Analyzing the Evolution of User-Visible Features: a Case Study with Eclipse

Daqing Hou[†] and Yuejiao Wang[‡]
Electrical & Computer Engineering[†]/Computer Science[‡], Clarkson University, Potsdam, NY 13699
{dhou, wangyue@clarkson.edu}

Abstract

Integrated Development Environments (IDEs) help increase programmer productivity by automating much clerical and administrative work. Thus, it is of great research and practical interest to learn about the characteristics on how IDE features change and mature. To this end, we have conducted an empirical study, analyzing a total of 645 "What's New" release note entries in 7 releases of the Eclipse IDE both quantitatively and qualitatively. It is found that majority of the changes are refinements or incremental additions to the feature architecture set up in early releases (1.0 and 2.0). Motivated by this, a further analysis on usability is performed to characterize how these changes impact programmers effectiveness in using the IDE. We summarize our study methodology and lessons learned.

1. Introduction

There has been a long history of research and development in Integrated Development Environment (IDEs) (see, e.g., [10, 11, 9, 3]). The competition between major Java IDEs can be intense. [1]

We are interested in understanding how IDE features have evolved for several reasons. First, IDEs are important productivity tools where many programmers spend much of their workday. An account of how IDE features have evolved helps us better understand the IDE problem space. Second, such an account of feature evolution may also help IDE developers better plan the direction where their IDE should head to. Third, this study analyzes how IDE usability has evolved. As evident in a recent quality assurance report available at NetBeans' web site [2], the evaluation and comparison between competing IDEs are being done at a level of details where usability concerns are considered the core. Thus, findings on how usability evolves may particularly be of great interest to IDE developers who are eager to improve their product. More generally, such findings may also be applied to the development of workpiece software applications like word processors and spreadsheets.

[1] `javaworld.com/javaworld/jw-03-2008/jw-03-java-ides0308.html`. URLs verified June 28, 2009.

[2] `http://qa.netbeans.org/cvut/2008-RomanHak_Report.pdf`

To study IDE feature evolution, we chose the Eclipse Java IDE (a major subset, to be precise) as the study subject because Eclipse has a reasonably long history, offers a rich set of features, and is open-sourced with much development information available. We relied on Eclipse's release notes as a main source of information for its feature evolution. Our approach to feature evolution is similar to the reverse engineering approach in that both approaches recover information from existing artifacts. The main difference is that in our approach, we analyze release notes manually rather than source code automatically.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 describes the study methodology. Two analyses are performed on feature evolution. Section 4 presents the result of an activity-based analysis, and Section 5 an analysis of usability evolution.

2. Related work

Many studies of system growth or evolution from the software-maintenance perspective address evolution as changes in the size and relational complexity of the code base of the product (e.g. [2, 4, 7]). There have been few studies of feature evolution that are code-independent, where aspects of feature architectures are objectively defined and measured temporally.

To our knowledge, the two most closely related work are [6] and [1]. In [6], Hsi and Potts propose to study feature evolution with a feature architecture that consists of three views (*morphological view* for GUI, *functional view* for operations available to a user, and *object view* for the entity and relationship within a domain). Similar to us, they also observed that their study subject (MS Word) was experiencing "conservative growth". However, their emphasis is on tracking "true" conceptual change in a domain rather than usability. In fact, it is not clear how their proposed feature architecture can be made to express usability concerns. We believe that usability concerns should be accounted for in the feature evolution of IDE and other similar systems.

In [1], Antón and Potts study the evolution of user-visible services in the 50-year history of a telephony system within a major city. By manually analyzing the user manuals, they profile the burdens and benefits associated with each telephony service (*functional morphology*) and iden-

tify several evolution patterns in how services were introduced in the system over time, for example, that the functional evolution of the system is punctuated rather than incremental or gradual. The main difference between the two studies is in the models used to code the raw data. Motivated by research in goals-driven requirement engineering, they developed and adopted a burden-benfit model. In our study, we used two models, one based on development activities and the other on usability. This interestingly demonstrates that the two studies are instantiation of a same high-level research methodology where models can and should be chosen based on the particular goals of a study.

Wermelinger, Yu, and Lazono used Eclipse as a subject to investigate the relevance of some structural design principles [12]. Hou investigated the evolution of the internal design of the Eclipse Java editor [5].

3. Study methodology

Table 1. 8 major releases of Eclipse (release dates and distribution of 645 entries).

Releases	Build Date	#Entries
3.4	Tue, 17 Jun 2008	70
3.3	Mon, 25 Jun 2007	91
3.2	Thu, 29 Jun 2006	79
3.1	Mon, 27 Jun 2005	113
3.0	Fri, 25 Jun 2004	113
2.1	Thu, 27 Mar 2003	70
2.0	Thu, 27 Jun 2002	109
1.0	Wed, 7 Nov 2001	-

In this study, we investigate the feature evolution in 8 consecutive major releases of the Eclipse IDE, whose release/build dates are depicted in Table 1. These releases represent approximately 6.5 years of development. It took about 2.5 years to deliver 2.0, 2.1, and 3.0, and another 4 years for the 4 subsequent releases. There are another 14 minor releases between the major ones, which were excluded because they contained mostly bug fixes or internal design restructuring not directly accessible to the end users.

Our study is primarily based on manually analyzing and coding the release notes for the 7 major releases between 2.0 and 3.4, inclusive. We used the release notes bundled as part of the help content rather than the ones archived on the internet as the former appeared to contain more details and to be more accurate. To help control the scope of study, we chose to limit our investigation to the Java Development Tool and the Eclipse Platform. In particular, we did not look at work on the Eclipse infrastructure like SWT and plugin engineering or features that support revision control. The right column in Table 1 shows the number of entries each release note contains. We studied totally 645 entries. An sample entry from 3.0 is shown as follows.

> Title: Toggle Comment Command
>
> The old Source > Comment and Source > Uncomment commands in the Java editor have been replaced by the Source > Toggle Comment (Ctrl+/) command that uncomments the currently selected source lines if all of them are commented and comments them otherwise. ...

To gain a high-level understanding and to see trends and patterns, a conceptual model is needed to reduce and simplify the entries. A conceptual model is made of multiple categories into which the entries can be coded. These entries are then counted by category for each release and the resulting data can be used to detect patterns and trends.

Each conceptual model may reflect a different purpose. Since our initial goal was to see over time what kinds of service the IDE provides and how features are evolved to assist programmers in the programming activities, we first used a simple model that is based on development activities. One conclusion from this first analysis is that usability is a major component in the evolution of Eclipse. Consequently, a second model is used to study Eclipse's usability evolution.

Understanding and coding the release note entries took the largest amount of effort. In order to be able to accurately interpret the 645 entries, we relied on running the corresponding releases of Eclipse. This was necessary because Eclipse contains so many features and details that even experienced users are likely to have used only a relatively small fraction of them [8].

It is critical to find out the true intention behind an entry, and care must be taken to truly understand what an entry is really about before drawing a conclusion. It is especially error-prone to categorize based only on a superficial reading of an entry title or description. For example, the sample entry shown above is titled "Toggle Comment Command". Since the "Comment Command" in Eclipse is an editing feature that can quickly comment out a selected region of code, it may be tempting to categorize this entry as *editing*. However, the actual work represented by this entry is that the "Comment" and "Uncomment" commands under the "Source" menu in 2.1 were combined and replaced by the single "Toggle Comment" command in 3.0. The nature of the work done behind this entry is providing better usability, not a new editing operation. Therefore, it was categorized as *usability* rather than a new function. As another example, Eclipse has a feature that assists the programmers restoring code from the local history. An entry in 2.1 titled "Multiple Method Restore." By reading the entry description, it became clear that this entry should also be coded as *usability* rather than something else.

Entries vary in the amount of details they contain, perhaps because they are contributed by different individuals.

One difficulty was to decide whether an entry should be split into multiple smaller ones or alternatively, given more weight if it appears to contain more contents. For example, entries on refactoring, code assists, and code fixes tend to list multiple work items under separate bulletin points, each representing, for example, a new refactoring. It may be okay to measure and compare such entries according to the number of items they contain as long as their categories are comparable. But it becomes unclear what it means when the measure for such an entry on refactoring is compared with that for an entry on a view involving multiple GUI changes. To simplify the coding and the interpretation of results, we decided to count an entry only once as a whole unit instead of attempting to split it, and only split entries when it absolutely makes sense to do so. [3]

To assure the coding quality, both authors categorized the entries independently and compared the results. Disagreements were resolved by discussions between the authors to reach consensus. The categorization took several passes before a stable version of coding was obtained. (Unfortunately, we did not record the exact numbers of disagreements during the coding process. But by conservative estimation, the disagreements were reduced from about 30% initially to less than 5%.)

4. Activity-model-based analysis

Our first model categorizes IDE features by development activities, including *project setup*, *code manipulation (read and write)*, *build and run*, *debugging*, and *testing*. Soon after the coding process was started, it became obvious that some entries purely represent work about graphical user interfaces and user interactions. Thus, another category, *usability*, was added for such changes. Note that this category is exclusively for features that address *only* usability concerns. A feature that belongs to both *usability* and another category will not be categorized as *usability*. There are also 16 changes that do not belong to any of these categories. The *others* category was added to group such changes. The coding result is depicted in Table 2.

Some comments can be made on the overall trends shown in Table 2. It can be seen that 2.0, 3.0, and 3.1 produce more changes than the other four releases. It seems that the number of entries produced by each release is declining as the product gets older. Code manipulation (*read* and *write* combined) consistently represents the largest area of work in all seven releases. Overall, fifty percent of the release note entries concern code manipulation. This is not surprising as much of what an IDE does is assisting programmers in dealing with code.

There are always some non-trivial numbers of *usability* entries in the seven releases. Overall, both the raw number of *usability* entry and their percentage (not shown in Table 2) are increasing over releases. It seems that over time, Eclipse developers were paying more attention on the usability of user interface and interactions.

4.1 Eclipse was experiencing gradual evolution

During the coding process, we observe that the growth of the Eclipse feature architecture has been conservative and the growth of "true" new features slow. Instead, most new work either creates a new instance of an existing feature category (like adding a new refactoring or a new quick assist) or refines an existing feature (like adding filters to a view). This observation motivated the detailed analysis of Eclipse usability evolution in Section 5. In what follows, we present two supporting evidence for this observation.

Since it is known to be costly to build a full feature architecture [6], we used as proxies, the most frequently used 10 views and 10 navigation and search commands identified in a study of Eclipse usage [8]. This is a reasonable approach as Eclipse mainly consists of views and commands. The standard Eclipse 3.4 contains 42 views, 16 of which are out of scope for our study as they are about plugin development, revision control, and help. For the remaining 26 views, including the 10 most frequently used, we found that 20 of the 26 views were introduced in early releases 1.0 and 2.0, and that all 10 most frequently used navigation commands were also added in early releases 2.0 and 2.1. These would suggest that work on views and commands in subsequent releases are more likely to be enhancements rather than additions of new features at the architectural level.

To gather more direct evidence, another analysis was performed on all the features that appear to be "new" in 3.3 and 3.4. We found that only 3 features in 3.3 and one in 3.4 may be considered conceptually new. That is, only 4 out of 28, or a ratio of 1/7 "new" features are really new. The other features either enhance or refine existing features, or are considered only peripheral to the core feature architecture, which are not "true" new features [6].

5. Analysis of usability evolution

To seek out a high level understanding how Eclipse usability evolves, we iteratively built a usability model consisting of 9 usability categories and coded the release note entries accordingly. Out of the 645 release note entries, 390 (60.5%) were identified as having contributed to at least one usability factor, including the 115 user interface usability entries shown in Table 2. In the remaining 255 entries without a usability label, 59 are about refactoring, which were not assigned a usability label. Thus, only 196 entries are truly not related to usability. They are due to language extension, secondary or only peripheral to the core features. In the remaining of this section, we summarize the high level observations on Eclipse's usability evolution.

[3] A small number of entries (less than 20) that mix work items from different topics, like refactoring and code fixes, were split.

Table 2. Distributions of 645 release note entries from seven releases of Eclipse IDE.

Releases	Usability	Setup	Write	Read	Build/Run	Debug	Test	Others	Total
2.0	13	11	24	30	8	17	6	0	109
2.1	8	2	24	18	4	11	3	0	70
3.0	19	2	25	38	12	15	2	0	113
3.1	16	19	30	29	2	13	1	3	113
3.2	18	7	15	20	11	6	2	0	79
3.3	21	7	22	22	1	12	2	4	91
3.4	20	5	11	16	1	7	1	9	70
Sum (%)	115 (18%)	53 (8%)	151 (23%)	173 (27%)	39 (6%)	81 (13%)	17 (3%)	16 (2%)	645

Working more efficiently with structures. Programs often contain rich structures that need to be explored, like type hierarchies, use and definition of identifiers, and hierarchical module structures. Indeed, IDEs have provided some basic functionalities in supporting the navigation of such structures. For example, the Package Explorer view can be used to navigate hierarchically the packages, classes, and members in a Java project; the Javadoc hover can display the Javadoc for a selected program element directly inside the editor, saving the programmer from navigating to the file that contains the Javadoc. We noticed enhancements to these basic functionalities in three general areas, tighter integration of operation and information into context (e.g., the "Bread Crumb" feature added in 3.4), greater automation (e.g., Smart Pasting of Code Snippet), and information pushing (e.g., the various iconic hints about program information), all involving structural information.

Broader applicability of features. A feature initially created and applied in one context over time may be used in other contexts or applied to new kinds of data. This may seem obvious for generic user interface services like undo, text filter, or search; and common utilities like spell checking. Some IDE-specific features were also broadly applied (e.g., the working sets feature) or generalized (e.g., between the Java editor and the generic text editor).

Standard operations in views. Views have long been identified as a useful mechanism for presenting and navigating program information in IDEs [9]. A view is typically used to display a table or a tree hierarchy, for example, the problems view and the type hierarchy view in Eclipse. Although each view has its own purpose, they share several common features generally applicable beyond IDEs to any view that manages such structural information. These include *sorting*, *filtering*, *grouping*, *item navigation*, *navigation history*, and *link with editor*. It then became interesting to know whether Eclipse had adopted these features in a planned and systematic fashion across all the views. To answer this question, we examined the eight releases to find out (1) for each feature, the earliest release where the first instance of the feature appeared, and (2) for each view, the earliest release where a given feature was used. Based on this examination, we conclude that the introduction of the standard view features in Eclipse exhibits quite some irregular behavior, and, thus, could have been better organized and benefited from a stronger global coordination. These issues are important because they may have negatively affected users as they may be less clear as to what they should expect from views. They also impede the users in seeing the connection among the same operations offered by different views. These lessons can be applied beyond IDEs to any software that needs to present information in a similar way.

References

[1] A. I. Antón and C. Potts. Functional Paleontology: The Evolution of User-Visible System Services. *IEEE Trans. Softw. Eng.*, 29(2):151–166, 2003.

[2] L. Belady and M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15(3):225–252, 1976.

[3] J. R. Cordy, N. L. Eliot, and M. G. Robertson. TuringTool: A User Interface to Aid in the Software Maintenance Task. *IEEE Trans. Softw. Eng.*, 16(3):294–301, 1990.

[4] M. W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In *ICSM'00*, pages 131–140, 2000.

[5] D. Hou. Studying the evolution of the eclipse java editor. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 65–69, 2007.

[6] I. Hsi and C. Potts. Studying the Evolution and Enhancement of Software Features. In *ICSM'00*, pages 143–151, 2000.

[7] T. Mens, J. Fernández-Ramil, and S. Degrandsart. The Evolution of Eclipse. In *ICSM'08*, pages 386–395, 2008.

[8] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[9] S. P. Reiss. PECAN: Program Development Systems that Support Multiple Views. *IEEE Trans. Softw. Eng.*, 11(3):276–285, 1985.

[10] W. Teitelman. Automated Programmering - the Programmer's Assistant. In *Proceedings of the Fall Joint Computer Conference, AFIPS*, pages 915–921, December 1972.

[11] W. Teitelman and L. Masinter. The Interlisp Programming Environment. *Computer*, 14(4):25–33, 1981.

[12] M. Wermelinger, Y. Yu, and A. Lozano. Design Principles in Architectural Evolution: A Case Study. In *ICSM'08*, pages 396–405, 2008.