates a higher complexity (consistent with type 1 view); (ii) second, we computed the ratio of STAN issues against number of classes for each subproject (an indicator related to complexity of type 5). In both cases, we counted in how many releases the ratio was above average. We excluded all subprojects that were available in one release only. The results are summarised in Table 2. The values in the tables are fractions A/B where B is the number of releases in which the subproject is present, and A the number of times the value is above average. For example, `ant` has a value of 2/7 for "issues/NOC", indicating that the ratio is above average in 2 out of 7 releases. Observe that 2 out of 3 of the biggest subprojects (`ui` and `pde`) do not appear on this list because they are never above average. In contrast, three of the smaller subprojects (`swt`, `core` and `jdi`) appear to have a significantly higher relative complexity when looking at Table 2. In future work we intend to investigate further why this is the case.

Table 2. Relative complexity of subprojects.

subproject	issues/NOC above average	LOC/NOC above average
jdt	7/7	7/7
swt	6/7	7/7
jdi	7/7	5/7
core	5/7	6/7
osgi	3/4	4/4
ant	2/7	2/7
update	2/7	1/7
compare	7/7	
jface	4/7	
search	3/7	
debug	3/7	
platform	2/6	
team	1/6	
tomcat		6/6
text		3/4

7. Threats to validity

This section lists specific threats to the validity of our results. These are in addition to the more general threats to validity that one may encounter in similar studies [4].

Eclipse is a large system and there are different ways of measuring it. For practical reasons, we focused on the Eclipse SDK for Windows only. This explains why there are differences in measurement values (e.g., in the number of plug-ins) with respect to other studies (e.g., [14] that purely focused on Eclipse's architecture).

Law 2 states that "complexity increases unless work is done to maintain or reduce it". This means that, for rigorously assessing it, one should also measure the amount of *anti-regressive* (i.e., complexity control) work [7] (e.g., refactorings which have led to a decrease in complexity). Unfortunately, we could not do this due to the sheer size of Eclipse and our effort limitations. In future work, refactoring identification tools (e.g., [15]) could be helpful.

The five types of complexity (section 5.3) involve some assumptions that we were not able to check due to lack of data or effort to extract them. For example, a link between increasing complexity and a sublinearly increasing size trend (type 2) requires that the system have been evolved at a constant level of effort. Changes in effort level could trigger changes in growth and growth rate, even stronger than those potentially related to changes in complexity. Similarly, the use of defect data as an indicator of complexity (type 4) assumes that the testing effort is constant, since higher testing may lead to more defects found, despite lack of any significant changes in complexity.

When using STAN to compute metrics for the whole Eclipse SDK, for reasons of computer memory requirements we needed to restrict ourselves to computing the *class-level* metrics only, thereby excluding all data that could be gathered at the level of methods and fields. In practice, this means that we were not able to compute and study three well-known coupling and cohesion metrics: coupling between object classes (CBO), response for a class (RFC) and lack of cohesion for methods (LCOM) [2].

8. Related work

A partial survey of empirical studies of OSS that are related to the laws is reported in [4]. We briefly discuss below related work that is particularly relevant to our research.

Godfrey and Tu [5] studied the growth of Linux, and found superlinearity in its growth (size in LOC). In our study, we also found superlinear growth in Eclipse for size in plug-ings and in MBytes (downlodable source code). As seen in section 5.3, superlinearity does not support an increase of type 3 complexity. However, complexity has other dimensions (cf. section 5.3) and, in general, superlinear growth can be seen as an indicator of increasing type 1 complexity. In particular, for Linux, the frequent addition of device drivers by a large community of contributors can explain the observed increase in growth rate. For the evaluation of complexity, device drivers should be considered "external" to the studied system.

Herraiz *et al.* [6] studied the evolution of 13 OSS projects, finding superlinear growth in 6 of them, linear growth in 4 and sublinear in the remaining 3 systems. This study looked only at the size (in number of files and LOC) at the global level, finding similar results using any of the two measures. In our study we looked at a single system, but using a larger number of metrics and we considered evolution at both the global and subproject level.

Xing and Stroulia [15] investigated the structural evolution of Eclipse to find out which of the changes were due to refactorings. Their focus was different than ours, and was oriented towards detailed change information at the level of classes, interfaces, methods and fields. They only looked at the `jdt` subproject, which accounts for about one third of the total size of Eclipse (see Fig. 8). In addition, only the differences between 3 pairs of Eclipse releases were investigated (namely 2.0-2.1, 2.1.3-3.0, and 3.0.2-3.1).

Wermelinger *et al.* [14] studied the evolution of Eclipse to find out to which extent it complies with architectural design principles that have been argued to impact software maintainability. To this extent, they analysed the Eclipse architecture at the level of plug-ins. Our goal was not to study the architectural evolution of Eclipse, but to assess, more generally, whether Eclipse conforms to three of the laws of software evolution. However, our own approach complements the research of these authors in several ways. First, we relied on different data sources (i.e., source code, compiled code and bug reports) for our analysis. Second, our analysis was performed at a different level of granularity (mainly subprojects, classes and LOC).

9. Conclusions and further work

In this paper, we studied data from the popular open source Eclipse IDE, reflecting about six years of its evolution. We examined empirical evidence for three of the laws of software evolution, both at the global and "namespace" (subproject) level. We looked at the behaviour of about 17 indicators for Eclipse as a single entity and 7 or so indicators at subproject level. At the global level, Eclipse's data supports laws 1 and 6. Law 2 is only partially supported. At subproject level, we observed that only a few of the 25 subprojects concentrate most of the code. A negative exponential model seems to capture well the relationship between size and the size ranking. We identified a variety of size and complexity behaviours at subproject level. Our approach and results can help in the study of Eclipse's further evolution and in making comparisons to other IDE's (e.g., NetBeans) and other OSS in general.

The study we report in this paper can be extended in different ways, some of which have already been mentioned. One natural extension of this work is to evaluate whether Eclipse's evolution is consistent with the remaining five laws (laws 3, 4, 5, 7 and 8). For this it will be necessary to extract and analyse data from other data sources such as versioning systems (CVS and Subversion repositories), change request reports and community mailing lists. Another question for further research is how the Eclipse developer community evolves and how it relates to the technical evolution characteristics of Eclipse itself.

Currently, there is a lack of a generally accepted set of indicators, measurements and data analysis techniques to evaluate the laws of software evolution. In order to achieve this, further work is needed including comparative analysis of different possible indicators and approaches.

Since Eclipse follows a plug-in architecture, it would be helpful to study the evolution of the "external" Eclipse plug-ins (in terms of lifetime, quality, popularity, effort and so on) and how this relates to the evolution of Eclipse itself. In particular, it would be interesting to assess the evolution impact of a plug-in architecture on the quality and evolvability of the "ecosystem" made by the core Eclipse and the many external plug-ins.

Finally, further work will be required to establish whether our present findings and approach can be generalised by comparing the evolution of Eclipse to other systems that have similar characteristics (e.g., other OSS, other IDE, other systems that follow a plug-in architecture).

Acknowledgements. Drs M. Wermelinger and Y. Yu shared with us their insights about Eclipse. Israel Herraiz brought to our attention the topic of asymmetrical distributions in software size. Yann-Gaël Guéhéneuc provided helpful comments on an early draft of this paper. We thank C. Beck for clarifications about STAN. Support from the F.R.S.-F.N.R.S. through postdoctoral scholarship 2.4519.05 to the research stay of one of the auhors (JFR) at UMH is gratefully acknowledged.

References

[1] E. J. Barry, C. F. Kemerer, and S. A. Slaughter. How software process automation affects software evolution: a longitudinal empirical analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(1):1–31, 2007.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

[3] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley Professional, 2 edition, April 2006.

[4] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi. Empirical studies of open source evolution. In T. Mens and S. Demeyer, editors, *Software Evolution*, pages 263–288. Springer-Verlag, 2008.

[5] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 131–142, Los Alamitos, California, 2000. IEEE Computer Society Press.

[6] I. Herraiz, G. Robles, J. M. Gonzalez-Barahona, A. Capiluppi, and J. F. Ramil. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In *Proc. European Conf. Software Maintenance and Reengineering (CSMR)*, Bari, Italy, March 2006.

[7] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change*. Apic Studies In Data Processing. Academic Press, 1985.

[8] M. M. Lehman and J. F. Ramil. Towards a theory of software evolution - and its practical impact. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*, pages 2–11. IEEE Computer Society Press, November 2000. Kanazawa, Japan.

[9] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. In *Special Issue on Software Management*, volume 11 of *Annals of Software Engineering*, pages 16–44, 2001.

[10] M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proc. IEEE Symp. Software Metrics*, pages 20–32. IEEE Computer Society Press, 1997.

[11] Odysseus Software. STAN - Structure Analysis for Java. `http://www.stan4j.com`, Fall 2007.

[12] D. Rubel. The heart of Eclipse. *ACM Queue*, 4(8):36–44, October 2006.

[13] I. Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[14] M. Wermelinger, Y. Yu, and A. Lozano. Design principles in architectural evolution: a case study. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, 2008.

[15] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an Eclipse case study. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 458–468. IEEE Computer Society Press, 2006.