Bottom-up Design-Evolution Concern Discovery and Analysis

Zhenchang Xing and Eleni Stroulia
Computing Science Department
University of Alberta
Edmonton AB, T6G 2H1, Canada
{xing,stroulia}@cs.ualberta.ca

Abstract

Software system grows in size and complexity as it evolves over time. The fact that object-oriented software is increasingly developed using an evolutionary development process makes the situation even worse. The developers face increasing difficulties in comprehending the system design and its rapid evolution, since the amount of information is overwhelming. Traditional top-down approach to software evolution understanding does not work very well to precisely capture the changes and their underlying motivations. In this paper, we present our bottom-up design-evolution analysis approach, implemented in the JDEvAn tool. The JDEvAn tool has been equipped with a suite of longitudinal and data-mining analysis methods and a set of change-pattern detection queries to automatically recover the interesting core evolution concerns, such as sets of co-evolving classes or instances of refactorings, by aggregating elementary design changes into composite concerns. Given the key participants of an evolution concern, the JDEvAn Viewer allows developers to interactively explore the relevant elements, relations, and their changes over time so that they can incrementally build up their knowledge about what has been changed, how and why. We evaluate the effectiveness of JDEvAn with two case studies on realistic open-source object-oriented software, in the context of which we show how JDEvAn help us capture the completely different rationale for two pairs of seemingly similar evolution concerns.

1 Introduction

Capturing and maintaining the design rationale has been a long-term goal of several different methods developed in support of different activities in the software lifecycle [2,3,5,14,18,22]. These methods aim at recording and maintaining information about why developers have made the decisions they have, so that it can be used to ease further development and improve the quality of future decisions by increasing their consistency with past decisions.

Today, as much of software is developed using some evolutionary lifecycle process, the software design rationale is embedded in the "evolution decisions" of the developers, i.e., the changes they have made to the system from its first version to its current state. On one hand, this indicates that the objective of maintaining software-design rationale becomes that of understanding *what* sequence of changes the system has undergone to reach its current state and *why*. On the other hand, rapid evolutionary development of software system greatly increases software engineers' difficulties in comprehending the system design and its evolution, since the amount of information about the system and its changes grows overwhelmingly.

There has been a substantial amount of work [1,6,8,16] on understanding the overall system-level evolution history of software systems from the information stored in software repositories. Another line of research [7,11,15,32] has focused on the visualization of software-process statistics, source-code metrics, CVS-like deltas and their derivatives, etc. However, little work has been done on design-level evolution concern discovery and analysis. Furthermore, these existing approaches to software evolution understanding generally adopt a top-down method, which means they usually start with an overview of the whole subject system and hope their users to be able to drill down to the interesting parts of the system evolution. They assume a substantial interpretation effort on behalf of their users and they do not scale well for large systems with numerous components.

Several approaches [19,23,33] are available to allow developers to associate code features that are scattered throughout the program text into concerns or aspects and to support for bottom-up exploration by incrementally building and refining concerns. However, these tools focus on helping developers locate and manage scattered concern code. None of them have explored the product of their tools in service of software evolution understanding.

In this paper, we present our bottom-up design-evolution analysis approach, implemented in the JDEvAn tool [37]. In our work on JDEvAn, we have adopted a design-level differencing methodology to capturing and analyzing the design evolution of object-oriented software. Our approach is based on the design-level changes reported by the *UMLDiff* algorithm [31]. *UMLDiff* is a domain-specific differencing algorithm that automatically detects between the subsequent versions of a system: (a) additions, removals, moves, renamings of subsystems, packages, classes, interfaces, fields and methods, (b) changes to their attributes, such data type, visibility, and deprecation-status, and (c) changes of the containment, inheritance, and usage dependencies among these model elements.

Based on the elementary design changes reported by *UMLDiff*, we have developed a suite of longitudinal and data-mining analysis methods [25,26,27] and a set of change-pattern detection queries [30] in order to automatically recognize interesting evolution concerns (e.g., evolution phases, styles and refactorings) in the evolution history of individual system classes, clusters of classes and the system as a. Given the key participant elements and relations of an evolution concern, JDEvAn provides software developers with an intuitive UML-style view – JDEvAn Viewer [38], which supports developers to interactively create, explore and maintain the recovered evolution concerns they are interested in. JDEvAn supports bottom-up comprehension of design-evolution concerns: with JDEvAn, the developers start with the bare minimum amount of information about the automatically recovered core evolution concerns and then selectively explore and refine the relevant elements, relations, and their changes so that they can incrementally build up their knowledge about what has been changed, how and why.

Our case studies demonstrate that our approach is applicable in practice and can effectively focus the developers' attention on the relevant parts of the system evolution. In this paper, we report our

own experience using JDEvAn for examining several seemingly similar evolution concerns and their participants to gain insight into their completely different underlying software-quality motivations. We believe that the very process of recognizing and reflecting upon concrete interesting instances of evolution concerns in the evolutionary history of the software system can help developers better understand the system design and its evolution so that it can be consistently evolved.

The remainder of this paper is structured as follows. Section 2 relates this work to previous research. Section 3 presents an overview of the JDEvAn tool. Section 4 discusses in detail the features of the JDEvAn Viewer that enable developers to explore and maintain the evolution concerns of their interest. Sections 5 evaluate our approach through two case studies and argue for its usefulness in understanding the rationale behind the design evolution of object-oriented software. Finally, concluding remarks are outlined.

2 Related research

There already exists a substantial body of literature on the general "software-evolution understanding" topic. A large subset of work in this area relies on the high-quality history data recorded by version-management systems. In general, these related research efforts are aimed at understanding the overall system-level evolution, such as investigating laws of software evolution [16], calculating code-decay indices to predict fault potential and change effort [6], classifying evolution styles of software systems [1], and populating a release-history database by combining CVS data and bug reports [8]. In contrast, our work focuses on design-level evolution analysis. Furthermore, our approach removes the dependence on such high-quality change data by basing on the automatically recovered design changes between snapshots of system's class model derived from source code.

Another line of related research has focused on the visualization of software-process statistics, source code metrics, static dependence graphs, CVS-like deltas and their derivatives, etc. Eick et al. developed SeeSoft [7] for visualizing code-line deltas and change data such as developer, size, effort, etc. Zimmermann et al. [32] visualizes historical data stored in a CVS archive to help developers recognize the coupling between fine-grained program entities like methods and fields. German and Hindle developed softChange [11] that can be used to visualize information stored in various forms to assist the programmers in understanding how software has evolved to its current state. Lanza [15] describes how to use a simple two-dimensional graph to convey the implicit information of software metrics. These visualization approaches are limited in their applicability, however, due to two important reasons: first, they assume a substantial interpretation effort on behalf of their users and second, they do not scale well: they become unreadable for large systems with numerous components. In contrast, our JDEvAn Viewer adopts a bottom-up method. Its users start with a few key participants involved in a particular evolution concern, which are automatically recovered by various JDEvAn analyses, and they then interactively query, manipulate, and navigate the relevant elements and their relations and changes in order to understand the underlying rationale behind the evolution concern.

There has been some work on reverse engineering interesting evolution concerns by mining software repositories, such as capturing co-evolution of design elements [10,22,32], detect refactorings [4,12,13,20].These approaches are based on either examining change documentation, comparatively analyzing code-line deltas or source-code metrics, or investigating history data achieved in repository. In contrast, JDEvAn's design-evolution analysis relies on the intuitive design-change facts reported *UMLDiff* [31], which enables the identification of a richer set of elementary structural changes and fairly complex evolution patterns, which provides a solid base for us to study the design evolution of a large object-oriented software project [21,26,29]

Several approaches [19,23,33] are available to help developers locate and manage source code that are scattered throughout the program text. Robillard and Murphy developed FEAT tool [19] that supports defining, locating, and analyzing the code implementing one or more concerns. Relo [23] monitors the developer's exploration of code within an IDE and builds automatically the relevant elements and relations into a centralized view. ActiveAspect [33] produces interactive graphical models of program structures affected by aspects in AspectJ. These approaches support bottom-up exploration of code concerns or aspects in the context of program understanding. However, none of them have explored the product of their tools in service of software evolution understanding. In contrast, our JDEvAn tool supports the bottom-up evolution-concern discovery and analysis.

3 Design-evolution analysis with JDEvAn

JDEvAn – "Java Design Evolution and Analysis" – is a tool [37] designed to offer developers comprehensive support for analyzing and understanding the design-evolution history of the software systems on which they work. In this section, we review briefly the JDEvAn capabilities; interested readers are referred to [25,26,27,30,31] for in-depth descriptions. JDEvAn supports:

- *Design-level fact extraction:* JDEvAn's Java fact extractor relies on the Eclipse Java DOM/AST model to recover facts that conceptually belong in the logical UML model of the subject system [17,31].
- *Pair-wise version differencing:* JDEvAn's *UMLDiff* [31] implementation is an Eclipse [34] plugin that compares the extracted logical-design models of two system versions to identify pairs of same-type elements from the two versions that correspond to the "same" conceptual element that may have been left unchanged, renamed, or moved to another part of the system, or somehow modified.
- *Longitudinal design-evolution analysis:* The results of *UMLDiff* can be fed to third party tools to perform several types of analyses – phasic, gamma and optimal sequence matching analysis [26], and association-rule mining [27] – in order to recognize interesting evolution phases and styles in the evolution history of individual system classes, clusters of classes and the system as a whole.
- *Query-based change-pattern detection:* As both design facts and design-change facts are stored in JDEvAn's PostgreSQL relational database, JDEvAn users can posit queries to detect a variety of change patterns of interest to them or invoke the refactoring-detection queries built in JDEvAn [30].

4 The JDEvAn Viewer

Through our experience with using JDEvAn's analysis reports, we recognize the need to provide software developers with an intuitive view of the evolution concerns of potentially large design models and their evolution history. To that end, we have equipped JDEvAn with a UML-style visualization component – the JDEvAn Viewer [38]. The JDEvAn Viewer is implemented as an Eclipse plugin and it relies on the Eclipse GEF (Graphical Editor Framework) [34].

As presented in case studies, when JDEvAn users want to investigate in detail a particular design change, a so-called *core evolution concern*, such as an instance of refactoring or a set of co-evolving classes, they typically start with the design elements involved in the change and their relationships. Then, they can iteratively augment the core evolution concern with the relevant model elements, their relations, and their change status by querying design models and their evolution history and by determining which model elements and relationships returned as part of the queries contribute to the concerns of their interest. In this manner, they incrementally build up their knowledge about what has been changed, how and why. Figure 1 displays such a snapshot, at some point in our investigation process, of two sets of co-evolving classes, which will be discussed in detail in case study section.

Let us now discuss in detail the features of the JDEvAn Viewer, which enable its users to create, manipulate and maintain the evolution concerns.

4.1 Presenting design-evolution concern

As can be seen in Figure 1, JDEvAn Viewer divides the screen into three areas: the main panel visualizes the UML diagram consisting of the concern elements, relations, and their changes, the bottom-left view outlines the same diagram in a tree view or thumbnail display, and the bottom-right properties sheet displays the detailed properties of the selected element or relation.

The Outline view can switch between tree mode and thumbnail mode, whose main purpose is to facilitate the navigation of large diagrams. The tree mode presents model elements and their changes in a containment change tree [27]. The trees are easier to layout and navigate than the diagrams, which makes it easier to locate an element. The JDEvAn Viewer synchronizes its main diagram display and its tree outline so that selecting an element in the outline tree reveals and highlights the corresponding visual part in the main display, and vice verse. The thumbnail outline (not shown in Figure 1) shows the thumbnail display of the main display area, in which the user can drag and move a shadow window to quickly reveal parts of the main diagram.

In JDEvAn Viewer, all the model elements and relations being visualized are selectable from either the main display diagram or the tree outline view. When an element/relation is selected, its detailed model and change information can be inspected in the Properties view with a [Property, Value] table. Different types of elements and relations may have slightly different properties sheet. In Figure 1, for example, the renamed class `ColorBar` is selected. Its corresponding properties sheet lists its entity category, visibility, name, *UMLDiff* status, ID in JDEvAn database, incoming and outgoing relations from and to other elements, and its location and size in the main display area. For those properties that have been reported as changed by *UMLDiff*, the corresponding value columns are shown in the form of "oldvalue → newvalue". For example, the `ColorBar` class was originally named as `HorizontalColorBarAxis`. Therefore, the value of its Name property is "HorizontalColorBarAxis → ColorBar". The row of properties sheet is expandable by clicking the plus sign (if applicable) to the left of a particular row. For instance, by expanding "As source" row, the users can find out the relations originated from the selected element and the related elements at the other end of the relations.

The main diagram of the JDEvAn Viewer displays *part of* the models *UMLDiff* compares and its comparison results in the form of UML class diagram. In the evolution concerns shown in Figure 1, three packages are under investigation, each of which contains one or more classes. The classes declare fields and methods/constructors, which are shown in attribute and operation compartments respectively. The model elements are decorated with the standard Eclipse icons. The model elements may be related to each other with generalization/abstraction relations and/or usage dependencies. Different types of relations are visualized with different line styles and arrow heads.

The *UMLDiff* status of model elements/relations is visualized by coloring the name (identifier for method/constructor) of model elements and their relations, which is defined as follows:

- Black: Matched model elements and relations
- Blue: Newly added model elements and relations
- Red: No longer existing model elements and relations
- Green: Renamed[1] model elements
- Grey/Orange: Move-source and move-target elements respectively
- Light grey: Matched usage dependency with decreasing occurrence
- Dark grey: Matched usage dependency with increasing occurrence

The names of removed elements are struck through. The original name of renamed elements (identifier for method/constructor) is shown with a strikeout line as well. Furthermore, the matched parameters of methods/constructors are initially hidden with "…" placeholder, which can be expanded and collapsed by clicking the "+" or "–" handle of the placeholder. For field/method/parameter type, they are shown in black font, following the corresponding field/method/parameter. If the type changes, the old type is struck through and is followed by the new type. Visibility and modifier(s) are shown as adornments to the icon of the model elements, according to the Eclipse Java model convention. If the visibility and/or modifiers change, they are shown with the original element icon being struck out followed by the new element icon.

In Figure 1, the main diagram view shows three matched packages. The class `HorizontalColorBarAxis` is renamed to `ColorBar`. It no longer implements the interface `ColorBarAxis`, which is removed, and no longer extends the class `HorizontalNumberAxis`, which is removed as well. Instead, it starts extending the matched class `Object`. The renamed class `ColorBar` declares one new field `axis` and one new method `getAxis()`. Its method `doAutoRange()` is removed. Its method `setMaximumAxisValue()` and `setMinimunAxisValue()` are renamed to `setMaximunValue()` and `setMinimunValue()` respectively. However, their parameter lists stay unchanged. The class `ObjectTable` is newly introduced. It becomes the new declaring class of the moved field `rows` and the moved method `getRowCount()`, which are originally declared in its two subclasses `PaintTable`, `StrokeTable` and `ShapeTable` respectively. The matched class `NumberAxis` is no longer abstract. The data type of the renamed field `ContourPlotDemo.zColorBar` changes from the class `NumberAxis` to `ColorBar`.

Finally, the JDEvAn Viewer provides additional information in the form of tooltip pop-ups when the user browses the diagram. In Figure 1, the cursor is pointing to a no longer existing super-call relationship between the renamed method `Color-`

[1] For method/constructor, the renaming may involve the identifier change and/or parameter list change.

`Bar.setMaximumValue(double)` and the matched method `ValueAxis.setMaximumAxisValue(double)`.

4.2 Exploring the neighborhood of a concern

When an element is selected, the set of appropriate handles appears around the selected element, such as those around the selected class `ColorBar` shown in Figure 1, including M – More children, G – Generalization, S – Specialization, I – Incoming usage, O – Outgoing usage, SN – Similar Name (based on regular expression of the words in the element name), ET – Evolve To, and EF – Evolve From. The handles allow the JDEvAn users to query the relevant model elements, relations, and their changes and to interactively include those that most likely contribute to the evolution concerns of their interest. Thus, the model elements and relations that are visualized in a particular diagram may be only a very small subset of all the model elements and relations. For example, in terms of the *replace inheritance with delegation* refactoring shown in Figure 1, the user would most likely be interested in three generalization/abstraction relationships originating from the renamed class `ColorBar`, a few newly added, removed, and renamed field and methods of `ColorBar`, and the class `ContourPlotDemo` in which the class `ColorBar` is used.

Left-clicking on a handle adds to the diagram all relevant elements and relations that the handle is concerned about; right-clicking on a handle pops up a context menu (not shown in Figure 1), which allows the users to selectively add elements and/or relations to the current concern. To facilitate exploration, the entries of the context menu are grouped by *UMLDiff* status and are annotated with the proper icons that represent the *UMLDiff* status associated with the corresponding elements/relations. The handles and context menus keep the diagram as simple and clear as possible.

Since it is developed based on Eclipse GEF (Graphical Editor Framework), the JDEvAn Viewer leverages the GEF facilities to provide Undo/Redo and Zoom-in/Zoom-out. All the modifications to the diagram, such as adding elements and relations into the diagram, removing irrelevant ones, moving and/or resizing elements, bending connections, etc., are undoable and redoable. This enables the users to explore the evolution concerns freely.

4.3 Exploring the evolution trace of a concern

Two special handles – Evolve To (not applicable to removed element) and Evolve From (not applicable to newly added element) – are available to open a new JDEvAn Viewer and present the successor (predecessor) elements and their *UMLDiff* status of the selected element in a given following (previous) version. These two handles enable developers to inspect the entire evolution trace of an evolution concern under investigation, starting at a particular version, such as how an element is introduced in the system, what are their state before refactoring some elements/relations and how do they evolve into these state, what benefits the refactoring brings about, and so on.

4.4 Attaching user comments

The JDEvAn Viewer allows developers to attach one ore more comment(s) to model elements and relations and their changes to record the hard-earned evolution knowledge. For example, in Figure 1, a comment is attached to the generalization/abstraction changes of the class `ColorBar`, its newly added field `ColorBar.axis`, and the field `ContourPlotDemo.zColorBar` where the `ColorBar` is used in order to annotate that these changes are to replace inheritance reuse with object composition. A comment is also attached to the new superclass `ObjectTable` to explain the intention of this extract superclass refactoring.

4.5 Requesting source code

As users investigate the evolution of software system at the design level, JDEvAn maintains in parallel, a mapping between the design-level representation and the source code corresponding to each model element, which can be requested at any time during the investigation. The source code contains useful information such as comments and intra-method structure, which may complement and assist the understanding of the abstract representation. To access the source code, the users simply double-click on a model element being visualized. If the selected element is newly added or removed, JDEvAn brings up the Eclipse Java Editor and highlights the corresponding code fragment. If the model element is mapped (matched, renamed, or moved), JDEvAn brings up the Eclipse Compare Editor or Dialog to show the textual comparison results of the source code of the double-clicked element.

4.6 Persisting design-evolution concern

Focusing on a specific evolution concern in the JDEvAn Viewer and exploring its relevant elements and relations enables a compact and local view of otherwise scattered model elements and relations by collecting them together and by eliding irrelevant (non-concern) elements, relations, and their changes. This localization has been helpful in gaining insight into the evolution history of the subject system. Furthermore, the JDEvAn Viewer enables its users to persist the evolution concerns under investigation into files, which can be reloaded and further examined. As illustrated in the case study section, there are several advantages to documenting hard-earned knowledge about the evolution history of the software system. First of all, the knowledge associated with an evolution concern is much more descriptive than that in the change logs or the release notes. Other users may be able to use the knowledge without needing to perform all of the time-consuming investigation, which might involve false turns and the examination of unrelated elements and relations if they start from scratch. More importantly, a developer performing similar changes, or encountering similar evolution "smells" later, can use the documentation to help make the modification in a more systematic and robust fashion.

5 Case studies

In this section, we use two case studies to demonstrate the effectiveness of our approach for capturing the interesting evolution concerns and understanding their underlying rationale. In particular, we demonstrate, through two pairs of design-evolution concerns from our case studies, how JDEvAn and JDEvAn Viewer facilitate the understanding of the system design and its evolution, which we believe is crucial for maintaining and further evolving the system in a consistent manner.

Using JDEvAn, we analyzed (a) HTMLUnit [36] – a small unit-testing framework for web applications, and (b) JFreeChart [35] – a medium size Java library for drawing charts. HTMLUnit is a small-size open-source software system for unit testing. There are 11 releases in its history from May 22, 2002 to August 23, 2005. JFreeChart has been under development for more than 4 years and there are 31 major releases between the first version 0.5.6, released on December 1 2000, and the last version 1.0.0, released on November 29 2004. The subject systems are in different domains and of different size. They all have been developed for several years with multiple major releases. They all have undergone a substantial number of changes. Their varied design-

evolution history makes them appropriate test beds for the evaluation of our design-evolution analysis approach.

5.1 Different problems but same solution

First, let us discuss how the JDEvAn users, through the support of JDEvAn Viewer, are able to infer the completely different motivations behind the two seemingly similar *extract class* refactorings.

The types of refactorings that JDEvAn is able to automatically detect [30] constitute the basic building blocks for accomplishing many other refactoring tasks, listed in Fowler's refactoring catalog [9]. Table 1 lists some of these refactorings (right column) and their corresponding core refactorings (left column), which can be automatically detected by JDEvAn refactoring queries. The right-column refactorings do not differ substantially from their corresponding core refactorings in terms of the effects they bring on the software entities/relations. In fact, they may even be indistinguishable from one another in terms of *UMLDiff* change facts. The fundamental difference between them lies in their underlying motivation. Although, the motivation behind a particular refactoring cannot be precisely inferred through automatic process, JDEvAn Viewer can facilitate the analysis process.

Table 1. The motivations of core refactorings

Core refactoring	Motivations
Extract method	Replace temp with query Introduce foreign method Decompose conditional Separate query from modifier Parameterize method
Extract class	Replace method with method object Replace data value with object Duplicate observed data Replace type code with class Replace type code with state/strategy Introduce local extension
Extract subclass	Replace type code with subclass Replace conditional with polymorphism

The JDEvAn queries for detecting and classifying refactorings return the concrete instances of a particular type of refactoring and their participants (*which* parts of a system have changed and *how* they have changed). JDEvAn users can then examine the refactoring participants and the relevant model elements, relations, and their changes in the JDEvAn Viewer and draw their own conclusions regarding the motivation and rationale behind the given core refactoring.

Let us examine two particular instances of *extract class* refactoring in the evolution of the HTMLUnit and JFreeChart system from our case studies reported by JDEvAn. Their corresponding participant elements and relations are shown in Figure 2 and Figure 3 respectively. In HTMLUnit, a member class `ResponseEntry` is extracted from the class `FakeWebConnection`, which is used to holds the status and content information of the connection that used to be defined in `FakeWebConnection`. In JFreeChart, a final class `AxisLocation` is extracted, to which the definition of the possible locations of axes is transferred from the interface `AxisConstants`. From the viewpoint of *extract class*, there are no substantial differences between the two instances. They both involve introducing a new class and moving a few fields to it. However, the underlying motivations are completely different, which can be revealed by investigating the relevant model elements, relations, and their changes through JDEvAn Viewer.

In the case of HTMLUnit, the methods that used to modify the moved fields are either removed, such as `setStatus(code:int, message:String)`, or no longer modify the relevant field directly, such as `setContent(content:String)`. Instead, the `setContent(content:String)` starts delegating to the newly added method `setDefaultResponse()`, which receives the content and status information of the connection as parameters and uses them to instantiate the `ResponseEntry object`, which in turn set the values of the corresponding fields. The intention of all these changes is to *replace data value with object.*

On the other hand, in JFreeChart case, the data type of the moved static final fields change from `int` to the newly added class `AxisLocation`. The constructor of the new class `AxisLocation` is private, which means that the `AxisLocation` cannot be instantiated, except for the predefined instances `BOTTOM`, `TOP`, `LEFT`, `RIGHT`. The users of the moved fields, such as `Plot.getOppositeAxisLocation()`, still use them as before, but their corresponding return and/or parameter type changes accordingly. The underlying motivation of this *extract class* is to *replace type code with class*.

5.2 Same problem but different solutions

As software systems evolve over a long time, non-trivial and often unintended relationships among system classes arise. A most interesting such relationship is class co-evolution: because of implicit design dependencies clusters of classes change in "parallel" ways and recognizing such co-evolution is crucial in effectively extending and maintaining the system.

Applying Apriori association-rule mining to class evolution profiles [28] discovers co-evolution patterns among two or more classes, such as the set of classes {`HorizontalColorBarAxis`, `HorizontalLogarithmicColorBarAxis`, `VerticalColorBarAxis`, `VerticalLogarithmicColorBarAxis`} and the set of classes of {`PaintTable`, `StrokeTable`, `ShapeTable`} in JFreeChart case study. Figure 4 and Figure 5 show the similar changes made to the two sets of classes in a particular version of JFreeChart, when we inspect their evolution traces in JDEvAn Viewer. It seems that these classes suffered from the simultaneous development of "parallel inheritance hierarchies". The set of co-evolving classes essentially focuses the developer's attention to specific examples where the refactoring should be applicable, according to textbook [9], which advises informally specific types of refactorings in response to detecting various "smells". But the question then becomes: what is the appropriate refactoring in the given context of a particular "smell"?

In the case of four `?ColorBarAxis` classes, JDEvAn reports that they underwent a refactoring of *replace inheritance with delegation* when the system evolved from the version 0.9.8 to 0.9.9. The bottom-right part of the main diagram area in Figure 1 shows the relevant refactoring participants. The class `HorizontalColorBarAxis` was renamed to `ColorBar`. It stopped extending `HorizontalNumberAxis` and it started extending java.lang.Object. In addition, it started declaring a field `axis` of type `ValueAxis`, the abstract ancestor of all `?NumberAxis` classes. These changes imply that the `ColorBar` was no longer axis, but it can work with any axis objects, conforming to the inter-

faces defined by the `ValueAxis` abstract class. However, in the case of `?Table` classes, the JFreeChart developers applied *extract superclass* and *form template method* refactorings to address the co-evolution smell and reduce the duplicated code. The relevant refactoring participants are shown in the top-left part of Figure 1: a new superclass `ObjectTable` was introduced to hold the common features that were pulled up from the existing `?Table` classes; `?Table` classes were modified to extend `ObjectTable`, overriding the default behavior when necessary.

The choice is essentially between inheritance and composition. Inheritance is a powerful object-oriented design primitive that enables code and design reuse when two or more classes have similar features and capabilities. However, developers often do not notice the commonalities until they have already created some classes, in which case they have to impose the inheritance hierarchy post facto. In version 0.9.9 the JFreeChart developers were faced with the need to introduce six more similar `?Table` classes, such as `FontTable`, `BooleanTable`, `NumberTable` shown in Figure 1. At this point, however, they must have noticed the commonalities between them and the three existing `?Table` classes. Thus, instead of duplicating the existing code, they extracted the `ObjectTable` superclass and made all `?Table` classes extend it, overriding the default behavior when necessary.

In addition to white-box reuse through class inheritance, object-oriented software engineering also enables black-box reuse through object composition, which allows classes to reuse objects in terms of their well-defined interfaces, with limited implementation coupling and increased flexibility. However, sometimes, developers make the "stronger" commitment to white-box reuse when they only need black-box reuse. The introduction of the four `?ColorBarAxis` illustrates a poor choice of class inheritance vs. object composition. Whenever it comes time to change what these classes do, all of them have to be modified in a very similar way to accommodate the change. Furthermore, the inheritance-based reuse also limits the flexibility to draw color bar in other types (may not even exist at the time the `?ColorBarAxis` was introduced) of axes, which may potentially result in the explosion of the class hierarchy and a substantial code duplication if the developers want to deliver the color bar in all possible combinations of the axes. This design was subsequently amended with the modification of the `ColorBar` class that marked the transition from white-box to black-box reuse.

Clearly, inheritance is the simpler choice for the classes `PaintTable, StrokeTable` and `ShapeTable`, since they share interface as well as behavior. In contrast, the color bar feature is better accommodated using composition since it is independent of the other axis features.

We finally annotated these two evolution concerns, including sets of co-evolving classes and the corresponding instances of refactorings, with the above conclusion with JDEvAn Viewer's comment node as shown in Figure 1, and persisted all the relevant diagrams as a useful asset in support of future maintenance and evolution tasks. Such persistent evolution-concerns are much more informative than the textual change logs and release notes. They point out, not only the key elements of the evolution effort and the detailed changes they undergo, but also the relevant elements, relations, their changes, and the hard-earned evolution rationale that motivates the changes. If such evolution concerns were shipped with the new version of a framework or library, they would most likely smooth the learning curve that the application developers experience as they work to migrate their applications to the new version of the framework API. Application developers would be able to learn what has been changed and how exactly based on the evolution concerns, without needing to rely on the terse release notes or start their investigation from the source code. The framework or library developers themselves may also benefit from the documented concerns when performing similar changes or encounter similar smells. For example, when they are faced with co-evolution smells, the developers may compare the situation they have at hand with those documented, which may help them make the choice between *replace inheritance with delegation* and *extract superclass* and decide which one is more desirable.

6 Conclusions

In this paper, we present our approach to bottom-up design-evolution concern discovery and analysis, implemented in the JDEvAn tool. First, facts regarding design-level entities and their relations in each individual version of the system are extracted from the system's code, assumed to be managed in a version-control system. Next, the designs of subsequent system versions are pair-wise compared to determine how the elements and relations have changed from one version to the next. Finally, several longitudinal analysis methods and a suite of queries for detecting change-patterns are applied to recover the interesting design evolution concerns, such as co-evolving classes and refactorings. These automatic analyses aggregate the elementary design changes into bigger concerns at higher-level of abstraction, which are then examined to learn *which* parts of a system have changed and *how* exactly

The more important question is *why* they have changed, since the overall objective of this work has been to reverse engineer the rationale that has driven the evolution of the system to its current state. Inferring the developers' intent can never be accurate. However, as demonstrated in our case studies, our JDEvAn Viewer allows developers to review and explore the detected evolution concerns in the overall context of the system's design and its evolutionary history so that they can draw informed conclusions about the potential intent for the changes. The hard-earned evolution knowledge can then be annotated and persisted in support of consistently evolving the subject system in the future.

References

1. E.J. Barry, C.F. Kemerer and S.A. Slaughter. On the uniformity of software evolution patterns. *Proceedings of the 25th International Conference on Software Engineering*, pp. 106-113, 2003.
2. I.D. Baxter and C.W. Pidgeon. Software change through design maintenance. *Proceedings of International Conference on Software Maintenance*, 1997, pp250-259.
3. J.E. Burge and D.C. Brown. Design rationale for software maintenance. *Proceedings of the 16th International Conference on Automated Software Engineering*, 2001, pp. 433.
4. S. Demeyer, S. Ducasse and O. Nierstrasz. Finding refactorings via change metrics. ACM SIGPLAN notices, 35, 10 (2000), 166-177.
5. A. Dutoit and B. Paech. Rationale management in software engineering. *Handbook on Software Engineeering and Knowledge Engineering, World Scientific*, December 2001.
6. S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 2001, 27(1):1–12.

7. S.G. Eick, J.L. Steffen and E.E. Sumner. SeeSoft—A tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering*, 1992, 18(11):957–968.
8. M. Fischer, M. Pinzger and H. Gall. Populating a release history database from version control and bug tracking systems. *Proceedings of the 19th International Conference on Software Maintenance*, pp. 23-32, September 2003.
9. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
10. H. Gall, K. Hajek and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. *Proceedings of the 14th International Conferences on Software Maintenance*, November 1998.
11. D.M. German and A. Hindle. Visualizing the evolution of software using softChange. *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering*, pp. 336-341, 2004.
12. M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31, 2 (2005), 166-181.
13. C. Gorg and P. Weigerber. Detecting and visualizing refactorings from software archives. *Proceedings of the 13th International Workshop on Program Comprehension*, 2005, pp.205-214.
14. A. Jarczyk, P. Loeffler and I.F. Shipman. Design Rationale for Software Engineering: A Survey. *Proceedings of the 25th Annual IEEE Computer Society Hawaii Conference on System Sciences*, pp. 577-586, January 1992.
15. M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pp. 37-42, 2001.
16. M.M. Lehman and L.A. Belady. Program evolution-processes of software change. *Academic Press*, London, 1985, 538pps.
17. *OMG Unified Modeling Language Specification*, formal/03-03-01, Version 1.5, (2003), http://www.omg.org.
18. F. Pea-Mora and S. Vadhavkar. Augmenting design patterns with design rationale. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 11, Cambridge University Press, pp. 93-108, 1996.
19. M.P. Robillard and G.C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. *Proceedings of the 24th International Conference on Software Engineering*, pages 406-416, May 2002.
20. F.V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. *Proceedings of International Workshop on Principles of software Evolution*, pp. 126–130, September 2003.
21. C. Schofield, B. Tansey, Z. Xing and E. Stroulia. Digging the development dust for refactorings. *Proceedings of the 14th International Conference on Program Comprehension*, 2006.
22. J.S. Shirabad, T.C. Lethbridge and S. Matwin. Supporting software maintenance by mining software update records. *Proceedings of the 17th International Conference on Software Maintenance*, 2001.
23. V. Sinha, D. Karger, R. Miller. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. Visual Languages and Human-Centric Computting (VL/HCC 2006). Sep. 4-8, 2006, Brighton, United Kingdom.
24. I. Sommerville. Software documentation. *In Software Engineering, vol 2: The supporting Processes*. R.H. Thayer and M.I. Christensen (eds), Willey-IEEE Press.
25. Z. Xing and E. Stroulia. Understanding class evolution in object-oriented software. *Proceedings of the 12th International Workshop on Program Comprehension*, pp. 34-43, June 2004.
26. Z. Xing and E. Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31(10):850-868, Oct. 2005.
27. Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. *Proceedings of the 20th International Conference on Automated Software Engineering*, pp. 54-65, 2005.
28. Z. Xing and E. Stroulia. Understanding the evolution and co-evolution of classes in object-oriented systems. *International Journal of Software Engineering and Knowledge Engineering*, Vol. 16, No. 1, 23-52, February 2006.
29. Z. Xing and E. Stroulia. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. *Proceedings. of the 22nd International Conference on Software Maintenance*, Philadelphia, Pennsylvania, September 24-27, 2006.
30. Z. Xing and E. Stroulia. Refactoring Detection based on UMLDiff change-facts Queries. *Proceedings of the 13th Working Conference on Reverse Engineering*, Benevento, Italy, October 23-27, 2006.
31. Z. Xing and E. Stroulia. Differencing logical UML models, The special issue of Automated Software Engineering Journal of selected papers from Automated Software Engineering 2005 conference (Accepted. To appear).
32. T. Zimmermann, S. Diehl, and A. Zeller. How History Justifies System Architecture (or not). *Proceedings of International Workshop on Principles of Software Evolution*, September 2003.
33. ActiveAspect, http://www.cs.ubc.ca/labs/spl/projects/activeaspect/
34. Eclipse, http://www.eclipse.org
35. JFreeChart: http://www.jfree.org/jfreechart/
36. HTMLUnit: http://htmlunit.sourceforge.net/
37. JDEvAn, http://www.cs.ualberta.ca/~xing/jdevan.html
38. JDEvAn Viewer, http://www.cs.ualberta.ca/~xing/jdevanviewer.html

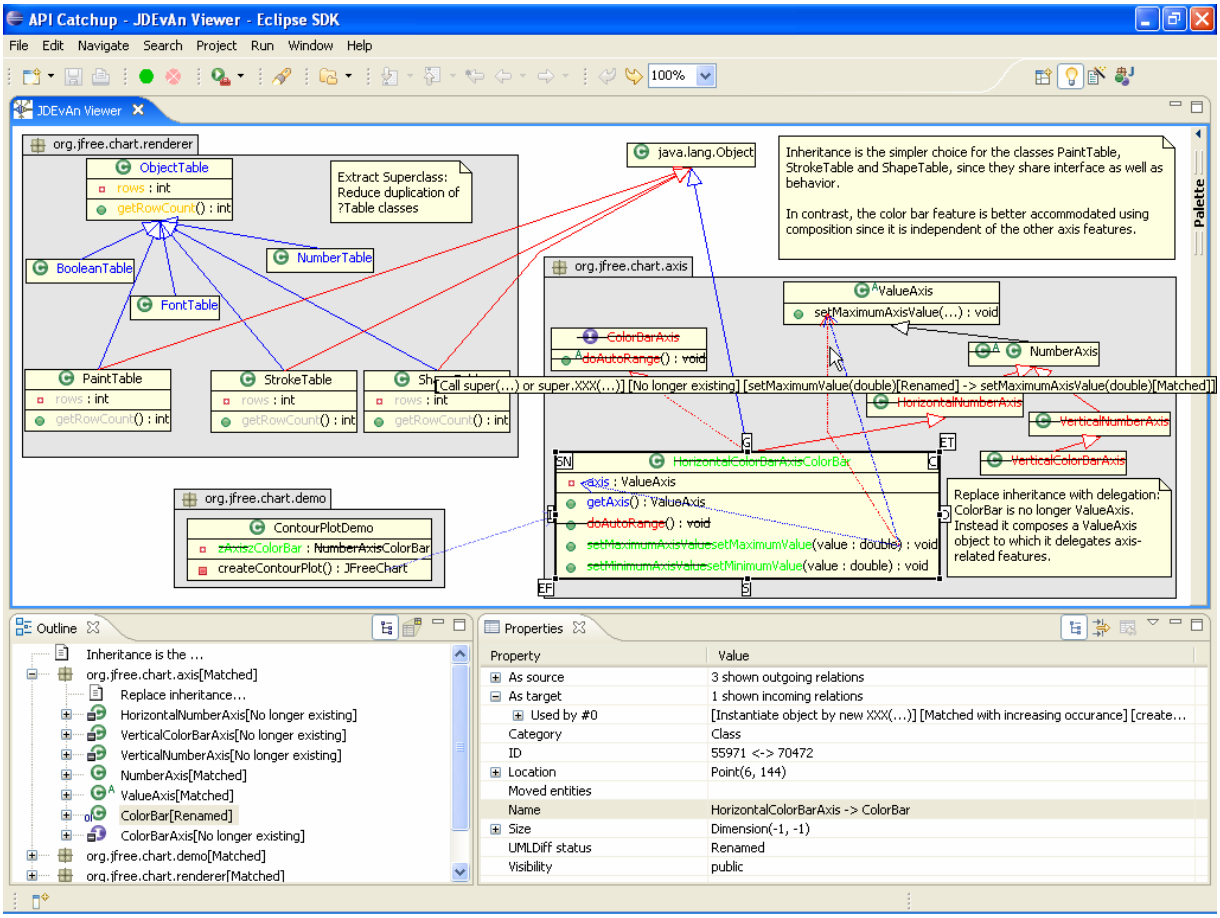

Figure 1. The JDEvAn Viewer

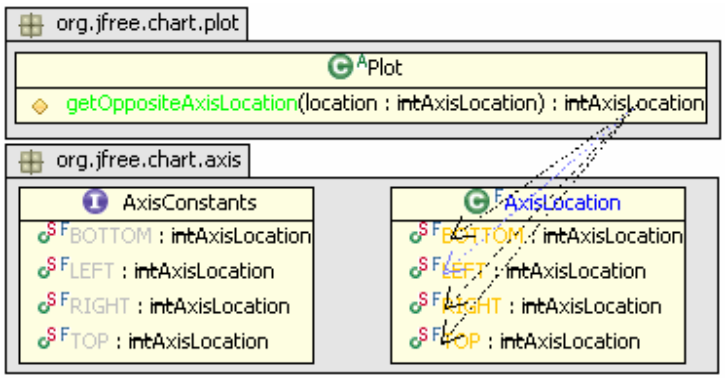

Figure 3. Replace type code with class

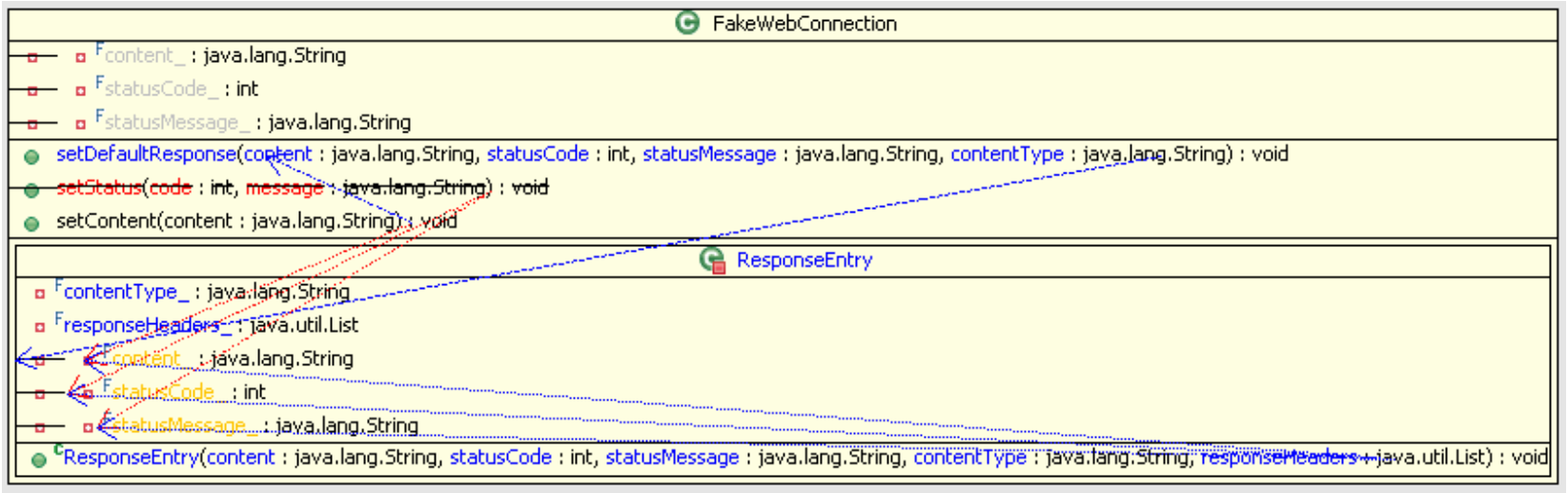

Figure 2. Replace data value with object

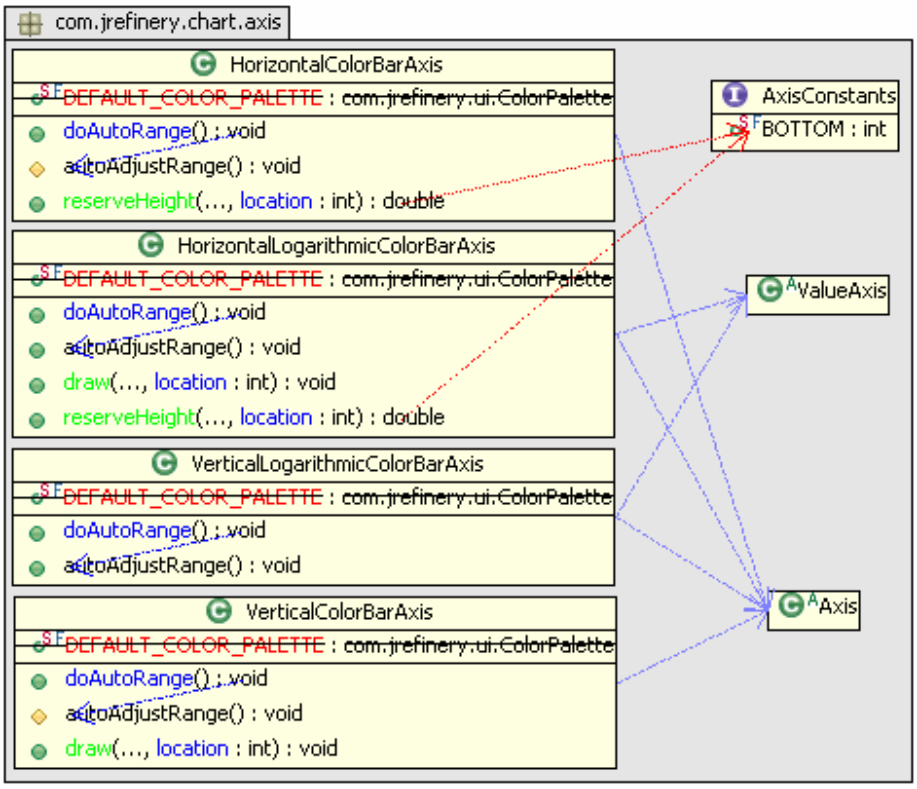

Figure 4. The co-evolution of ?ColorBarAxis classes

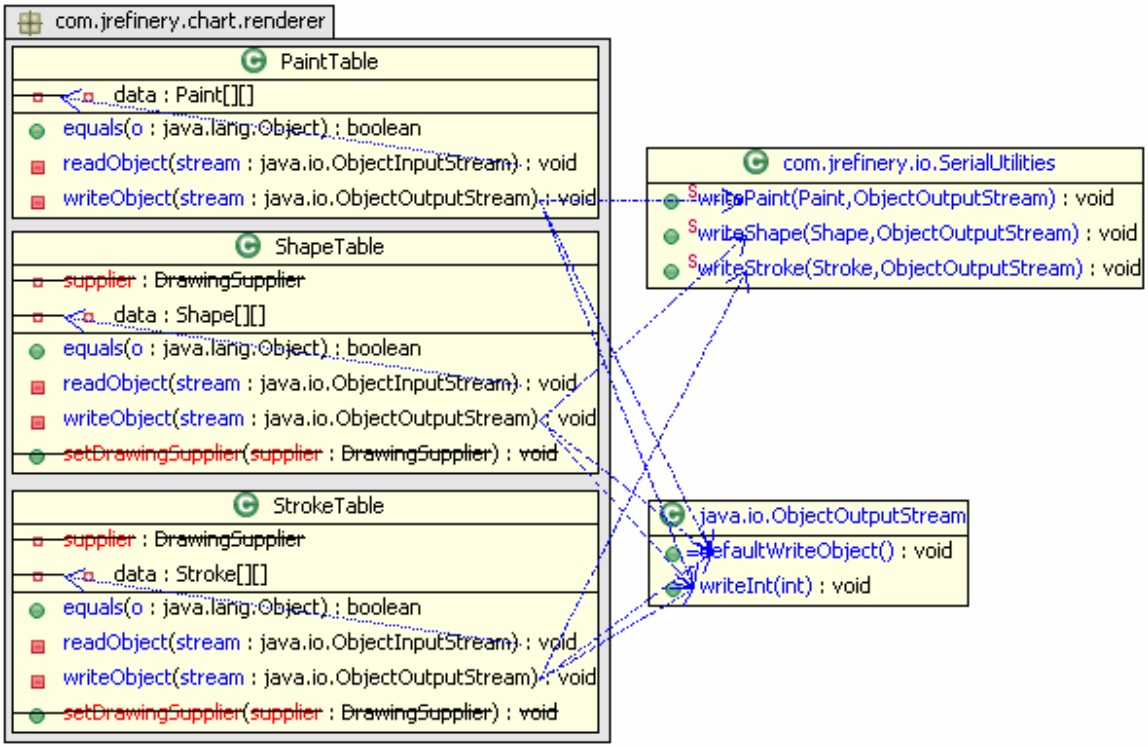

Figure 5. The co-evolution of ?Table classes