Analyzing the Evolution of Eclipse Plugins

Michel Wermelinger
Centre for Research in Computing
The Open University
Milton Keynes, UK
m.a.wermelinger@open.ac.uk

Yijun Yu
Centre for Research in Computing
The Open University
Milton Keynes, UK
y.yu@open.ac.uk

ABSTRACT

Eclipse is a good example of a modern component-based complex system that is designed for long-term evolution, due to its architecture of reusable and extensible components. This paper presents our preliminary results about the evolution of Eclipse's architecture, based on a lightweight and scalable analysis of the metadata in Eclipse's sources. We find that the development of Eclipse follows a systematic process: most architectural changes take place in milestones, and maintenance releases only make exceptional changes to component dependencies. We also found a stable architectural core that remains since the first release.

Categories and Subject Descriptors

D.2.8 [**Metrics**]: Product metrics; D.2.6 [**Programming Environments**]: Eclipse; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version control;
D.2.9 [**Management**]: Software configuration management

General Terms

Design, Measurement

Keywords

architectural evolution

1. INTRODUCTION

Our analysis of Eclipse's architectural evolution will be based on the metamodel in Figure 1, which shows the relevant concepts, and their relationships.

Each *release*[1] of the Eclipse SDK (except for release 1.0) provides one or more high-level *features*. Each feature may be composed of other, more specialised, features. For example, in release 3.3.1.1, feature sdk (we omit the org.eclipse prefix) includes feature platform which in turn includes rcp (Rich Client Platform).

Each feature is implemented by a set of *plugins*, Eclipse's components. For example, in release 3.3.1.1, feature platform is implemented by more than 70 plugins, including core.boot, compare, and platform, to name a few. Notice that features and plugins may have the same name.

Each plugin may *depend* for its compilation on Java classes that belong to other plugins. For example, the implementation of plugin platform depends in release 3.3.1.1 on eight other plugins, including core.runtime and ui. Each plugin *provides* zero or more *extension points*. These can be *used* at run-time by other plugins in order to extend the functionality of Eclipse. A typical example are the extension points provided by the ui plugin: they allow other plugins to add at runtime new GUI elements (menu bars, buttons, etc.).

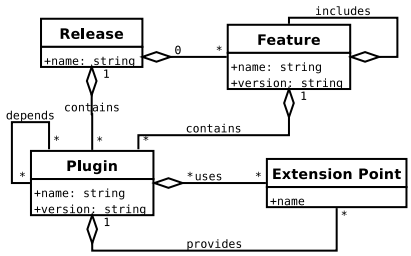

Figure 1: Metamodel for architectural evolution

In the remaining of the paper, we say that plugin X *statically depends* on plugin Y — the `depends` association in Figure 1 — if the compilation of X requires Y, and we say that X *dynamically depends* on Y if X `uses` an extension point that Y `provides`. Henceforth, we call plugins, extension points, and static and dynamic dependencies the *architectural elements*, or simply elements, of Eclipse. The architectural evolution of Eclipse corresponds therefore to the creation and deletion of elements over several releases.

There are various types of releases in the Eclipse project. In this paper we analyse *major releases* (e.g. 2.0 or 2.1) and the *maintenance releases* that follow them (2.0.1, 2.0.2, 2.1.1, etc.). In parallel to the maintenance of the current major release, the preparation of the next major release starts. The preparation consists of some *milestones*, followed by some *release candidates*. For example, release 3.1 was followed by milestone 1 of release 3.2 (named 3.2M1) and five other milestones, followed by seven release candi-

[1]The Eclipse project uses the term 'release' just for certain kinds of 'builds'. We only analyse stable builds meant for users, hence our preference for the term 'release'.

MSR'08, May 10-11, 2008, Leipzig, Germany.

dates (3.2RC1, 3.2RC2, etc.) until culminating in major release 3.2. Figure 2 shows part of the 47 releases we analysed, and their chronological and logical order. The logical order is indicated by solid arrows: each release may have multiple logical successors. The chronological order is represented by positioning the nodes from left to right: each release has a single chronological successor. Due to page width constraints, we split the timeline in two, at release 3.1. The dotted arrows indicate that some sequences of releases were omitted due to space constraints. We only did that when the chronological and logical orders coincide.

Having determined the objects of our analysis (the four kinds of architectural elements and the four kinds of releases), it remains to decide what to analyse. Some of the questions we would like to be answered are:

1. How many architectural changes occur between major releases and what is the overall growth of the system?
2. Is the architecture changed in maintenance releases?
3. Is there any difference between milestones and release candidates, in terms of architectural evolution?
4. Are there any deletions, or just additions?
5. Is there any substantial architectural core that is stable, i.e. that remains throughout all releases?

Question 1 is a first step towards future work to compare architectural and code growth and to check if Eclipse follows the same growth rates as other systems. Questions 2 to 4 should provide insights into the development process and help project managers check whether a systematic process is being followed. Questions 2 and 4 are also useful for developers of third-party plugins to be aware of any potential and unexpected compatibility problems. Question 5 is useful for Eclipse developers to know which parts of the architecture are fundamental and where architectural changes would likely have the biggest impact.

The next section explains how the mining was done to address the questions, and Section 3 reports on the results.

2. DATA PROCESSING

For each plugin there is an XML file, called `plugin.xml`, that lists the extension points provided and used by that plugin, and the other plugins it depends on for compilation. Since release 3.0, the static dependency is in another file, `MANIFEST.MF`, which is not in XML format. These metadata files are hence a straightforward source of dependency information between plugins, saving us from having to delve into their source code.

We first considered extracting the metadata files for each release directly from the CVS repository, for example by checking out all files with tag `R_3_1` in order to obtain the information about release 3.1 (CVS tags cannot include periods). However, after a while we found out that there is no direct correspondence between CVS tags and releases. In other words, comparing the set of metadata files obtained from the CVS repository with the set of those included in the actual releases, we found that often the two sets didn't coincide. We also tried to check out the files according to the known date of the release, but again there was a mismatch. In fact, the Eclipse project uses for each release a complicated file that indicates which source files should be checked out for that release. The input to our analysis is therefore not a CVS repository, but a set of compilable source code archives, one per release we wish to analyse. How each source code archive was obtained is not of concern to our tool, making it independent of any configuration management system. In our case, for each of the releases we analysed, we downloaded the source code of the whole SDK from http://archive.eclipse.org or its mirrors.

The source code is first processed by some shell, AWK and XSLT scripts that extract the information about the existing architectural elements from the `plugin.xml` files (and `MANIFEST.MF` files, depending on the release). The result of this processing is a text file in Rigi Standard Format (RSF) [2], that captures the relations of Figure 1.

Next, we use Crocopat [1], a relational calculator, to take as input the relations extracted from the metadata and produce as output metrics and further relations. In particular, for each release we compute the dynamic dependency relation, the missing and unused extension points, the missing and unused plugins, and the sizes of those sets. An element is missing if it is required but not provided, and unused if it is provided but not required by any other element.

The missing plugins and extension points enable us to detect compile- and run-time problems, whereas the unused plugins and extension points tell us how extensible the architecture is. A completely self-contained and closed architecture would have no missing nor unused elements.

Having the above sets, other Crocopat scripts compute the architectural elements that were kept, added and deleted between any given pair of releases. However, it does not make much sense to compute the changes between *all* pairs of releases. Instead, we let users define a sequence of releases $r_1, r_2, \ldots, r_n$ they wish to analyse, and the tool then only computes the changes between consecutive releases in the sequence. The tool also computes the elements that were kept between r_1 and r_n, in order to see what (if any) is the core stable architecture during the whole sequence.

Finally, all metrics are put into automatically generated spreadsheets that conform to OpenOffice's XML format.

3. RESULTS AND ANALYSIS

We have defined two sequences of releases to analyse. One sequence includes the major and maintenance releases from $r_1 = 1.0$ until $r_{20} = 3.3.1.1$. Another sequence we analyzed includes all milestones and candidate releases that led from $r_1 = 3.1$ to $r_{30} = 3.3$.

For the purposes of analysing the architectural evolution, it makes more sense to order the releases by their numbers, rather than by their dates. For example, whereas the chronological order is 3.1, 3.2M1, 3.2M2, 3.1.1, 3.2M3, 3.2M4, 3.1.2 (see Figure 2), we either follow the maintenance branch sequence 3.1, 3.1.1, 3.1.2, ..., 3.2 or the milestone branch sequence 3.1, 3.2M1, 3.2M2, 3.2M3, ..., 3.2.

For both release sequences, we found out that the number of missing plugins and extension points is always zero, i.e. there are no compilation or run-time errors due to missing components. We can only conjecture whether the result would be the same if we had taken nightly builds into account, which unfortunately are not accessible from the archive site. We expect these two metrics to be more relevant when analysing applications built on top of the Eclipse platform, as those numbers would indicate "how much" of the SDK is required in order for the application to work.

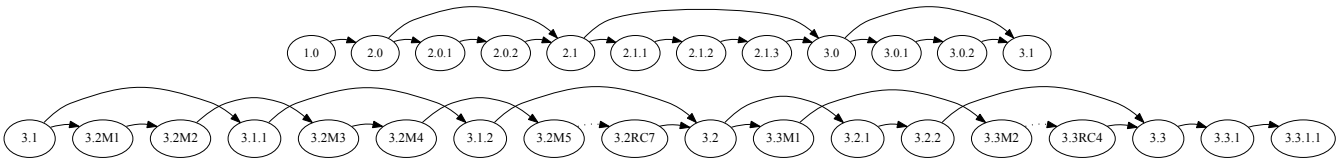

Figure 2: Chronological and logical sequence of analysed releases

To represent the evolution of the other metrics, like the number of plugins, we adopted bar charts, the height of each bar representing the total number of architectural elements in a given release. Since the total number is the sum of the elements kept and added w.r.t. the previous release, we divide each bar in two sections, the lighter one representing the added elements, the darker one the elements kept. We further subdivide the latter section, showing in the darkest tone the elements that have been kept since r_1. By definition, the bar for r_1 is completely drawn in the darkest tone. Furthermore, if the number of elements kept by a release is smaller than the total number of elements of the previous release, then elements have been deleted. To make this more explicit, we extend each bar below the zero axis by as many elements as deleted. Figure 3 shows the bar charts for the first sequence and Figure 4 shows for the second sequence we defined. From top to bottom, the bar charts show the evolution of the number of plugins (NP), extension points (NE), static dependencies (NSD), dynamic dependencies (NDD), and unused extension points (NUE). In each chart we show the number of elements added (A), kept from the previous release (K), kept from r_1 (K1), and deleted (D).

Observing the bar charts, we can proceed to answer the questions posed in the introduction.

1. We can observe that the architecture has grown by a factor between four and five. The highest growth rate occurred from release 1.0 to release 2.0. Although release 3.0 has smaller growth rates (and even a drop of 1% in plugins), it introduces, in absolute numbers, the major architectural change in the Eclipse project, due to the many additions and deletions to all four kinds of elements.

2. Figure 3 shows that no plugins or extension points are added or deleted in any maintenance release. One cannot see it in the reduced size charts, but there are a few very small dependency changes in maintenance releases. For example, release 2.0.1 removed three static dependencies and release 3.2.1 added one dynamic dependency.

3. Figure 4 shows that architectural changes can occur in both milestones and release candidates, but there are many more changes (both in size and frequency) during milestones.

4. As both figures show, additions make the bulk of changes, deletions being comparatively few and small in size. In particular, there are hardly any deletions in candidate releases, with the notable exception of the plugins deleted in 3.3RC3. Most deletions occurred in release 3.0, with many static dependencies being also removed in 3.1. Given that the number of plugins actually increased in 3.1, this indicates a major refactoring effort in which the implementation of plugins was changed to decrease their dependencies.

5. Surprisingly, a rather stable architectural core is apparent throughout the 6 year span: 11% (= 24) of the plugins and 14% of the extensions points of release 3.3.1.1 come from release 1.0. Among the core plugins are pde, swt, help, jdt.launching, core.runtime, core.resources and various user interface and documentation plugins. The services these plugins provide via their extension points have been reduced slightly in release 3.0, but the core dynamic dependencies have remained the same. The stability is of course much greater in the last 2 years: 60% of plugins and 77% of extensions points of release 3.3 come from release 3.1.

The evolution of the unused extension points is also interesting, with many changes, leading to none of the original unused extension points being still offered to third-party components. Releases 3.2 and 3.3 have removed several such extension points, which of course may have impact for the compatibility between older third-party applications and recent Eclipse releases. In spite of the many changes since release 1.0, it's interesting that the "openness" of Eclipse's architecture has remained similar: the percentage of unused extension points (w.r.t. the total number of extension points), grew just from 12% in release 1.0 to 16% in release 3.3. But these figures might be misleading, as many of the extension points used by the SDK itself can also be used by third-party components.

4. CONCLUDING REMARKS

In this paper, we briefly presented a tool that can efficiently extract compile-time and run-time dependencies between Eclipse components and hence form the basis for structural evolution studies of this popular open-source project. The tool is scalable because, as befits architectural analysis, it treats components as black boxes: only relatively few metadata files are processed, disregarding the many Java source code files. Our tool is a set of scripts that read/write text files in RSF or XML format. This light-weight approach makes it easy to extend the tool and integrate it with existing tool chains. This is further facilitated by our tool being independent of any configuration management system.

We have applied the tool to two sequences of releases of the full Eclipse SDK, i.e. including PDE, JDT, Platform, RCP, etc. We have found that the Eclipse project follows a systematic process, with most architectural changes taking place in milestones, a few in release candidates, but during maintenance only exceptional changes in the dependencies of existing components take place. The SDK is self-contained (no missing plugins) and uses most of its own extension points, less than a sixth of them being provided exclusively to third-parties. However, those extension points

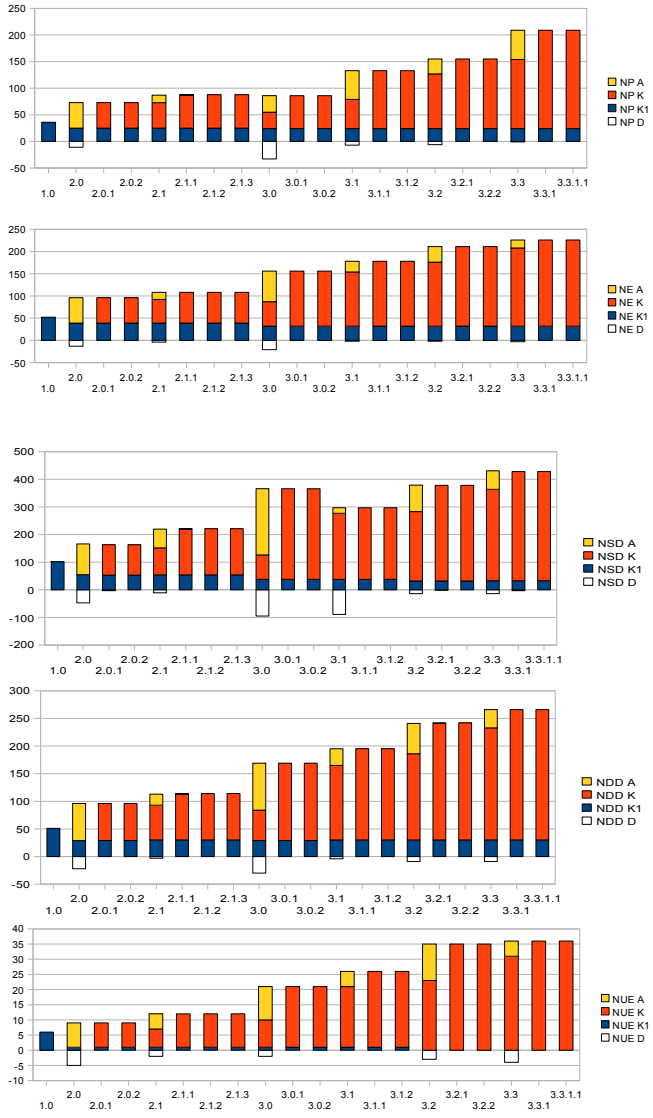

Figure 3: The evolution of architectural elements along major and maintenance releases

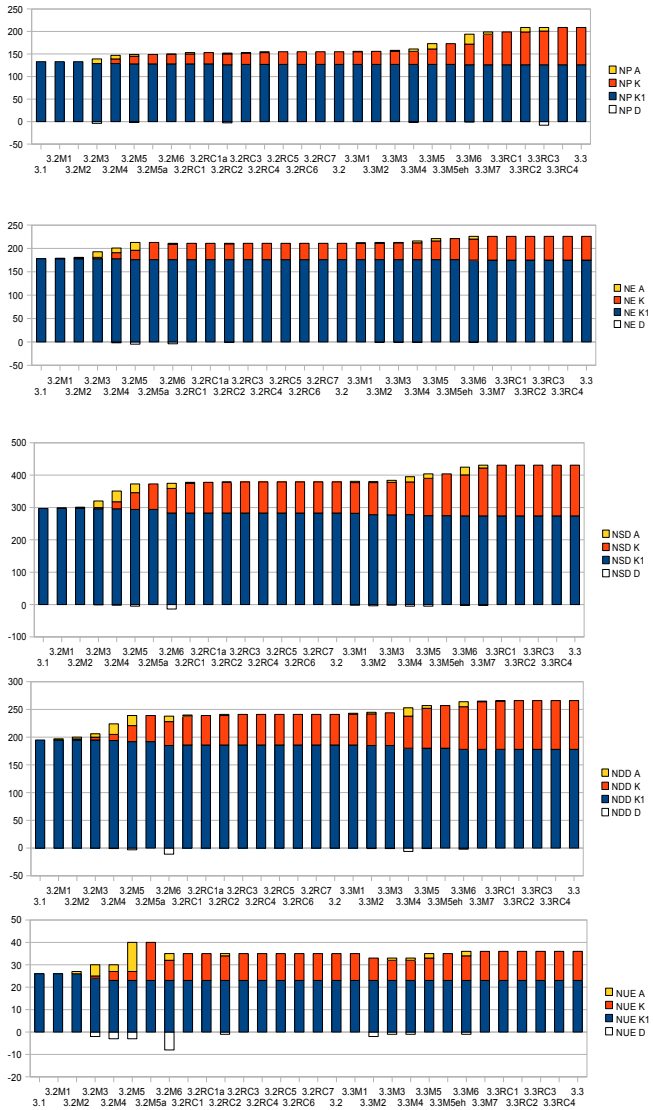

Figure 4: The evolution of architectural elements along milestones and release candidates

have changed substantially throughout the releases, raising compatibility issues for third-party components.

Release 3.0 introduced major architectural changes, with many additions and deletions to all elements. The drastic reduction of features in that release, making them more coarse-grained, reflects this. Our light-weight architectural metrics also allows to observe internal changes to components: release 3.1 reduced the static dependencies while adding new plugins, which indicates a refactoring effort.

In spite of all changes throughout 6 years of development, there is a stable architectural core: most of the elements of release 1.0 were kept until release 3.3.1.1, making up more than a tenth of its architecture. Whether this indicates good architectural foresight or not, can only be said after comparing these numbers with other long-lived systems.

Encouraged by the insights into Eclipse's architecture and its evolution that we could obtain just from a handful of metrics, we plan to continue this line of research. In future work we intend to analyse the evolution of third-party Eclipse components and applications. This will also give us further insights about the SDK, namely which of its plugins are most useful to other developers.

5. REFERENCES

[1] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Software Eng.*, 31(2):137–149, 2005.

[2] K. Wong. *The Rigi User's Manual, Version 5.4.4*, June 1998.