Design Mentoring based on Design Evolution Analysis

Zhenchang Xing
Computing Science Department
University of Alberta
Edmonton AB, T6G 2H1, Canada
xing@cs.ualberta.ca

Abstract

The objective of this work is to develop a method for analyzing the evolution of object-oriented systems by comparing, at the logical design level, subsequent system versions.

Research area: design, software evolution.

1 Introduction

Producing a good design is often a daunting task for novice programmers, and so is evolving an existing system in a manner consistent with its design rationale. Today, software is often developed using an evolutionary process, which makes the design task even harder. Textbook software-design knowledge is not sufficient any more; skills and experiences acquired through practice and comprehension of the system’s design-evolution history to date are crucial for dealing with the ubiquitous design evolution.

In the context of software-evolution understanding, the history recorded by version-management systems has been analyzed thoroughly [6], [9], [10], [14] through differencing tools like GNU diff and CVS-like deltas. The major shortcoming of these analyses is that these tools report changes at a very detailed level, i.e., code source listing, since they are mainly intended to aid software developers in merging different revisions of a program rather than to aid them in understanding the evolution of a software system at the design level. Therefore, a lot of useful knowledge about the system evolution can be lost, such as class renamings or method movements.

It is important that software engineers be able to comprehend systems that have undergone substantial structural evolution, such as those resulting from refactorings in the context of object-oriented software system, which frequently involve moving features from one class to another and restructuring data structure or class interface. Recognizing such design-level structural changes is essential to building an accurate picture of system evolution and understanding the evolution of its design rationale.

Source-code metrics [5], [16] and consistent change-log documentation [8], [18] may help to draw inferences regarding the system’s design evolution. However, code-level metrics are frequently too detailed and do not correspond to the developers’ intuitions about the system and more frequently than not, documentation is sparse and inconsistent [3]. Visualization of low-level data, such as CVS-like deltas and code metrics, may help to capture the higher-level connections between moved or renamed elements. However, visualization approaches [7], [13] assume a substantial interpretation effort on behalf of their users and become unreadable for large systems with numerous components.

On the other hand, human experts, such as senior designers, often serve the role of the design mentos, who may supervise and advise junior, less experienced members to help them understand the design of the system and the rationale behind its evolution history so that they can maintain and evolve it consistently.

Unfortunately, the time of such experts is so valuable and they are not always available to consult with. Recommendation systems [1], [24] relieve the need of human experts by using information sources associated with the software development to present relevant software artifacts to the developer’s task on hand. However, the objectives of these systems is mainly to facilitate the developer’s programming tasks, such as locating a component that could be reused, suggesting a potential solution to a particular type of bug, etc.

Clearly, most of previous researches, based on code-line level deltas and metrics or consistently change-log and bug reports, focuses themselves on providing support to either the developer’s programming tasks or the system-wide software management. There is the lack of research at design-level software understanding and mentoring. Thus, there is a need to investigate automatic tools that can assist software engineers in recovering and reasoning about, at the granularity of design level, how and why the structural deltas have occurred in long-lived evolving software systems. The goal of this work is to develop a design-evolution understanding and contextual design-mentoring tool, which, based on recovering the design-level structural changes from source-code versions that reside in version-management system, automatically analyze the evolution of the object-oriented software design in order to provide insights into the rationale of the system evolution to date, which can then enable more informed decisions on future evolution and maintenance activities.

2 JDEvAn: Design-evolution analysis

The objective of our design-evolution analysis method is to recover a model of the design-level structural changes that have occurred in a long-lived evolving software system. JDEvAn (Java Design EVolution and Analysis) is an Eclipse plugin that implements and evaluates this method. It analyzes the design structure of software artifacts and automatically produces reports of interesting design properties they may exhibit or events and trends that they may have suffered during their evolution. It extracts a multi-perspective model of Java software from code versions stored in a version-control system. The crux of its analysis capabilities is its structural differencing algorithm, UMLDiff, which tracks how the logical design of the system changes over time through the UML-aware structural comparison. Based on the design changes produced by UMLDiff, JDEvAn supports querying, navigation, and visualization of evolution knowledge from system-, class-, and structural-change perspectives.

JDEvAn focuses on the logical view of object-oriented Java systems as the first design artifact to analyze. Its primary input is the system’s source code, residing in a versioning system. JDEvAn’s fact extractor recovers a data model of the subject system’s class design. Formally, the meta-data model is a graph,
The model information is stored in a relational database. Derived relations, such as method implementations and overrides, constructor, class usage, etc., are defined based on database views on the ground entity and relation facts. To address the lack of recursive computation capability of the database, which is essential to compute the transitive closure of various relations among entities, Simon’s transitive closure algorithm [19] (its time complexity (worse case) is $O(|V|^2|E|)$) has been implemented as a database server-side extension to pre-compute, at the end of the fact-extraction process, the transitive closure of containment hierarchy, class hierarchy, field access/method call, and class usage.

At the core of JDEvAn lies UMLDiff, a structural differencing algorithm that compares the extracted models of two system versions to identify the structural changes of the software design from one version to the other. UMLDiff is a domain-specific structural-differencing algorithm, aware of the UML semantics. The basic idea for identifying that two entities match is that they have the same or similar contents. We have identified two heuristics for establishing "matching".

- **Identifier similarity:** Similarity of identifiers can serve as the first indicator for matching two entities of the same type. Of course, a developer can remove a class, and then add a new class with the same name and different functionality. However, this case should rather be a rare exception.
- **Relationship similarity:** When an entity is renamed or moved, its relationships to other entities, such as the members it contains, fields it reads/writes, methods it calls or is called by, etc., tend to remain the same or similar. Therefore, by comparing the before and after relationships between two entities, renamings or moves may be inferred. UMLDiff allows its users to define parameters to determine what can be considered as rename or move, such as for example, “more than 60% members of the two entities should match”.

The matching process assumes that software changes are frequently saved back to the versioning system. If many changes are made without saving the intermediate versions back to the repository, the accuracy of UMLDiff will most likely suffer, although some of them may still be recovered by post-querying after the differencing process.

UMLDiff traverses the containment spanning trees of the class models of two versions of a software system and tries to identify corresponding entities based on the heuristics discussed above. The comparison result is represented as a change tree, summarizing the modifications (additions, removals, moves and renames/signature-changes) of the various design entities and their dependencies.

Based on the change trees produced by UMLDiff four different types of automated analyses are then performed.

1. First, using phasic analysis [1], [11], [17] on a sequence of change trees, a high-level evolution history of the system as a whole is produced, in terms of distinct evolution phases and their corresponding styles [23].
2. A similar analysis is also performed on a per-class basis: i.e., for each class, the various phases of its evolution and their types are identified [22].
4. Finally, pattern-matching of the change trees with refactoring-specific change patterns is used to recognize well-understood patterns of structural design changes, such as refactorings [18].

JDEvAn provides a set of visualizations such as change tree, class evolution histogram, and system evolution matrix, to communicate aspects of the design-evolution analyses to its users. The structural changes reported by UMLDiff and the analysis results (including diagrams) produced by JDEvAn are also stored in the relational database. A set of database views, queries, and server-side extensions can then be defined to allow the user to query and navigate the design-evolution knowledge, such as for example instances of refactorings, and so on. JDEvAn also allows its users to define their own rules (in terms of SQL queries) to query its fact base.

The logical design of the system is only one design artifact among several that could be of interest. Another candidate is the structure of its user interface. The second version of JDEvAn currently still under development – has been extended with the capability to reverse engineer a model of the user interface of Java swing applications, to critique it in terms of user-interface guidelines and to analyze its evolution. The user-interface design analysis extension of JDEvAn currently relies on Java GUI Ripper [15] for the extraction of the user-interface model. A GUI differencing algorithm has also been developed, similar to UMLDiff, to report the changes between subsequent version of user-interface designs, such as changes to the length of a menu, or to the order of its items, or to the containment structure of windows.

## 3 Design-evolution mentoring

To a great extent, software design is hard to teach and to learn because it has few "cut and dry" rules. Good design is subjective; there are few precise criteria for determining what is correct or what needs to be improved and their application is contextual. Skilled designers usually have long-term experience designing and can point to examples of past designs, both good and bad.

The basic intuition underlying this work is that experienced designers are able to point out problematic patterns in the design structure of an artifact and questionable events and trends in its evolution. JDEvAn, through its fact extraction capabilities, captures the logical design of object-oriented java software and the structure of its user interface. Furthermore, through its structure differencing capabilities, it analyzes the evolution of these two types of designs.

A set of heuristics, implemented as queries on the model of design-evolution fact base, has been defined to recognize potentially problematic patterns in the results of the above analyses. Associated with each of these queries are potential design modifications that may be applied to remedy the
discovered problems or general advise on how the design process could potentially proceed. Clearly, the final arbitrators of whether or not to follow this advice are the developers themselves. However, we believe that the very process of recognizing and reflecting upon specific interesting designs and design-evolution examples may help developers acquire valuable design experiences.

JDEvAn produces a set of class evolution profiles for each individual system classes, which summarizes the design-structural changes made to each class in each subsequent system version. The class evolution profiles can be analyzed to determine the class evolution types and interdependency between classes [22]. This information can then be used to query for classes with a particular evolution style, such as idle, i.e., a class with no changes over a long sequence of versions, and active, i.e., a class that has been modified in at least, for example 75%, of the versions between versions i and j; or classes with common change behavior.

The question then becomes what is the practice, in the context of the particular project, with these classes. Further queries can be issued to identify similar cases in the past of the same software system that can then be evaluated to form some project-specific mentoring rules. For example, active classes with more than 15 instance variables were often refactored to fan out their features by such refactorings as “Extract Class” and then stopped being active. Or if methods with same or similar names were added to the classes in almost each version during their co-evolution period, the “Form Template Method” or “Extract SuperClass” refactorings have usually been applied afterwards. Such contextual advices could help the software engineers to determine what they should do with the current cases on hand.

As another example, we know, after a sequence of features-expanding versions, the system design may deteriorate and refactoring phases should be performed in order to keep the design coherent. However, in practice, different projects inject in their process refactoring phases with different frequency. The question then becomes, for any given project how frequently should the developers attempt to refactor it?

In its system-wide evolution analysis, JDEvAn uses statistics to classify system-evolution profiles, in terms of five distinct types of change activities: Active, Rapid developing, Restructuring, Slow developing, and Steady-going. This categorization enables then sequential analyses to discern trends in the evolution history of the system [23]. Comparing the trends from various similar projects, one may be able to form an opinion about when to refactor.

Finally, user-interface design is another design task that can benefit from mentoring. There are several sets of user-interface design guidelines, each one consisting of a multitude of specific guidelines on the design of window structures, menu hierarchies and controls. For example, a widely accepted guideline is that the length of a menu should be less than fifteen. We are now in the process of extending JDEvAn with a component aimed to analyzing and critiquing user interfaces and their evolution. Guidelines, such as the one mentioned above, are a particular type of facts that are queried to comment on the user interface of a specific project. In addition, the GUIDiff algorithm, similar to UMLDiff, produces a GUI change tree, which is also critiqued in order to evaluate whether modifications to the user-interface structure are in the right direction or not. For example, increasing the number of menu items from seven to thirteen is flagged as a potential problem, because it becomes suddenly close to the limit.

4 Implementation status

JDEvAn is part of the JRefleX project (http://www.cs.ualberta.ca/~stroulia/JRefleX). It is seamlessly integrated into the Eclipse development environment. As the software system is being developed, it extracts class models from the system source code versions maintained in the version-management system, tracks (through UMLDiff) how the software design changes over time, and enables analyzing, querying and visualization of the design-evolution knowledge from multiple perspectives. At the same time, the JDEvAn analysis results are stored in the JRefleX server-side database for off-line analysis.

To evaluate JDEvAn we first conducted two case studies (reported in [18], [21], [22], [23]): the study of the evolution of a long-term XP project [12], and the study of the collaborative software development process of a few small undergraduate teams. These studies illustrated that all these design-evolution analyses can be potentially used to assist software engineers in their tasks of understanding the evolution of software design and planning future maintenance activities. We are currently working on a new case study on a much more complex software system, Eclipse itself, which is not yet complete.

5 Conclusions and future work

In this paper, we discussed our initial work on design-evolution analysis and design mentoring. To our knowledge, little effort has been spent to compose sets of design-level structural changes and use them to study software evolution and mentor future software maintenance.

We have already collected some promising experiences on analyzing the design evolution of object-orient software using JDEvAn. But the substantial research and case study is still necessary to properly test our hypothesis. We expect to be able to recognize more types of structural change patterns, to analyze the usage changes of classes of different evolution types, or to identify the changes of interdependence of two subsystem, and so on. Moreover, more general to the fact base that stores the design-evolution knowledge, queries will become possible, such as “Find all Classes that extend Class A and override its methods M that start using field F of Class B since Nor 30, 2004”.

Our final goal is a software design-evolution mentor that could advise developers on the desired course of action given a modification request or his own assessment of the current system design, based on learned experiences from past evolution activities, whether mistakes or successes, especially at the design-level. The challenge is to formulate an interesting collection of design mentoring rules and to specify queries to the JDEvAn fact base for recognizing where these rules are applicable.

Acknowledgment

This work was supported by an Eclipse Innovation Grant, by IBM and by CSER the Consortium for Software Engineering Research, and NSERC the National Sciences and Engineering Research Council of Canada.

I am very grateful for my thesis supervisor Professor Eleni Stroulia for all her thoughtful assistance and excellent advice and suggestions.

1 We expect to have the results of this case study for the final camera-ready version of this paper, in case it is accepted.
References


