Detection of High-Level Changes in Evolving Java Software

Timo Kehrer, Pit Pietsch, Hamed Shariat Yazdi, Udo Kelter
Praktische Informatik
Universität Siegen
{kehrer,pietsch,shariatyazdi,kelter}@informatik.uni-siegen.de

Abstract

Software re-engineering is faced with the huge challenge of understanding the structure and behavior of complex programs. In case of legacy systems, understanding the past of a system is often required to understand certain design rationales that help to understand the system itself.

In this paper, we report about our ongoing work on analyzing the evolution of Java programs using techniques from the domain of comparison and versioning of software models. Working on reverse-engineered class diagrams, a model matching algorithm is used to identify differences on the design-level. Based on that, a highly customizable operation detection engine is used to derive the actual changes between revisions. Our experience has shown that the approach is capable of identifying high-level software changes such as refactorings or the introduction of design patterns.

1 Introduction

Understanding the structure and behavior of a software system is an essential prerequisite to software re-engineering. In case of re-engineering of legacy systems, program understanding often emerges as particular challenge. The only reliable information about the software is often the source-code itself. Reverse-engineering techniques can be used to extract information about a system on higher abstraction levels. However, a complete understanding of a system can only be achieved by studying its evolution. Thus, understanding the past of a system is often the key to understand the system itself.

Usually, software repositories provide the necessary data to statically analyze the evolution of a system [1]. However, the unit of versioning in case of generic repositories is typically a file. If one is interested in a more fine-grained structural analysis, much effort must be invested in order to retrieve the structural changes on the system.

In this paper, we report about our ongoing work on analyzing the evolution of Java programs using methods and techniques which originally stem from the research context of comparison and versioning of software models[1].

[1] http://pi.informatik.uni-siegen.de/CVSM

The main objective of our study [5] is to gather statistical information precisely describing the evolution of design models of a system. This information is in turn used by the SiDiff Model Generator[2], a tool to create synthetic test model histories with defined statistical properties [7].

To this end, we have reverse-engineered a set of open-source Java projects, all of them reported in the Helix Software Evolution Data Set[3], to a simple form of class diagrams. The most important elements of this class diagrams are introduced in Section 2. We finally computed the differences between subsequent revisions. Our experience has shown that recently emerged approaches to detect high-level changes in models are principally well-suited to detect software refactorings and other complex restructurings. An overview of our approach to the detection of changes is presented in Section 3. Section 4 concludes the paper and points out directions for future research.

2 Model-based Representation of Java Projects

Assuming a structured representation of Java programs to be available, model comparison algorithms can be applied to source code as well.

In terms of our study presented in [5], we are interested in design-level changes. Hence, we reverse-engineered the source code of each revision of our selected Java software systems into an appropriate design-level model similar to a class diagram. The simplified core of the meta-model is depicted in Figure 1, while the complete meta-model consists of 15 different element types and is presented in [6].

3 Detection of Changes

Having the appropriate class diagrams at hand, a difference between two revisions can be obtained by using model comparison technologies. Initially, a matcher identifies corresponding elements [4]. Based on this matching, a low-level difference can be derived: elements and references not involved in a correspondence are considered to be deleted or created.

Some recently proposed approaches [2, 3] address

[2] http://pi.informatik.uni-siegen.de/qudimo/smg/index.php
[3] http://www.ict.swin.edu.au/research/projects/helix/

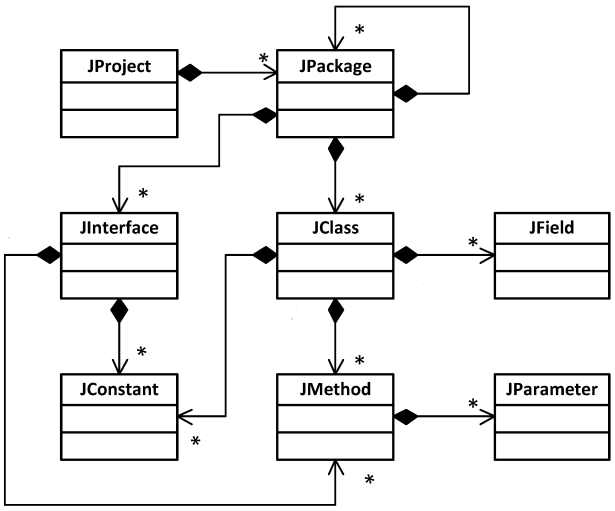

Figure 1: Meta-model for Class Diagrams of Java Source Code - Simplified Core

the problem of grouping low-level changes to sets of changes which represent the effect of complex edit operations. Thus, differences are semantically "lifted" to a higher level of abstraction. An implementation of the approach presented in [2] is available in the SiLift[4] tool environment which we have used in our study.

Obviously, the types of high-level operations that shall be detected by the operation detection engine have to be specified manually. In SiLift, pattern-based model transformation rules implemented in EMF Henshin[5] are used to specify edit operations. The operations defined for our design-level class diagrams can be found at [6]. In sum, we have yet specified a total number of 188 high-level edit operations in class diagrams, including 12 refactorings.

A particular challenge to the comparison of our reverse-engineered class models is the lack of persistent identifiers; matchings must be computed based on similarity heuristics. Thus, possibly "incorrect" matches can lead to false negatives in the recognition of high-level changes, i.e. edit operations which have actually been applied but which were not detected [2]. In our statistical analysis, we calculated the rate of ungrouped low-level changes which was below 0.3%.

4 Conclusion and Future Work

Our experience shows that recent advances in the field of model comparison tools can be effectively adapted to detect high-level changes in evolving Java source code. Besides the recognition of commonly known object-oriented refactorings, the semantic lifter can also be customized to detect other complex changes such as the introduction of design patterns. These might be specific to a certain programming language or even project-related guidelines.

[4]http://pi.informatik.uni-siegen.de/Projekte/SiLift/
[5]http://www.eclipse.org/henshin/

We are convinced that high-level operation detection can yield additional insight into the evolution of a software system which can also be helpful in reverse- or re-engineering scenarios and is worth to be discussed in the respective community. For example, approaches to history analysis [8], which are yet restricted to the tracing of single entities, could be extended to trace sets of related elements, e.g. all elements that are affected by a complex refactoring, change request or a bug fix.

Future Work We are planning to use the KDM meta-model provided by the MoDisco project instead of the proprietary Java AST meta-model currently used. This allows us to analyse the changes on software systems on the abstraction level of source code instead of the design level. An open question in this regard is the scalability of the involved comparison technologies; ASG representations of source code are by nature large models, which result in long computation times of the involved comparison algorithms. Hence, a current research focus lies on how model instances can be partitioned so that they can be processed more efficiently by comparison technologies.

References

[1] D'Ambros M.; Gall H.; Lanza M.; Pinzger M.: Analyzing Software Repositories to Understand Software Evolution; p.37-67 in: Software Evolution, Springer, 2008

[2] Kehrer, T.; Kelter, U.; Taentzer, G.: A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning; p.163-172 in: Proc. ASE 2011; ACM; 2011

[3] Langer, P.; Wimmer, M.; Brosch, P.; Herrmannsdörfer, M.; Seidl, M.; Wieland, K.; Kappel, G.: A Posteriori Operation Detection in Evolving Software Models; Journal of Systems and Software, 86:2; 2012

[4] Kolovos, D.S.; Ruscio, D.D.; Pierantonio, A.; Paige, R.F.: Different Models for Model Matching; p.1-6 in: Proc. CVSM 2009; IEEE; 2009

[5] Shariat Yazdi, H.; Pietsch P.; Kehrer T.; Kelter U.: Statistical Analysis of Changes for Synthesizing Realistic Test Models; in: Software Engineering 2013; Aachen; 2013

[6] Shariat Yazdi, H.; Pietsch P.; Kehrer T.; Kelter U.: Accompanied material and data for the SE2013 paper; http://pi.informatik.uni-siegen.de/qudimo/smg/se2013; 2012

[7] Pietsch P.; Shariat Yazdi, H.; Kelter U.: Controlled Generation of Models with Defined Properties; in: Software Engineering 2012; Berlin; 2012

[8] Wenzel, S.: Unique identification of elements in evolving software models; Software & Systems Modeling; Springer; 2013