Studying the Evolution of the Eclipse Java Editor

Daqing Hou
Electrical and Computer Engineering
Clarkson University, Potsdam, New York 13699
dhou@clarkson.edu

ABSTRACT

In evolutionary software development, knowing how design evolves with features can be valuable in guiding future projects. It helps answer questions like "How much upfront design should and can be done?" and "How and why are designs changed?" To shed light on these questions, we report on a study of the evolution history of the Eclipse Java editor. We find that the MVC-based design was cleanly laid out in the beginning of the project and carefully followed and maintained, which has contributed positively to the continuous growth of the editor features. Although design changes at the individual feature level happened for reasons like extensibility and reusability, they appear to be local and manageable. The AST facility is a key service that enables more than one half of the Java editor features.

1. INTRODUCTION

Good software designs enable concurrent development, facilitate evolution and maintenance, and support reuse [7]. Ideally, we would prefer to lay out the complete design in a project up-front. In practice, however, many successful software systems are developed in an evolutionary fashion. One reason is because they must be constantly improved to remain useful [8]. Consequently, new features are added and important design decisions made dynamically in the process.

Studying the evolution of a successful system can improve the understanding of both the practice of evolutionary development and the particular domain to which the system belongs. In particular, it helps conceptualize the feature space for the problem domain, and understand how designs have supported and evolved with feature evolution during the process. Good designs should anticipate and accommodate future changes [7]. Designs that hinder the evolution of features need to be reworked to remove the obstacles.

In this paper, we report on a case study on the evolution of the Eclipse Java editor. Our goal was to track down how the Java editor was designed initially and evolved subsequently to host all of the features that it has acquired over time. We classify the features according to whether they need the support of ASTs, among other criteria. We find that more than one half of the Java editor features make use of ASTs. A noticeable pattern is the continuous growth of feature groups like coding assists. Modular design supports the addition of new features as well as the incremental growth of feature groups. In particular, the MVC architecture established in early releases appears to have accommodated new features nicely and remains stable throughout the releases we examined, and the modular design for a feature group not only enables the smooth addition of sub-features, but also protects them from the ripple effect of other design changes. As an example, we describe some design changes made to coding assists, which was driven primarily by the need to support reuse and extension.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Eclipse Technology eXchange Workshop (ETX), October 21, 2007, Montréal, Québec, Canada.

1.1 Methodology

In this study, we relied on artifacts publicly available for the multiple releases of Eclipse [10], including the "new and noteworthy" notes, source code, and executables.

We extracted the evolution of the Java editor features (Table 1) by running six versions of Eclipse (1.0, 2.0, 2.1, 3.0, 3.1, and 3.2) and studying the "new and noteworthy" notes for releases 2.1, 3.0, 3.1, and 3.2. A "new and noteworthy" note summarizes the new features and major improvements in a release. These notes are necessarily brief. Even though experienced in using the Java editor, we did not know all of its features. To verify what is described in the notes, we installed and ran six versions of the Eclipse IDE. This has also helped us identify features not covered by the notes.

To investigate design evolution, we studied the Eclipse source code obtained from the CVS repository [9]. The earliest release available is 2.0. We started with version 3.2 to recover the design of the Java editor, and then proceeded to study and compare with the designs of earlier versions. Subsequent design recovery took considerably less effort than the first one.

1.2 Organization

The remainder of this paper is organized as follows. Section 2 provides an overview and classification of the Java editor features. Section 3 summarizes observations on the functional evolution of the editor. Section 4 investigates how the editor design supports and evolves with the feature evolution. Section 5 surveys related work. Finally, Section 6

concludes the paper.

2. FEATURE CLASSIFICATION

As can be seen from Table 1, the Java editor offers a rich set of features. At the basic level, the Java editor presents Java code and supports code browsing and editing. With the assistance from the compiler back end, it reveals to a developer deep information embedded in the source code like the existence of problems and relationships between program elements. To help grapple with these features, in particular, the role that AST support plays, we classify them into four categories: (1) simple, text model-based features, (2) advanced, AST-based features, (3) coding assists, and (4) management and miscellaneous. Coding assists are separated from AST-based features because they not only require AST support, but also rewrite code.

2.1 Simple, Text Model-based Features

The text model of code is character and line oriented, and is used to support the display of program text as well as implement features like line numbers, quick diff, block commenting, and syntax highlighting. Note that syntax and semantic highlighting are different. Syntax highlighting, based on lexical patterns, highlights keywords and JavaDocs. Semantic highlighting is AST-based and highlights, for example, the invocation of static methods.

2.2 Advanced, AST-based Features

This category of features, in addition to the text model, also requires the support of the parsing and semantic analysis from the compiler. These features help integrate information and increase a developer's awareness of the status of the program text in the editor. Occurrence marking and hovering are examples of information integration. With the support of ASTs and bindings, occurrence marking highlights within the current compilation unit all occurrences of the identifier that is currently selected. Another example is hovering. For example, hovering over a class name will display the JavaDocs for the class. Reconciliation is an example of increasing awareness. It parses and checks source code as it is typed. Consequently, feedback and syntax and semantic highlighting are done to the code, and a developer is informed of code status in real time. Another awareness feature, prominent status indication, is the red icon on the top of the overview ruler that indicates that there are annotations in the current compilation unit.

2.3 Coding Assists

Coding assists infer actions from a developer's current context of interaction with the source code, from which the developer can choose and apply one that does what he wants to do. Such features may increase programming productivity. Eclipse distinguishes three kinds of coding assists, content assist (content completion), quick assist, and quick fix. We will discuss more about the design evolution of coding assists in Section 4.3.

2.4 Management and Miscellaneous

The number of annotations and opened editors can be overwhelmingly large. To help a developer manage his interaction with the editor, the Java editor maintains the history of annotations visited and files opened and supports the navigation of these elements. These features are especially useful when a developer works with a large project. Other capabilities like spell checking are also integrated into the editor.

	new features	**enhanced**
1.0	syntax highlighting/S problem navigation/M vertical ruler/M code formatting/A	None
2.0	reconciliation/A quick fix/T quick assist/T content assist/T hovering/A line number/S	None
2.1	editor navigation/M member navigation/A quick outline/A overview ruler/M prominent status/A hyperlink in code/A current line highlighting/S print margin/S	quick fix hovering (sticky note) hovering on ruler
3.0	quick hierarchy/A quick reference/A marking occurrences/A marking method exits/A marking exceptions/A marking overridings/A code folding/A semantic highlighting/A quick diff in rulers/S roll-over hover/S block commenting/S update imports on paste/A spell checking/M	content assist quick assist quick fix quick outline (inherited members) code formatting annotation navigation
3.1	marking methods from supertype/A view all key bindings/M	quick assist quick fix hovering (reference in JavaDocs) semantic highlighting new folding indicators code formatting
3.2	marking jumping targets/A	content assist quick assist quick fix

Table 1: Evolution Profile of Eclipse Java Editor Features. S: Simple, A: Advanced, T: Coding assists, M: Management.

3. FEATURE EVOLUTION

Based on the "new and noteworthy" notes, we tabulated the evolution of the editor features as shown in Table 1. We make the following observations on this table:

1. Features that are based on AST support dominate the feature space. Overall, 21 out of 35 features require AST support. These include three coding assists, and

18 advanced features. Other features, like support for line numbers and quick diff, use only the text model.

2. The majority of enhanced features involve ASTs.

3. While there are enhancements to aspects of individual features, like showing the inherited members in the quick outline view, a noticeable pattern is the continuous addition of sub-features to a feature group, which spreads over multiple releases. For example, the three coding assists were enhanced in almost all releases.

4. Coding assists were added early in 2.0. Most other features that use ASTs were added in 2.1 and subsequent releases. This might be the result of a deliberate prioritization of coding assists, which transform code, over other advanced features that do not.

A feature group may provide very fine-grained support. For example, in 3.1, quick assist supports the manipulation of boolean expressions and string literals. In 3.2, there are approximately fifty sub-features for coding assists and twenty for semantic highlighting.

There can be several reasons for the incremental addition of sub-features. It may be attributed to the lack of development resource or that the sub-features are discovered only at a later release. Lastly, when the underlying model evolves (e.g., from JLS 2 to JLS 3), sub-features may be added for new language constructs such as generics and annotations.

4. DESIGN EVOLUTION

In this section, we examine how the Java editor design evolves with and supports the addition of new features and sub-features to a feature group. The MVC-based modular design laid out in early releases appears to be adequate in supporting the addition of new features. This design has been carefully followed and maintained during the addition of new features, as evidenced by the fact that it is relatively straightforward to locate the respective code for the roles of model, view, and controller in all releases examined. There are also changes to the design of individual features, for reasons like extensibility and reusability. Due to the careful design of interfaces, such design changes seem to be localized and do not unnecessarily impact other design elements. We describe the design changes to coding assists as an example.

4.1 Core Design

As shown in Figure 1, the text editor design follows the model-view-controller model. The model for a text editor consists of two parts, a basic text model (IDocument) and an annotation model (IAnnotationModel). IDocument models the text as a sequence of characters and lines, and supports both presentation and text replacement. Program text is constrained by language specific syntactic and semantic rules. These rules provide the editor with additional elements like compilation errors and warnings, which are mapped back to positions in an IDocument in order to be displayed in the editor. Furthermore, Eclipse generalizes these into the concept of an *annotation*, which is defined by a location in the text model and a description. An *annotation model* manages the annotations associated with a compilation unit. In addition to errors and warnings, users can also define annotations like tasks and bookmarks in program text.

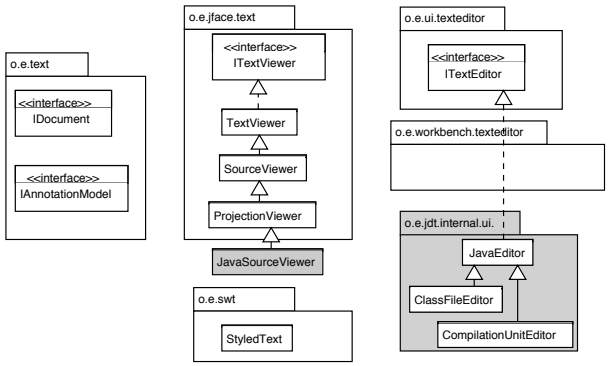

Figure 1: Eclipse Editor Design (MVC)

The view part is defined by the ITextViewer type hierarchy, which composes a StyledText widget and an IDocument. The StyledText widget from the SWT library can display program text with a rich variety of fonts and styles like italic, color, underline, and strike-through.

A controller listens and responds to events produced by the view and the model. The controller usually requires access to both the model and the view to produce a proper response. In Eclipse, the ITextEditor type hierarchy is responsible for configuring the various controllers with an ITextViewer object. Table 2 shows nine editor features that are implemented as controllers.

To support reuse, a general design layer is factored out that all editors inherit and extend. In Figure 1, the general design is colored white and mainly located in 4 plug-ins *o.e.text*, *o.e.jface.text*, *o.e.ui.editors*, and *o.e.workbench.texteditor*. The Java-specific design is colored grey and located in *o.e.jdt.ui*.

4.2 Core Design and Feature Addition

Table 1 shows the features that were added to the Java editor in each release. In order to understand how the core design has supported the addition of these features, we examined the design of a few Java editor features from the source code. Table 2 summarizes these features, the event that triggers each feature and its origin, and the "seed" classes that implement the feature. Seeds are classes that one would look at first to explore a feature. Typically, these classes register listeners to listen to the events. In addition to the seeds, a feature usually has other helper classes and interfaces.

We find that the core design for the editor has been stable since introduced in the beginning of the Eclipse project, and appears to be adequate in accommodating the new features. With the support of the core design, adding a new controller consists of three activities: (1) registering event listeners, (2) modeling the context of interaction, and (3) designing an interface between the listener and the compiler and implementing the feature behind the interface. Typically, steps (1) and (2) are located in the seed classes, and step (3) is implemented in the compiler back end.

editor features	triggering events and origins	seed classes
syntax highlighting	text change/IDocument	PresentationReconciler
reconciliation	text change/IDocument	Reconciler
content assist	key/StyledText	ContentAssistant
quick fix/assist	key/StyledText	QuickAssistAssistant JavaCorrectionAssistant
hovering	mouse/StyledText	AnnotationBarHoverManager AbstractHoverInformation
semantic highlighting	reconciled/reconciliation	SemanticHighlightingManager SemanticHighlightingReconciler
occurrence marking	selection/ITextViewer	ISelectionListenerWithAST
quick outline	action/AbstractTextEditor	InformationPresenter
quick hierarchy	action/AbstractTextEditor	InformationPresenter

Table 2: Features Implemented as Controllers in 3.2

4.3 Design for Extensibility and Reusability

Software designers should design for changes [7], but anticipating changes may take a lot of thought [8]. In this study, we observe that sometimes the 'right' designs may simply emerge in response to external demands for extensibility and reusability. Next, we use the evolution of coding assists to illustrate this point.

The Java editor supports three coding assists, content assist, quick fix, and quick assist. Content assist infers actions using what has been typed in as a prefix. For example, it may return a set of identifiers that share the same prefix. Quick fix offers actions for a compilation problem. For example, if a class does not exist, one action may be to create the class. Quick assist covers all other actions that may be offered. For example, for an *if* with a conditional that does an *instanceof* test, a reasonable action would be to create a local variable and cast and assign the object being tested to it. In response to a user action, an assistant passes the context where the coding assist is requested to the compiler back end. Using the context information, the compiler back end proposes possible transformations that can be applied to the compilation unit in the active editor. Usually, there can be more than one possible transformation. The assistant lets the user choose one from these transformations and applies it to the compilation unit.

Figure 2 depicts the initial design for coding assists in 2.1, and the subsequent changes in 3.0 and 3.2. In 2.1, IContentAssistant and ContentAssistant are responsible for interacting with the user. In response to a user action, a ContentAssistant queries the multiple IContentAssistProcessor associated with it for possible code transformations that can be applied to the current context in the editor. These three types are the main design elements for the content assist feature. However, in 2.1, they were also (mis)used to implement quick fix and quick assist. In particular, JavaCorrectionProcessor was made to implement IContentAssistProcessor in order to register itself to JavaCorrectionAssistant. This design worked, but apparently was not conceptually clean, since quick assist and fix are distinct from content assist, and thus their implementation should be independent of that of content assist.

In 3.0, a decision was made at the Java editor level to support the addition of extensions to quick assist and quick fix, using the extension point mechanism in Eclipse. To enable the extensions, two new interfaces, IQuickAssistProcessor and IQuickFixProcessor, were introduced for QuickAssistProcessor and QuickFixProcessor, respectively. The ICorrectionProcessor in 2.1 was deprecated.

In 3.2, another decision was made to move the Java support for quick assist and fix to the general design layer so that they can be reused by other editors. Support for content assist (IContentAssistant and ContentAssistant) has already been available for reuse. To reuse quick assist and quick fix similarly, IQuickAssistAssistant and IQuickAssistProcessor were added. Now, JavaCorrectionAssistant implements IQuickAssistAssistant, instead of IContentAssistant, via the newly introduced QuickAssistAssistant. The design finally seems to become right because now quick assist and fix become independent of content assist. For example, now it is possible to design a new editor that supports only content assist, but neither quick assist nor quick fix. This is impossible before 3.2 due to the tight coupling of the three.

Despite these design changes to coding assists, they have had no impact on the actual proposals for code transformation, which are modeled by the ICompletionProposal interface. This interface has remained stable since introduced in 2.0. As a result, the implementation for the fifty or so proposals is not impacted by these changes.

An interesting anecdote in Figure 2 is that the two IQuickAssistProcessor in the 3.0 box and the 3.2 box actually represent two different interfaces that are located in two different packages with two different bodies. Thus using the same name is confusing for both future maintainers and users of these APIs. This might have been due to a lack of communication between the two teams who own the two packages.

5. RELATED WORK

Functional Evolution Studies in functional evolution focus on system services that are visible to end users, characterizing the introduction, refinement and enrichment, and displacement of services. Unlike ours, these studies do not focus on the design of the systems studied.

In [1], Antón and Potts developed a theory for functional

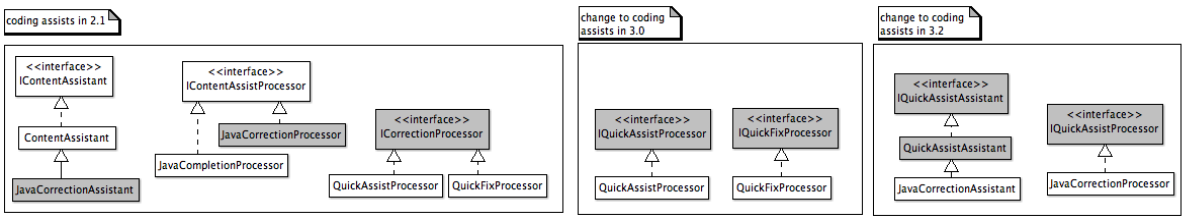

Figure 2: Evolution of Coding Assists

evolution. In this theory, *functional morphology* refers to the overall profile or shape of *benefits* and *burdens* exhibited by a system at a given point during its evolution. *Functional evolution* is then defined as the changes in functional morphology over time. Functional evolution can be characterized as either *gradual* or *saltationist* (meaning that services are introduced in bursts or expansions that separate comparatively stable "*epochs*.") In each epoch, a new set of services may be added (*service cohort*) to the *service baseline* exhibited by the previous epoch, and an existing set of services may be displaced (*displacement cohort*). As a validation of this theory, the authors characterized the evolution of telephony services in a USA city over a 50-year period from 1950 to 1999.

In [6], Hsi and Potts studied the enhancement and evolution of software features over three releases of Microsoft Word (2.0, Word 95, and Word 97). They built a feature profile for each release, and characterized the changes between releases. They observed that user interface features of Word has grown tremendously, and that functional growth, while steady over releases, tends to be focused on one or more areas, while keeping others unchanged. New features tend to be added in a manner loosely coupled with existing core, either as small extensions of existing concepts or as a new set of features that expand the system independently of existing features.

Inferring Knowledge from Evolution of Artifact Size
Studies in this category use size information in terms of modules, programs, or lines of code to infer knowledge about software development, e.g., to characterize the trend in system growth [2], explain evolutionary patterns [4], or identify suspicious subsystems [3]. In this study, we looked into the detailed designs in order to understand how designs have supported and evolved with features.

Lehman and Belady [2] studied the release history of IBM OS 360 and observed that the system is growing sub-linearly. Godfrey and Tu [4] studied the growth patterns of *subsystems* of the Linux kernel during a six-year period to explain why the kernel is exhibiting a super-linear growth rate. They identified that mainly two subsystems, device drivers and architectures, contributed to this super-linear growth, and that the Linux kernel itself is relatively small and exhibits a "steady" growth. Gall et al. studied the release history of a telecommunication switching system to identify a suspicious subsystem that may be subject to restructuring [3]. Grosskurth and Godfrey studied the reference architecture for web browsers [5].

6. SUMMARY

In this paper, we report on a case study on the evolution of the Eclipse Java editor, with a focus on how designs have supported and evolved with features. We find that although there are changes to the design of individual features, architecturally, the editor benefits from the MVC based design laid out in the outset of the project. We also find that AST support is required for more than one half of the editor features, which shows the importance of domain semantics in enabling user-visible features. The Eclipse editor has gone through a long period of evolution, and, thus, necessarily contains more evolution information than we have discussed here. In the future, we plan to look further into other aspects of its evolution, like supporting backward compatibility.

7. REFERENCES

[1] Annie I. Antón and Colin Potts. Functional paleontology: The evolution of user-visible system services. *IEEE Transactions on Software Engineering*, 29(2):151–166, 2003.

[2] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[3] Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *ICSM'97*, pages 160–167.

[4] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *ICSM'2000*, pages 131–140.

[5] Alan Grosskurth and Michael W. Godfrey. A reference architecture for web browsers. In *ICSM'2005*, pages 661–664.

[6] Idris Hsi and Colin Potts. Studying the evolution and enhancement of software features. In *ICSM'2000*, pages 143–151.

[7] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[8] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, 1994.

[9] The Eclipse Foundation. CVS Access to Eclipse Source Code. http://wiki.eclipse.org/index.php/CVS_Howto, last verified: May 1, 2007.

[10] The Eclipse Foundation. Eclipse Binaries. http://www.eclipse.org/downloads, last verified: May 1, 2007.