

University of Alberta

**Supporting Object-Oriented Evolutionary Development
by Design Evolution Analysis**

by

Zhenchang Xing



A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

Edmonton, Alberta

Spring 2008



Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN: 978-0-494-45630-9
Our file *Notre référence*
ISBN: 978-0-494-45630-9

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

■ ■ ■
Canada

Abstract

Object-oriented software is increasingly developed through evolutionary processes and, as a result, design has become a continuous activity throughout the course of the entire development. Understanding the process of a system's design evolution is critical for understanding the architecture qualities important to the project and for consistently maintaining and further evolving the system. The importance of this research problem is evident in the various types of evolution analyses proposed to date. However, in spite of the large body of research in this area, there has been no systematic method yet for capturing and analyzing changes between subsequent versions of software at the design level. In this work, we have developed *UMLDiff*, a heuristic algorithm for comparing logical models of object-oriented software to recognize additions, removals, matches, moves, renamings of software design elements and changes to their attributes and relations from one version to the next. Based on the design changes reported by *UMLDiff*, we have developed further analyses (a) for recognizing the long-term trends in the evolution of individual classes, clusters of classes and the system as a whole, and (b) for detecting design-change patterns over a set of related design entities. Through our detailed case studies with three real-world software systems of different type and size, we have demonstrated that our design-evolution analyses methods, as implemented in our tool suite, are accurate and practical and that they can be used to support specific development tasks, such as (a) migrating client applications to appropriately use the evolving component-framework APIs, and (b) recommending design and code fragment for addressing problematic evolution patterns.

Acknowledgement

This thesis is the result of five years of work whereby I have been accompanied and supported by many people. It is a pleasure that I have now the opportunity to express my gratitude for all of them.

I would like to thank my supervisor Dr. Eleni Stroulia for her supervision, advice, and guidance throughout the work. Her enthusiasm and integral view on research inspired my growth as an independent researcher. Besides of being an excellent supervisor, Eleni is a good friend to me. I am really glad that I have come to get know her in my life.

I would also like to thank the other members of my Ph.D. committee who monitored my work and took effort in reading and providing me with valuable comments on this thesis: Dr. J. Nelson Amaral, Dr. H. James Hoover, Dr. Paul G. Sorenson, Dr. Marek Reformat and Dr. David Notkin. I thank you all.

I am very thankful to my colleagues, Dr. Kenny Wong, Warren Blanchet, Ying Liu, Curtis Schofield, and Brendan Tansey, for their contributions to this thesis. I would also like to thank Dr. Gail Murphy, Dr. Hausi A. Muller, Dr. Barbara G. Ryder and Dr. Daqing Hou for their advices on this research.

I feel a deep sense of gratitude for my father and mother who formed part of my vision and taught me the good things that really matter in life. I am very grateful for my wife Jinshu, for her love and patience during the Ph.D. period. I am also grateful for my father- and mother-in-law and my sister for their loving support. Without their encouragement and understanding it would have been impossible for me to finish this work.

Finally, I would like to thank everybody who was important to the successful realization of this thesis, as well as expressing my apology that I could not mention personally one by one.

Table of Contents

| | | |
|------------|---|----|
| Chapter 1: | Introduction..... | 1 |
| 1.1 | The research problem..... | 1 |
| 1.2 | Methodology..... | 4 |
| 1.2.1 | <i>UMLDiff</i> | 5 |
| 1.2.2 | Design-evolution analysis..... | 6 |
| 1.2.3 | Design-evolution support..... | 7 |
| 1.3 | Tools..... | 8 |
| 1.3.1 | JDEvAn..... | 8 |
| 1.3.2 | JDEvAn Viewer..... | 9 |
| 1.3.3 | Diff-CatchUp..... | 9 |
| 1.4 | Empirical evaluation..... | 9 |
| 1.5 | Contributions..... | 10 |
| 1.6 | Thesis outline..... | 11 |
| Chapter 2: | Related Work..... | 13 |
| 2.1 | Software differencing..... | 13 |
| 2.2 | Refactoring..... | 19 |
| 2.3 | Software evolution analysis..... | 21 |
| 2.4 | Supporting API evolution..... | 24 |
| 2.5 | Recommending software artifacts..... | 27 |
| Chapter 3: | Model Differencing with <i>UMLDiff</i> | 29 |
| 3.1 | The meta-model..... | 29 |
| 3.2 | UML model reverse engineering in JDEvAn..... | 31 |
| 3.3 | Comparing logical models of object-oriented software..... | 33 |
| 3.3.1 | The running example..... | 33 |
| 3.3.2 | <i>UMLDiff</i> overview..... | 34 |
| 3.3.3 | Similarity metrics..... | 38 |
| 3.3.4 | Mapping model elements..... | 45 |
| 3.3.5 | Mapping relations..... | 58 |
| 3.3.6 | Recognizing behavior redistribution..... | 60 |

| | | |
|------------|--|-----|
| 3.3.7 | Comparing attributes of mapped model elements | 62 |
| 3.4 | Evaluation | 63 |
| 3.4.1 | <i>UMLDiff</i> effectiveness | 63 |
| 3.4.2 | <i>UMLDiff</i> robustness | 68 |
| 3.4.3 | <i>UMLDiff</i> run-time performance | 74 |
| 3.5 | Summary | 76 |
| Chapter 4: | Query-based Change Pattern Detection | 78 |
| 4.1 | Detecting refactorings with change-pattern queries | 79 |
| 4.2 | Refactoring practice: How it is and how it should be supported | 81 |
| 4.2.1 | The empirical assessment of the design evolution of Eclipse..... | 81 |
| 4.2.2 | Analysis of the case-study findings | 96 |
| 4.3 | Summary | 100 |
| Chapter 5: | API-Evolution Support with Diff-CatchUp | 102 |
| 5.1 | An illustrative example | 103 |
| 5.2 | API-evolution catch-up | 104 |
| 5.2.1 | Selecting an API migration problem..... | 105 |
| 5.2.2 | Determining the changes to a broken API..... | 106 |
| 5.2.3 | Proposing replacements for a changed API..... | 107 |
| 5.2.4 | Recommending usage examples of a concerned API | 113 |
| 5.2.5 | Presenting replacement and usage example proposals | 116 |
| 5.3 | Evaluation | 117 |
| 5.3.1 | A usage scenario of the Diff-CatchUp tool..... | 119 |
| 5.3.2 | The effectiveness of Diff-CatchUp..... | 125 |
| 5.4 | Threats to validity | 127 |
| 5.4.1 | API changes without syntactic effects | 127 |
| 5.4.2 | The quality of <i>UMLDiff</i> results..... | 128 |
| 5.4.3 | Availability of “voluntary” migration examples..... | 129 |
| 5.5 | Summary | 131 |
| Chapter 6: | Longitudinal Design-Evolution Analysis | 132 |
| 6.1 | Classifying evolution behavior | 133 |
| 6.2 | Analyzing class-evolution phases and styles | 137 |

| | | |
|------------|--|-----|
| 6.2.1 | Phasic analysis | 138 |
| 6.2.2 | Gamma analysis | 139 |
| 6.2.3 | Optimal matching analysis | 140 |
| 6.3 | Detecting co-evolving classes | 141 |
| 6.4 | Evaluation | 145 |
| 6.4.1 | Class-evolution phases | 145 |
| 6.4.2 | Relative order of evolution phases | 148 |
| 6.4.3 | Class-evolution styles | 151 |
| 6.4.4 | Class co-evolution | 153 |
| 6.5 | Summary | 160 |
| Chapter 7: | Exploring Design Evolution Concerns | 162 |
| 7.1 | The JDEvAn Viewer | 163 |
| 7.1.1 | Presenting design-evolution concern | 163 |
| 7.1.2 | Exploring the neighborhood of a concern | 166 |
| 7.1.3 | Exploring the evolution trace of a concern | 168 |
| 7.1.4 | Attaching user comments | 168 |
| 7.1.5 | Requesting source code | 168 |
| 7.1.6 | Persisting design-evolution concern | 169 |
| 7.2 | Demonstrations | 169 |
| 7.2.1 | Different problems but same solution | 169 |
| 7.2.2 | Same problem but different solutions | 171 |
| 7.3 | Summary | 174 |
| Chapter 8: | Towards Mentoring Object-Oriented Evolutionary Development | 175 |
| 8.1 | Design mentoring | 176 |
| 8.2 | The JFreeChart pilot study | 178 |
| 8.2.1 | Adherence to “first principles” of object-oriented development | 178 |
| 8.2.2 | Refactorings | 182 |
| 8.2.3 | Contextual project-specific hints | 186 |
| 8.3 | Summary | 187 |
| Chapter 9: | Conclusions, Contributions and Future Plan | 189 |
| 9.1 | Contributions | 190 |

| | | |
|--------------|---|-----|
| 9.1.1 | Theoretical contributions | 190 |
| 9.1.2 | Software engineering tools | 192 |
| 9.1.3 | Empirical case studies..... | 194 |
| 9.2 | Future Work..... | 194 |
| 9.2.1 | The software evolution in non-object-oriented paradigms | 195 |
| 9.2.2 | The collaborative production of evolving software systems | 195 |
| 9.2.3 | The knowledge collaboration in software evolution..... | 197 |
| Bibliography | | 199 |
| Appendix A: | UML meta-model..... | 210 |
| Appendix B: | Reverse-engineering Java Software..... | 213 |
| Appendix C: | JFreeChart..... | 215 |
| Appendix D: | HTMLUnit..... | 216 |
| Appendix E: | Eclipse..... | 217 |
| Appendix F: | Refactoring-Detection Queries | 218 |
| F.1 | The simple “standard” refactorings | 218 |
| F.1.1 | Dealing with containment hierarchy..... | 218 |
| F.1.2 | Dealing with generalization..... | 218 |
| F.1.3 | Moving features between objects..... | 219 |
| F.1.4 | Refactoring class internals..... | 219 |
| F.2 | The composite “standard” refactorings..... | 220 |
| F.2.1 | Dealing with containment hierarchy..... | 220 |
| F.2.2 | Dealing with generalization..... | 221 |
| F.2.3 | Moving features between objects..... | 223 |
| F.2.4 | Refactoring class internals..... | 223 |
| Appendix G: | Refactoring Reports | 226 |
| Appendix H: | API Migration Problems | 227 |

List of Tables

| | |
|---|----|
| Table 3-1. The containment hierarchy of UML model elements..... | 35 |
| Table 3-2. The summary of design changes reported by <i>UMLDiff</i> | 37 |
| Table 3-3. The UML relations for computing structure similarity | 40 |
| Table 3-4. The related model-element sets of <i>Customer.statement()</i> ₂₃ and <i>Customer.plainStatement()</i> ₂₇ | 42 |
| Table 3-5. The transitive outgoing usage of <i>Customer.statement()</i> ₂₃ and <i>Customer.plainStatement()</i> ₂₇ | 43 |
| Table 3-6. The partial model-element sets V_{23} and V_{27} | 45 |
| Table 3-7. The sets of renaming candidate pairs | 49 |
| Table 3-8. The initial not-yet-mapped model elements after the match/renaming reorganization steps | 52 |
| Table 3-9. Mapping relations of the renamed [<i>Customer.statement()</i> ₂₃ , <i>Customer.plainStatement()</i> ₂₇] | 59 |
| Table 3-10. Redistribute semantic behavior among operations..... | 61 |
| Table 3-11. <i>UMLDiff</i> results at renaming/move threshold 0.3 | 65 |
| Table 3-12. <i>UMLDiff</i> results at renaming/move threshold 0.35 | 69 |
| Table 3-13. Recognizing renamings with different name-similarity metrics | 72 |
| Table 3-14. Recognizing renamings and moves with and without comment-similarity .. | 73 |
| Table 3-15. Recognizing operation renamings and moves with/without transitive usage similarity..... | 73 |
| Table 3-16. The run-time complexity of <i>UMLDiff</i> in JFreeChart case study..... | 75 |
| Table 3-17. The actual time cost of <i>UMLDiff</i> in JFreeChart case study..... | 76 |
| Table 4-1. Fowler’s “standard” refactorings..... | 80 |
| Table 4-2. Rename program entities..... | 82 |
| Table 4-3. Move program entities..... | 85 |
| Table 4-4. Data-type changes that might be wrapped..... | 90 |
| Table 4-5. Containment-hierarchy refactorings in Eclipse evolution..... | 92 |
| Table 4-6. Inheritance-hierarchy refactoring in Eclipse evolution | 94 |
| Table 4-7. Class-relationship refactoring in Eclipse evolution..... | 94 |

| | |
|--|-----|
| Table 4-8. Class-internal refactorings in Eclipse evolution..... | 96 |
| Table 5-1. The Diff-CatchUp actions for adapting different types of API changes | 107 |
| Table 5-2. Proposing replacements for a “removed” API | 109 |
| Table 5-3. Input parameters for recommending obtain-object usage example..... | 114 |
| Table 5-4. Catch-up the evolution of the three PlotFit-related classes | 120 |
| Table 5-5. Replacement proposals for PlotFit.getFit() with full heuristics checking | 121 |
| Table 5-6. Replacement proposals for PlotFit.getFit() with only “newly used” heuristic. | 121 |
| Table 5-7. The success rate of Diff-CatchUp in the evolution of JFreeChart | 126 |
| Table 5-8. The success rate of Diff-CatchUp in the evolution of HTMLUnit | 126 |
| Table 6-1. The evolution profile of class CategoryPlot..... | 134 |
| Table 6-2. Classifications of change activities | 136 |
| Table 6-3. Transaction database for association-rule mining | 144 |
| Table 6-4. The summary of evolution phases in JFreeChart | 145 |
| Table 6-5. Pair-wise gamma scores of class Axis..... | 149 |
| Table 6-6. Separation and precedence scores of class Axis..... | 149 |
| Table 6-7. VerticalCategoryAxis vs. HorizontalCategoryAxis..... | 153 |
| Table 6-8. The evolution profiles of VerticalCategoryPlot vs. HorizontalCategoryPlot..... | 154 |
| Table 6-9. VerticalBarRenderer vs. HorizontalBarRenderer | 155 |
| Table 6-10. VerticalBarRenderer, VerticalBarRenderer3D, StackedVerticalBarRenderer3D.... | 157 |
| Table 7-1. The handles attached to model elements in JDevAn Viewer | 167 |
| Table 7-2. The motivations of refactorings..... | 170 |
| Table A-1. The UML model elements..... | 210 |
| Table A-2. The UML relations among model elements | 211 |
| Table A-3. The compositions and associations among model elements | 212 |
| Table A-4. <i>UMLDiff</i> -specific tagged values attached to model elements | 212 |
| Table B-1. Mapping Java language constructs to UML model elements | 213 |
| Table B-2. Mapping Java relations to UML metarelations..... | 213 |
| Table B-3. Mapping Java modifiers to the attributes of UML metaclasses..... | 214 |
| Table B-4. Mapping Java language features to <i>UMLDiff</i> -specific tagged values | 214 |
| Table C-1. The number of model element and relation facts of JFreeChart..... | 215 |
| Table C-2. The summary of <i>UMLDiff</i> changes in JFreeChart evolution..... | 215 |

| | |
|---|-----|
| Table D-1. The number of model element and relation facts of HTMLUnit | 216 |
| Table D-2. The summary of <i>UMLDiff</i> changes in HTMLUnit evolution..... | 216 |
| Table E-1. The number of model element facts of Eclipse..... | 217 |
| Table E-2. The number of relation facts of Eclipse..... | 217 |
| Table E-3. The summary of <i>UMLDiff</i> changes in Eclipse evolution..... | 217 |
| Table G-1. The refactorings in the evolution of HTMLUnit, JFreeChart and Eclipse... | 226 |
| Table H-1. The summary of API migration problems that Diff-CatchUp is able to handle | 227 |

List of Figures

| | |
|---|-----|
| Figure 1-1. Methodology overview | 4 |
| Figure 2-1. Eclipse text compare | 13 |
| Figure 2-2. The partial AST of class PlainStatement | 14 |
| Figure 2-3. The partial AST of changed local variable declaration..... | 15 |
| Figure 2-4. XML-differencing XMI representation of UML models..... | 16 |
| Figure 3-1. An example of UML model that <i>UMLDiff</i> compares | 30 |
| Figure 3-2. The impact of the user-specific renaming and move thresholds..... | 70 |
| Figure 5-1. Diff-CatchUp perspective in Eclipse | 122 |
| Figure 5-2. Explore PlotFit.getFit() and its replacing APIs with JDEvAn Viewer | 124 |
| Figure 5-3. Code differences demonstrating how to replace getFit() with createMovingAverage(...) | 124 |
| Figure 6-1. Refactoring frequency | 150 |
| Figure 7-1. A screenshot of JDEvAn Viewer | 163 |

Chapter 1: Introduction

The work in this thesis aims to address the problem of supporting object-oriented evolutionary development through capturing and analyzing the design-evolution history of object-oriented software. As object-oriented software is increasingly developed through evolutionary processes, software-engineering research is becoming increasingly aware of the need to analyze the by-products of these processes – i.e., software versions in repositories, their associated documentation, change requests, etc. – in order to better understand and support them. The research presented in this thesis is focusing exactly on this issue.

In this introductory chapter, we formulate the specific research questions examined by the thesis, place our work in the context of related research, outline the methodology we have adopted for our work, and review the tools we have developed to support this methodology and the experimentation we have conducted with these tools to evaluate their effectiveness. Finally, we identify the important contributions of this work to the field as a whole, and conclude by discussing the organization of the rest of this document.

1.1 The research problem

Object-oriented software is increasingly developed using evolutionary development processes. Design is no longer a one-time activity; systems, in addition to being incrementally designed to fulfill their evolving requirements, have their design continuously improved through behavior-preserving restructurings, as refactoring has become one of the most important core practices in object-oriented development.

As a result, at any point in time, the system design is the product of a sequence of design-evolution decisions, a fact that should be taken into account in subsequent development and maintenance tasks. Some tasks may require that the developer understands the evolution trajectory, through which the system design has reached its current state and the practices that the system-development team has adopted in the past. For example, understanding the types of code restructurings that have been applied in the past gives the developers insights regarding the design qualities that are considered important for the project, so that they can keep evolving the system consistently.

Managers may also be interested in high-level evolution patterns characteristic of desired process qualities or symptomatic of problems so that they can better manage the project and predict the evolution of other projects in the future.

Continuous design improvement also poses serious problems for building applications that reuse evolving component frameworks. Stable interfaces to a reusable component framework isolate the client application from changes to those components. However, the new component's versions change their application programming interfaces (APIs), sometimes dramatically, which implies a need for client applications to adapt. The fundamental challenge lies in the fact that these applications and their underlying component frameworks are subject to two independent, asynchronous, and potentially conflicting evolution processes. To migrate to a new component's version, the client-application developers have to understand the nature of component's API changes, their plausible replacements and how to use them.

All of these tasks essentially require answers to the same set of three basic and related research questions:

Research Question 1. How can one precisely and accurately recognize changes to a system's design from one version to the next?

Research Question 2. How can this information be further analyzed to gain insight in the rationale behind the system's evolutionary development?

Research Question 3. How can the identified changes and the analysis results be effectively used to support further evolving the system?

There has been some research towards recognizing software changes. Lexical differencing tools [118] view software programs as text documents; they report changes at the lexical level in terms of code-line deltas. Such tools are frequently used by developers to merge revisions of the system source code. The comparative analysis of source-code metrics [20] may also help to infer moves of program entities. Unfortunately, source-code metrics do not provide much intuition regarding whether program entities has been changed and how.

There also exist differencing techniques that make use of other types of program representations, such as Abstract Syntax Tree (AST) [82,107], program dependency graph [47], control flow graph [3,45], and XMI (XML Metadata Interchange) format of

UML model [128]. However, these program representations are designed for purposes other than understanding and supporting the evolutionary development of software system and their results are usually too close to the software representation they adopt instead of the design representations intuitive to the developers. Although recent research on mining software repositories has shown increasing interests in detecting refactorings [23,52,53,100], there still lacks systematic design-evolution analysis of long-lived object-oriented software about what has changed, how exactly, and why.

Consistently maintained modification request, change logs, and bug reports, if exist, are reliable source of information as to what has been changed and what is the rationale behind the change. They have been used to analyze and delay code decay [30], predict error-prone parts [39] of the system and infer change effort [38], understand how fast the system changes and how stable the system is [10,35,41], reveal co-evolution of design elements [9,33,87,110], define system-level evolutionary patterns [7,31,58], and infer the nature of individual developer's contributions to the system evolution [62]. Unfortunately, more frequently than not, such change documentation is vague, inconsistent or does not even exist [13].

Although there has been a substantial amount of research in the general area of understanding the evolution, i.e., "past", of software, there has been much less work on utilizing this understanding to "advising for the future". Several recommendation systems have been proposed to locate a component that could be reused [73,109], suggest a potential solution to a particular type of bug [17], recommend people who might have some expertise on a given problem [64], and present documents and code snippets similar to the one currently being edited [44,78,108]. However, these recommendation systems do not provide much in-depth analysis of the recommended software artifacts and their evolution history; the developers take the burden to draw analogies towards solving their current problem.

In the context of asynchronous API evolution in reuse-based software development, several practices [15,42,50,119,121] have emerged to support the specification of changes that may impact reuse and their consistent propagation. They rely on additional, and potentially substantial, information provided by the component-framework developers that document the changes and advise on how to adapt them. However, it is

seldom the case that the change documentation and scripts provided with a large framework are sufficient for a client-application developer to effectively migrate to the changed APIs, in spite of a substantial effort to write and maintain the change documentation or scripts on the part of component-framework developer. All too often, application developers become lost when trying to reuse a changed API, unsure of how to make progress on a migration problem.

1.2 Methodology

The objective of this work is to develop a methodology for building an accurate understanding of the design evolution of object-oriented software systems and supporting their subsequent development and maintenance activities based on this understanding. Figure 1-1 depicts the overall process of this work. The three phases of the process are discussed in the following section.

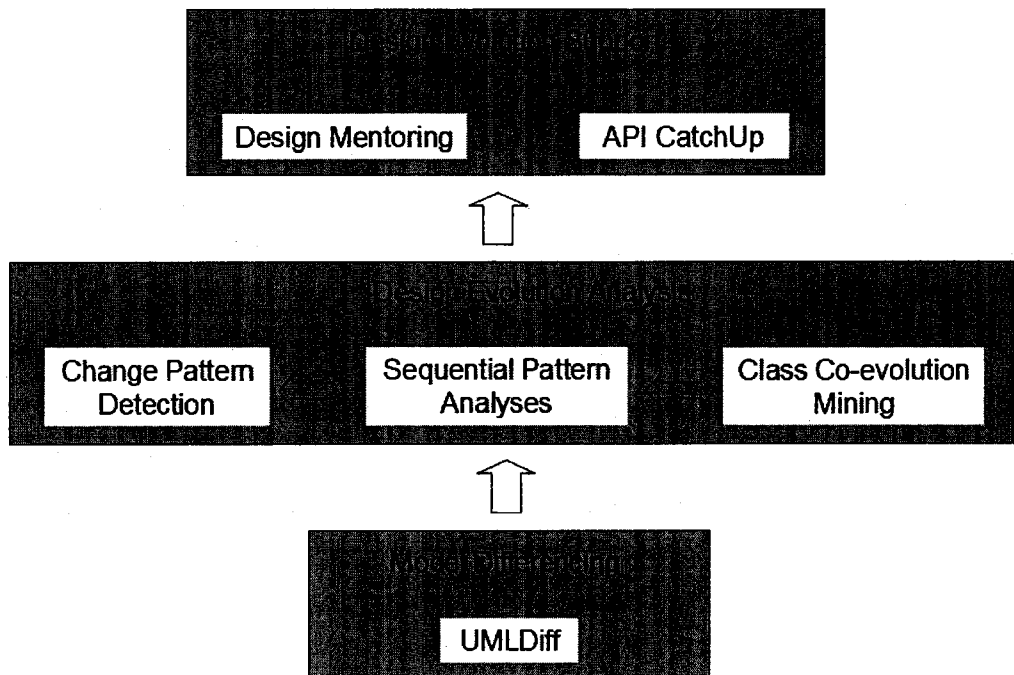


Figure 1-1. Methodology overview

1.2.1 *UMLDiff*

The input to our methodology is the logical model [56] of object-oriented software systems, which concerns classes, the information they may own, the services they can deliver, and the associations and relative organization among them. The underlying meta-model is defined according to the semantics of the UML (Unified Modeling Language) model [69]. As per adopted meta-model, a software system is modeled as a directed graph $G(V, E)$, where the vertex set V contains model elements and the edge set E contains relations among them. Note that this methodology has so far been only applied to logical models reverse-engineered from Java source code in this thesis. However, by adopting the semantics of the UML model as the meta-model underlying its input representations, it is readily applicable to models produced in up-front design, or models reverse-engineered from other object-oriented programming language, or a mix of them.

Given two versions, “before” and “after”, of the logical model of an object-oriented software system and their corresponding graphs $G_{before}(V_{before}, E_{before})$ and $G_{after}(V_{after}, E_{after})$, we developed the *UMLDiff* algorithm for automatically detecting the design changes that the system has gone through, as it evolved from one version to the next. *UMLDiff* traverses in parallel the spanning trees of the containment subgraphs of the two compared models, descending from one type of model elements to its children types. As it does so, it recognizes that a model element e_1 in the “before” version and an element e_2 of the same type in the “after” version are the “same”, i.e., they correspond to the same conceptual model element, based on their *lexical-* and *structure-similarity*. Once *UMLDiff* has completed mapping the vertex (model element) sets V_{before} and V_{after} , it proceeds to map the edge (relation) sets E_{before} and E_{after} , by comparing the relations of all pairs of model elements. Next, *UMLDiff* detects the redistribution of the semantic behavior among operations, in terms of usage dependency changes, and finally computes the changes to the attributes of all pairs of mapped model elements. *UMLDiff* produces as output a set of design-change facts reporting the various types of design changes it has discovered when comparing the two models, i.e., additions, removals, moves, and renamings of subsystems, packages, classes, interfaces, attributes and operations, and changes to the attributes and relations of these model elements.

1.2.2 Design-evolution analysis

UMLDiff is at the core of our methodology; based on the design-change facts it produces, we have developed a suite of analyses to study the design-evolution history of object-oriented systems.

First, a set of queries have been defined to elicit complex design-change patterns, as compositions of elementary design-change facts, such as refactorings [32], aimed towards improving the design of the system. The instances of design-change patterns are reported in terms of their particular types and their participant model elements and relations.

Secondly, for an evolving software system with N successive versions, *UMLDiff* is applied N times to recognize the design differences between the $(I+1)^{th}$ and I^{th} versions, where $0 \leq I < N$ (supposing there is a virtual version 0 with no entities), resulting in an quantitative trail of the design changes that the system classes have suffered throughout their evolutionary lifecycle, i.e., in which version it was created, how many signature-changes it underwent and how many of its member elements were newly added, removed, moved, signature-changed in a particular version, and, possible, in which version it was deleted. The quantitative report of *UMLDiff* design changes is then discretized to produce a qualitative record of the categorical volatility of each individual class, i.e., discrete class-evolution profile, throughout the system's history. Finally, four types of longitudinal analyses – phasic analysis [84], gamma analysis [75], optimal matching analysis [1], and association rule mining [2] – are applied to the class-evolution profiles to recover a high-level abstraction of distinct evolution phases and their corresponding styles and to identify class clusters with similar evolution trajectories.

The design-changes reported by *UMLDiff* and the subsequent design-evolution analyses reveal which parts of a system have changed and how they have changed. However, the motivation behind the changes, i.e., why they have changed, cannot be precisely inferred through automatic process. To help developers infer the potential intent for the changes, we developed an interactive visualization component, which enables a compact and local view of otherwise scattered model elements and relations and their changes 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 rationale behind the system evolution.

1.2.3 Design-evolution support

UMLDiff, the subsequent analyses it enables, and the interactive exploration of evolution concerns build an accurate picture of the design evolution of software system and its components. We have exploited two areas of supporting the evolutionary development of object-oriented software systems based on this evolution knowledge.

First, we investigated the feasibility of mentoring object-oriented software design and its evolution. A set of queries have been defined to recognize potentially problematic patterns in the logical model of the system and its evolution history, which violate the high-level object-oriented design principles and/or best practices, such as design patterns and refactorings, or the development styles previously adopted by the system. Associated with each of these queries are the design changes that have been applied to remedy the similar problems in the past of the system development. The very process of recognizing and reflecting upon the opportunities for specific design changes aimed towards improving the system design may help developers draw informed decisions on solving their current similar problems.

Secondly, we tackled the asynchronous API evolution problem in the context of reused-based software development. The API changes are automatically recovered with the *UMLDiff* algorithm, given the old and new versions of a component framework. When a API migration problem – which is reported by the compiler when building the client application with the new component-framework version – is selected, a heuristic process searches the logical model and API-change facts of the evolved component framework to formulate plausible answers to the three questions facing the client-application developers: (a) “what changes have been made to the existing component APIs?”, (b) “what are the plausible replacements of those APIs in the new version of the component framework?”, and (3) “are there any examples on how exactly these replacements are supposed to be used?”. Finally, a set of replacement and usage example proposals are formulated and presented to the client-application developers, which help them migrate their applications to appropriately use the evolved APIs.

1.3 Tools

This methodology has been implemented in three related software-engineering tools: JDEvAn (Java Design Evolution Analysis), JDEvAn Viewer, and Diff-CatchUp. We chose on purpose to implement our methodology on Eclipse, a popular Java development IDE, so that it can be tightly integrated within the development environment, and thus enable investigating the design-change patterns of object-oriented software evolution, exploring the underlying motivations behind them, and supporting future development and maintenance activities.

1.3.1 JDEvAn

JDEvAn supports the design-evolution analysis of Java software systems. Its main features include:

- (a) a component for reverse-engineering logical model from the system's code;
- (b) an implementation of *UMLDiff* for recovering the elementary design changes from one version of the logical model to the next;
- (c) support for inspecting the results of *UMLDiff* and also editing them in order to correct erroneously identified and missed changes;
- (d) a component for detecting various types of simple and composite refactorings;
- (e) a component for producing discrete class-evolution profiles, which are then used by third-party tools *WinPhaser* [43] and *Weka toolkit* [130] for sequential pattern analysis and detecting co-evolution rules; and
- (f) a component for detecting potentially problematic patterns, events, trends in the logical model and its evolution history, where some design improvement may be applied.

JDEvAn's front-end is an Eclipse plugin. Its backend repository is a PostgreSQL relational database, which stores all the logical model facts, the *UMLDiff* change facts, and the analysis results of the subject system.

1.3.2 JDEvAn Viewer

JDEvAn Viewer provides software developers with an UML-style diagram, which supports developers to: inspect the logical models of two compared versions and the design-changes between them; explore the neighbourhood and the evolution trace of a selected model element; attach comments to model elements, relations and their changes to record the hard-earned evolution knowledge; request source code of a selected element; and persist the evolution concerns under investigation into files. JDEvAn Viewer has been implemented as an Eclipse plugin and it relies on Eclipse GEF (Graphical Editor Framework) [117]. It leverages the GEF facilities to provide Undo/Redo and Zoom-in/Zoom-out features.

1.3.3 Diff-CatchUp

Diff-CatchUp tackles the real problem of unstable APIs of component frameworks. The Diff-CatchUp front-end, an Eclipse plugin, allows the client-application developer to highlight the code fragments or compilation errors/warnings he wishes to update. In response, it identifies the model element of the component API involved in a selected migration problem and displays the replacement and usage-example proposals for further exploration with the JDEvAn Viewer. Its server hosts a JDEvAn repository regarding the logical-model and API-change facts of the evolving component framework, which is populated with the JDEvAn tool before Diff-CatchUp can be used. Diff-CatchUp searches the JDEvAn repository for the changes to the component APIs, the plausible replacements and their potential usage examples.

1.4 Empirical evaluation

We used three Java software systems, HTMLUnit [123], JFreeChart [126], and Eclipse [116], to evaluate our methodology and the tools that implement it.

HTMLUnit is a small-size open-source software system for unit testing. We use it in evaluation of the effectiveness of *UMLDiff* algorithm, refactoring-detection queries, JDEvAn Viewer, and Diff-CatchUp approach. We use 11 releases in its history from May 22, 2002 to August 23, 2005.

JFreeChart is a medium-size open-source Java class library for generating various types of charts. We use it as the subject system of extensive case study to evaluate our work, from *UMLDiff* algorithm to refactoring-detection queries, longitudinal design-evolution analyses, JDevAn Viewer, design mentoring, and finally Diff-CatchUp. JFreeChart has been developed for more than 6 years. We used in our study 31 major releases between the first version 0.5.6, released on December 1 2000, and the version 1.0.0, released on November 29 2004. It is a substantial and realistic software system and, at the same time, it is of a manageable size, possible to inspect manually to establish the ground truth for evaluating our work.

Eclipse is a large-scale industrial framework that has been under development for about five years. In this work, Eclipse is mainly used to investigate the refactoring practice in the evolution of object-oriented software system and how it should be support. At the time of our study, Eclipse has nine releases between the first official version 2.0, released on June 27 2002, and the latest version 3.1, which was released on June 27 2005. In particular, we chose to compare three pairs of major releases 2.0 and 2.1, 2.1.3 and 3.0, and 3.0.2 and 3.1, because there were substantial changes between them. According to their associated documentation, the remaining versions, 2.1.1, 2.1.2, and 3.0.1, included mostly bug fixes and minor modifications and we ignored them in this case study. Eclipse consists of three subprojects and in our study. We have focused on the JDT subproject, which defines about half of the classes and interfaces of the whole Eclipse platform.

1.5 Contributions

This research has resulted in three different types of contributions:

1. **Theoretical contributions:** We developed a model-differencing based methodology to capture and analyze the design evolution of object-oriented software systems and support their evolutionary development, including pair-wise model differencing with *UMLDiff*, the query-based detection of refactorings, longitudinal design-evolution analyses, mentoring object-oriented evolutionary development, and API-evolution support in reuse-based software development.
2. **System Building:** To support this methodology, we built three software engineering tools. These tools implemented the whole work described in this thesis. They support

the design-evolution analysis of Java software systems and the migration of client applications reusing the evolved component framework

3. **Empirical case studies:** We conducted three empirical case studies in order to refine and evaluate our methodology and the tools that implement it. These studies have demonstrated that our approach is applicable and effective in practice. Furthermore, we believe the empirical results will be useful to other researchers in the area.

1.6 Thesis outline

The rest of this document is organized as follows. Chapter 2 reviews five research areas related to this work, i.e., differencing software artefacts, understanding software evolution, refactoring practices, supporting API evolution of software systems and recommending reusable or similar software artefacts. Chapter 3 describes the underlying meta-model assumed by the *UMLDiff* algorithm and the process by which these models are extracted from Java software. It discusses in detail the *UMLDiff* algorithm and reports on our evaluation of its runtime performance, effectiveness and robustness. Chapter 4 presents our approach to detecting refactorings as compositions of the design changes reported by *UMLDiff*, with which we investigated how refactoring is practiced in the evolution of Eclipse and how it should be supported in general. Chapter 5 discusses our approach to supporting the migration of client applications to appropriately use the evolved APIs of their underlying component frameworks, based on the API changes reported by *UMLDiff*, given the old and new versions of the component frameworks. Chapter 6 discusses a suite of automated design-evolution analyses, based on *UMLDiff* change reports, to recover distinct evolution phases and their corresponding evolution styles and to identify class clusters with similar evolution trajectories. Chapter 7 introduces our visualization component and its features, which supports the interactive exploration of the logical models *UMLDiff* compares, the design changes reported by *UMLDiff*, the detected refactorings, and the results of design-evolution analyses. Chapter 8 discusses the feasibility of design mentoring enabled by *UMLDiff*-based design-evolution analysis. In particular, it discusses its ability to uncover design changes aimed towards improving the system design and detect the opportunities for such changes.

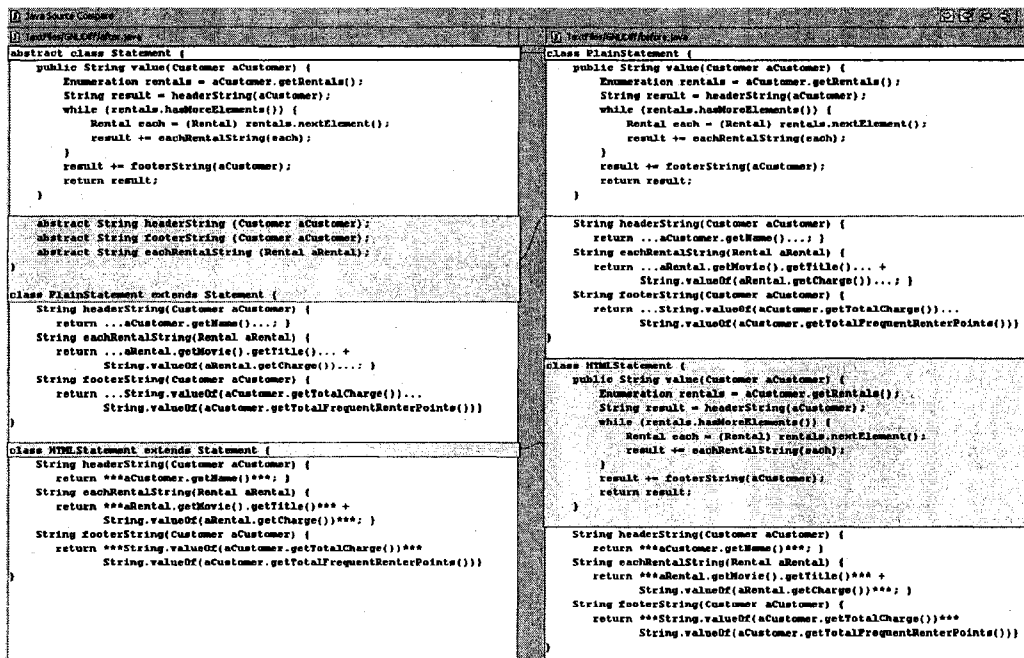
Finally, chapter 9 summarizes the concluding remarks and outlines several interesting directions we plan to continue this work.

Chapter 2: Related Work

This research relates to five research themes, i.e., differencing software artefacts, analyzing software evolution history, refactoring practices, supporting API evolution of software systems and recommending reusable or similar software artefacts, which we review in this chapter.

2.1 Software differencing

Software differencing techniques rely on comparing various program representations, such as code-line [118], source-code metrics [20], Abstract Syntax Tree [82,107], program dependency graph [47], control-flow graph [3,45], XML Metadata Interchange files of UML models [128], and unique identifiers of model elements [70,115].



```
abstract class Statement {
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }
        result += footerString(aCustomer);
        return result;
    }

    abstract String headerString (Customer aCustomer);
    abstract String footerString (Customer aCustomer);
    abstract String eachRentalString (Rental aRental);
}

class PlainStatement extends Statement {
    String headerString(Customer aCustomer) {
        return ...aCustomer.getName()...;
    }
    String eachRentalString(Rental aRental) {
        return ...aRental.getMovie().getTitle()... +
            String.valueOf(aRental.getCharge()....);
    }
    String footerString(Customer aCustomer) {
        return ...String.valueOf(aCustomer.getTotalCharge())...
            String.valueOf(aCustomer.getTotalFrequentRenterPoints());
    }
}

class HTMLStatement extends Statement {
    String headerString(Customer aCustomer) {
        return ***aCustomer.getName()***;
    }
    String eachRentalString(Rental aRental) {
        return ***aRental.getMovie().getTitle()*** +
            String.valueOf(aRental.getCharge()***);
    }
    String footerString(Customer aCustomer) {
        return ***String.valueOf(aCustomer.getTotalCharge())***
            String.valueOf(aCustomer.getTotalFrequentRenterPoints());
    }
}

class PlainStatement {
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }
        result += footerString(aCustomer);
        return result;
    }

    String headerString(Customer aCustomer) {
        return ...aCustomer.getName()...;
    }
    String eachRentalString(Rental aRental) {
        return ...aRental.getMovie().getTitle()... +
            String.valueOf(aRental.getCharge()....);
    }
    String footerString(Customer aCustomer) {
        return ...String.valueOf(aCustomer.getTotalCharge())...
            String.valueOf(aCustomer.getTotalFrequentRenterPoints());
    }
}

class HTMLStatement {
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }
        result += footerString(aCustomer);
        return result;
    }

    String headerString(Customer aCustomer) {
        return ***aCustomer.getName()***;
    }
    String eachRentalString(Rental aRental) {
        return ***aRental.getMovie().getTitle()*** +
            String.valueOf(aRental.getCharge()***);
    }
    String footerString(Customer aCustomer) {
        return ***String.valueOf(aCustomer.getTotalCharge())***
            String.valueOf(aCustomer.getTotalFrequentRenterPoints());
    }
}
```

Figure 2-1. Eclipse text compare

Lexical differencing tools, like *GNU diff* [118], are frequently used by developers, in concert with modification requests and bug reports, to reconstruct the changes between

subsequent versions of a software module [28,29,30,31]. Figure 2-1 shows the Eclipse text-comparison results between two versions of a program¹. In the after version, the duplicated method value(), which used to be implemented by classes PlainStatement and HTMLStatement, was pulled up into their new superclass, Statement. Unfortunately, the changes reported by the text-comparison tool are unintuitive: the first line was changed; five lines of code were added; a block of code was replaced by a single line. Since lexical differencing tools view software programs as text documents, they report changes at the lexical level, ignoring the high-level logical-design changes to which they correspond.

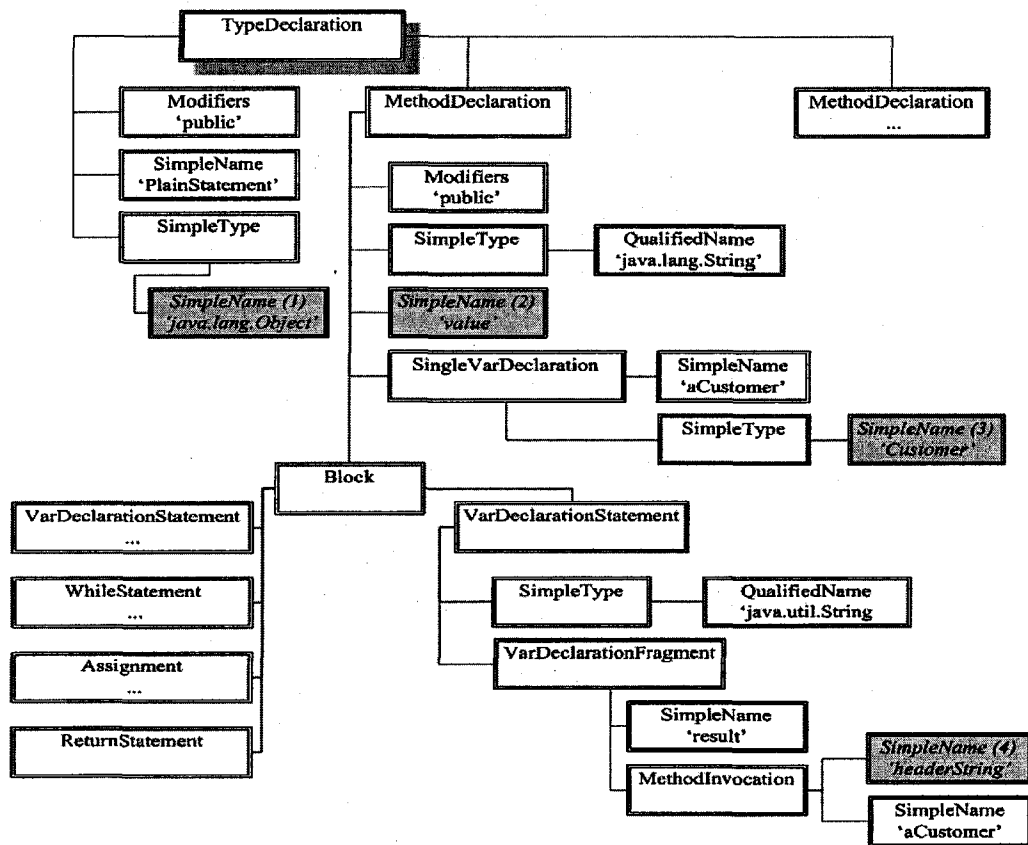


Figure 2-2. The partial AST of class PlainStatement

¹ The program is excerpted from the version 27 and 28 of the extended refactoring example at [129]. We adapt its version 23, 27 and 28 as the running example to illustrate *UMLDiff* algorithm in Section 3.3.1.

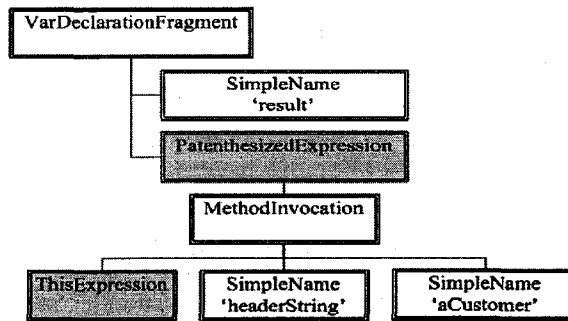


Figure 2-3. The partial AST of changed local variable declaration

The Abstract Syntax Tree (AST) is one view of the structure of a software program. Figure 2-2 depicts a partial AST of the class PlainStatement in the before version. Yang [107] developed a dynamic-programming tree-matching algorithm, for computing the similarity between ASTs. However, ASTs of realistic programs are big, which makes general tree-differencing algorithms impractical. Furthermore, they are often redundant. For example, Figure 2-3 shows the VarDeclarationFragment subtree, corresponding to a variant of the second local variable declaration – `String results = (this.headerString(aCustomer))`. Although there is no actual semantic difference between the two variants, a tree-differencing algorithm, comparing it against the original VarDeclarationFragment subtree (the bottom-right corner of Figure 2-2), would report the addition of node ParenthesizedExpression (which results in the MethodInvocation subtree being pushed one-level deeper) and the addition of node ThisExpression. AST is a low-level representation, designed for code compilation, optimization and transformation; interpreting AST changes into the higher-level logical changes requires substantial effort. For example, the value changes of four tree nodes of type SimpleName (gray highlight in Figure 2-2) represent completely different logical changes: (1) the change of PlainStatement’s superclass; (2) the renaming of the value() method; (3) the change of the associated type of the aCustomer parameter (4) the change in the outgoing usage of the method value().

There exist other differencing techniques that make use of other types of program representations. Semantic Diff [47] operates on a representation of the local dependency

graph and works at the intra-procedural level only, as opposed to the system as a whole. Horwitz developed a technique [45] for detecting statement-level semantic and textual modifications, based on augmented control-flow graphs; this method is applicable to a simplified C-like programming language and is not suitable for complex object-oriented software systems.

```

<UML:Class name="Customer" xmi.id="783">
  <UML:ModelElement.clientDependency>
    <UML:Usage xmi.idref="9E2A56"/>
  </UML:ModelElement.clientDependency>
</UML:Class>
<UML:Class name="PlainStatement" xmi.id="788">
  <UML:GeneralizableElement.generalization>
    <UML:Generalization xmi.idref="0147A1"/> [3]
  </UML:GeneralizableElement.generalization>
  <UML:Classifier.feature>
    <UML:Operation xmi.id="790793" name="valueheaderString"> [1]
      <UML:BehavioralFeature.parameter> ... </UML:BehavioralFeature.parameter>
    </UML:Operation>
    <UML:Operation name="headerStringeachRentalString" xmi.id="799799"> [1]
      <UML:BehavioralFeature.parameter> ... </UML:BehavioralFeature.parameter>
    </UML:Operation>
    <UML:Operation xmi.id="799" name="eachRentalString"> [2]
      <UML:BehavioralFeature.parameter> ... </UML:BehavioralFeature.parameter>
    </UML:Operation>
  </UML:Classifier.feature>
</UML:Class>
<UML:Class xmi.id="79C" name="Statement">
  <UML:Classifier.feature> ... </UML:Classifier.feature>
</UML:Class>
<UML:Generalization name="" xmi.id="0147A1"> [3]
  <UML:Generalization.child> ... </UML:Generalization.child>
  <UML:Generalization.parent>
    <UML:Class xmi.idref="01279C"/> [3][4]
  </UML:Generalization.parent>
</UML:Generalization>
<UML:Usage xmi.id="9E2A56">
  <UML:Dependency.client> ... </UML:Dependency.client>
  <UML:Dependency.supplier>
    <UML:Class xmi.idref="70079C"/> [4]
  </UML:Dependency.supplier>
</UML:Usage>

```

Figure 2-4. XML-differencing XMI representation of UML models

Object-oriented software systems are better understood in terms of structural and behavioral models, such as UML class and sequence models. The UML modeling tools often store UML models in XMI (XML Metadata Interchange) format for data-interchange purposes. XML-differencing algorithms, applied to such easily available XMI representations, report changes of XML elements and attributes, ignoring the domain-specific semantics of the concepts represented by these elements. Figure 2-4 shows the partial XML comparison results (by DeltaXML [128]) between the XMI representations (exported by ArgoUML [112]) of two versions of the UML class model of the program listed in Figure 2-1. The tool reports that two *UML:Operation* nodes (annotation [1]) were modified – their *name* attributes were changed – and that the *UML:Operation* with *name* attribute “eachRentalString” (annotation [2]) was removed, instead of recognizing that the *value()* method was moved to the new superclass *Statement*. Furthermore, a single logical change in the UML model may result in several changes in XMI. For example, a generalization change – the superclass of *PlainStatement* changed from *Object* to *Statement* – results in three changed XMI nodes (annotation [3]). Finally, similar XML element changes may represent completely different logical changes. For example, the attribute changes of the two *UML:Class* elements (annotation [4]) represent generalization and usage dependency change respectively. Similar to AST comparison, an interpretation step is required to aggregate and abstract the change reports of XML comparison tools in terms of higher-level logical changes.

Several UML modeling tools come with their own UML-differencing methods, such as [70,115]. They detect differences between subsequent versions of UML models, as long as these models are constructed and manipulated exclusively through the tool that assigns persistent identifiers to all model elements. This capability is clearly irrelevant when the whole development team does not use the same tool for all their development activities, which is usually the case. Furthermore, the persistent identifiers imply only one-to-one mapping between model elements, even when many-to-one or one-to-many mappings are preferable. For example, reporting that both *PlainStatement.value()* and *HTMLStatement.value()* have been moved to *Statement* better reflects the intention of the change, which is to pull up commonalities from several subclasses into the superclass, than reporting that one has been moved and the other has been removed.

There has also been some work on comparative analysis of different snapshots of a software system for drawing inferences regarding its evolution. Demeyer et al. [20] defined four heuristics based on the comparison of source-code metrics of two subsequent system snapshots to identify refactoring activities of three general categories. Rysselberghe and Demeyer [83] investigated the use of clone detection to identify move and renaming refactorings. However, the source-code metrics do not report the details of what has or has not been changed. For example, the `PlainStatement.value()` and `Statement.value()` have the same NOM (Number of Message sent in method body [63]) metrics. However, the `PlainStatement.value()` calls directly `PlainStatement`'s `header/footer/eachRentalString()`, while `Statement.value()` calls the `Statement`'s abstract methods that are implemented by the corresponding `PlainStatement` methods. Ryder's group has also worked on comparative analysis of structural changes [82]. They define a set of atomic changes derived from the comparison of the abstract syntax trees of corresponding classes in two versions of a project. Apiwattanapong et al. [3] use the enhanced control-flow graph to model methods of object-oriented programs and identify similarities and differences between two methods based-on graph isomorphism. The major objective of their work is to analyze the impact of changes on test cases, while our work is aimed at recovering higher-level design evolution knowledge.

All the above differencing techniques rely on various program representations that are designed for purposes other than understanding higher-level logical changes of software system. However, there has also been some research on analyzing the changes of software at the design level. Egyed [27] has investigated a suite of rule- and constraint-based and transformational comparative methods for checking the consistency of the evolving UML diagrams of a software system. Selonen et al. [86] have also developed a method for UML transformations, including differencing. Spanoudakis and Kim [91] developed a probabilistic message-matching algorithm that detects the overlaps between messages that are likely to signify the invocation of operations and check whether the overlapping messages violate the consistency rules. However, they cannot surface the specific types of changes as reported by *UMLDiff* and these projects have not explored the product of their analyses in service of software evolution understanding and future decision making. Godfrey et al., in their BEAGLE system [36,97], use origin analysis to

determine the “origin” of “new” files and to detect the merging and splitting of source-code entities. Origin analysis works at the file-structure level, corresponding to the physical model of the software rather than its logical model: it detects old functions as the “origin” of new ones based on a combination of clone detection and call-relation matching and assumes an interactive step for detecting file merging and splitting. In contrast, *UMLDiff* is fully automated and examines the UML logical model of the system to recognize the logical-design evolution of that system.

2.2 Refactoring

Refactoring has recently become an integral part of the evolutionary software development methodology, such as “Extreme Programming” [8]. The research area of refactoring was pioneered by Opdyke [72]. The books of Fowler [32] and Kerievsky [51] provide a good overview of the refactorings and how they can be used to accomplish architectural and design changes.

Refactorings are often associated with certain bad smells identified in the software system. Fowler lists a dozen of bad smells and informally associates them to refactorings [32], which would address the smells if they were applied to the code. Balazinska et al. [5] and Ducasse et al. [25] use clone-detection techniques to detect code duplication and propose refactorings to eliminate it. Tourwe and Mens [96] rely on logic meta-programming to specify and detect certain types of bad smells, such as obsolete parameters and inconsistent interfaces. Simon et al. [89] use cohesion metrics to measure which methods and fields belong together and then, based on this information, they propose “move method” and “extract/inline class” refactorings.

By definition, refactorings are behavior-preserving program transformations. Opdyke’s Ph.D. thesis [72] lists a set of invariants and preconditions to which a refactoring must conform in order to be considered behavior preserving. Mens et al. [66] study the application of graph-representation and graph-rewriting techniques to specify and determine whether or not a property is preserved when performing refactorings, such as pull-up methods. Tip et al. [95] use type constraints to model refactorings dealing with generalization and compute the allowable source-code transformations that preserve type correctness.

Although refactoring is usually a developer-intensive activity, tool support is considered crucial. Roberts et al. developed the first tool for providing semi-automatic refactoring support, which was implemented in the Refactoring Browser [79]. Modern IDEs, such as Eclipse [116] and IntelliJ IDEA [124], offer semi-automatic support for most commonly used refactorings. Moore [67] developed a fully automated tool, *Guru*, for the automatic restructuring of inheritance hierarchies and refactoring of methods in *Self* program in order to reduce program size and boost code reuse.

Refactorings affect the quality of software systems. Demeyer [21] investigated the effect of refactorings by replacing conditional branches with polymorphism. Tahvildari and Kontogiannis [93] analyzed the impact of refactoring on source metrics to provide evidence where a refactoring may improve a system quality, such as maintainability. Leitch [59] developed a method for estimating the return-on-investment for a refactoring, by estimating, in the event of a function-preserving change to a code-base, the savings in future maintenance costs because of the change compared to the effort required to perform the change.

There has also been some work on reverse engineering refactorings by mining software repositories. Demeyer et al. [20] defined four heuristics based on the comparison of source-code metrics of two subsequent system snapshots to identify refactorings of three general categories. Rysselberghe and Demeyer [83] investigated the use of clone detection to identify moves and renamings. Godfrey and Zou [36] used origin analysis to detect the “merging” and “splitting” of source-code entities. Dig and Johnson [22] conducted an empirical study to assess the role of refactorings in API migration. Their analysis relies on the changes documented in the release notes shipped with software systems. Gorg and Weisgerber [37] analyzed the changes of entity signatures in evolution transactions – i.e., changes to a set of files committed at the same time by the same developer - to detect refactorings, such as *rename method*, *move class*, *pull-up method*. Their approach is sensitive to the “multiple refactorings on the same location”.

Refactoring the reused components (library or framework) is often limited by the fear of breaking client code. When the breaking API changes happen, the developers of component-based applications take the burden of migrating their codes to the new version

of reused components. Balaban et al. [4] developed a tool that allows the developers to define a mapping specification between legacy classes and their replacements so that obsolete library classes (such as Java Vector) can be replaced with their newer counterparts (such as Java ArrayList). CatchUp [42] is another attempt to relieve this burden by recording the refactorings, such as type renamings, made by the component developers within an IDE, such as Eclipse, and then replaying them by the application developers on the client code to keep it updated. In the similar vein, Borland demonstrated their “Team Refactoring” support for JBuilder [50].

We have also been working on this general problem. We first outlined process for recognizing refactorings from the design changes reported by *UMLDiff* in [101] and the detailed catalog of refactoring-detection queries was recently discussed in [106]. We developed an alternative method for detecting a more limited set of refactorings based on metrics and lexical information analysis [85]. In this work, we compared our two methods and found that query-based refactoring detection is more accurate - it recognizes all documented refactorings and more. Interestingly enough, the not-documented refactorings do not appear to be equally effective from a maintainability-improvement perspective and might be accidental side-effects of other changes. Next [105], we reported in detail on our exploration of the Eclipse evolution history using our query-based refactoring-detection process and we summarized how refactoring is practiced in the Eclipse evolution and we discussed how more systematic support could be developed for performing refactorings and for automatically propagating the impact of framework refactoring to the applications using it.

2.3 Software evolution analysis

A large subset of work in this area involves analyzing the “history” recorded by version-management systems. Lehman and Belady [58] proposed laws of software evolution after analyzing change data from the evolution of the OS/360 operating system.

Eick et al. [30] developed a process for analyzing the change history of the code, which is assumed to reside in a version management system, calculating code-decay indices, and predicting the fault potential and change effort through regression analysis.

The objective of this research is mainly to support project management so that code decay is delayed.

Barry et al. [7] use software volatility, defined in terms of amplitude, periodicity, and dispersion of software changes, to define a set of evolutionary patterns in the lifecycle of software systems. Their work shows that systems evolve in different styles; in contrast, our phasic analysis focuses on understanding the structural properties of system classes' lifecycle behavior and the underlying factors that drive their evolution over time.

Hassan and Holt [41] and Bianchi et al. [10] claim that information entropy, i.e., the amount of uncertainty related to software products, is a good indicator of the degree of disorder of its structure. Intuitively, in the context of software evolution, if a software system is being modified across all its modules, it has highest entropy, and the software maintainers will have a hard time keeping track of all the changes. Their work relies on maintenance documentation to keep track of software modifications in order to compute information entropy of files that evolved over a period of time. Our class co-evolution analysis may provide a good primary input for these system instability analyses.

Gall et al. [31,33] developed an approach for populating a release-history database that combine CVS (Concurrent Version System) data (including CVS deltas produced by *GNU diff* [118]) and bug reports maintained in Bugzilla [113]. Queries can then be issued to obtain multiple views, including module coupling, that show the evolution of a software project. Fischer et al. [31] pointed out the need for investigating high-level structural changes to enable more accurate information in their release history database than using *GNU diff*.

Shirabad et al. [87] applied inductive-learning algorithms to address the problem of detecting the co-updating of two code modules and predicting whether updating a source file would necessitate a change in another file. Their chosen inductive-learning methods require pre-labelled positive examples of co-updating, i.e., sets of modules that were updated for the same change request. Therefore, a lot of effort is required to select and extract characteristic features and label training examples, which may significantly affect the quality of the learned model. In contrast, our class co-evolution detection is based on Apriori association rule mining [2].

The major drawback of this line of research is that it requires high-quality developers' comments and change reports recorded in the version-management system. In most real-world software projects, such comments and reports are inconsistent in their detail and sometimes they do not even exist. In contrast, our design-evolution analysis methods are based on the automatically detected design changes by *UMLDiff* between versions of system's logical model, without requiring consistently maintained high-quality change documentation.

Another line of 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 tools [28,29] for visualizing the evolution of software statistics at the source-code line level and change data such as developer, size, effort, etc.

Zimmermann et al. [110] have also developed visualizations of historical data stored in a CVS archive to help developers recognize the coupling between fine-grained program entities like methods and fields. Bevan [9] defines software instability as a set of related artefact elements that have often changed together. She used a static dependence graph to visually identify such related software artefacts.

German and Hindle [35] developed a visualization tool, *softChange*, that can be used to retrieve and visualize information stored in various forms, such as emails, releases notes, change logs, etc., and assist the programmers in understanding how software has evolved to its current state.

Lanza [57] describes how to use a simple two-dimensional graph to convey the implicit information of software metrics. Based on the visualization of the evolution of class metrics (e.g., *number of methods and number of instance variables*), the evolution matrix can be used as an indicator of the evolution phases of the software system and its components, similar but more coarse-grained than our taxonomy of the evolution phases and styles, thus failing to recognize restructuring activities.

These visualization approaches adopt a top-down method, which limit their applicability 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 design-evolution analyses starts with the minimum amount of information about the automatically recovered core

evolution concerns and support developers incrementally build up their knowledge about what has been changed, how and why.

The existing approaches [9,33,87,110] to elicit the co-evolution of software artefacts produce Boolean association rules, which concerns associations between the change and no-change of software artefacts. They have limited explanatory power on why the co-evolving artefacts are inter-dependent and cannot guide precisely the future development of the system. For example, it is hard for these methods to identify fine details of co-evolution relations, such as the following: “when adding members to class A, class B also gets additional members too”; or “when adding members to class A, it is often needed to restructure class B’s interface”. These fine-grained co-evolution rules concern associations between different types of changes. Our class co-evolution analysis is based on the discrete class-evolution profiles that characterize the categorical change volatility of the system classes over time, which allow us to produce four types of co-evolution rules: parallel function-extension, parallel refactoring, and parallel function-extension and refactoring.

Several approaches are available to help developers locate and manage source code that are scattered throughout the program text. Robillard and Murphy developed FEAT tool [80] that supports defining, locating, and analyzing the code implementing one or more concerns. Relo [88] monitors the developer’s exploration of code within an IDE and builds automatically the relevant elements and relations into a centralized view. ActiveAspect [111] 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.

2.4 Supporting API evolution

Some modern integrated development environments (IDEs), such as Refactoring Browser [79], Eclipse [116] and IntelliJ IDEA [124], provide semi-automatic support for applying the most commonly used, low-level refactorings, such as *rename field* and *move method* [32]. Refactoring support within IDEs has made it less cumbersome to improve code quality. However, the refactoring engine requires that the complete source code of the

refactored system is visible and modifiable by the tool, which is obviously irrelevant in the API evolution of reuse-based development, since parts of the system are either not accessible or not changeable at the time of refactoring.

Recently, configuration management systems that support merge of refactored branches of a system have been proposed in order to allow teams of developers to share refactorings. For example, CMEclipse [114] and MolhadoRef [24] aim at improving the configuration-management support for refactorings when merging parallel development branches. Those systems do not assume complete source-code access when performing refactorings, but they require a common, centralized development infrastructure be used. However, in the context of software reuse, the client applications and the component frameworks they reuse are usually much more loosely coupled. For example, a library may distribute the changes by posting new JAR files on its own website, from which the client-application developers download those updates and integrate them on the client side.

A critical problem in the effort to adapt client applications to the interface changes of their underlying component frameworks is the precise specification of the interface changes of the reusable components and the propagation of these changes to the client-application source code. Programming-language syntax for explicitly annotating API changes, such as the keywords “deprecated” in Java and “obsolete” in Eiffel, may be used to indicate the obsolescence of a construct, to discourage developers from further using an old version in the presence of a newer one. However, there is no standard language support for migrating existing client code, built on the deprecated API. Perkins [76] proposes a technique based on method inlining for replacing calls to a deprecated method by the method’s body, assuming that the deprecated method delegates to its replacement. However, Henkel and Diwan [42] found that better tool support is required: deprecated entities that are part of the published interfaces are almost never removed, which indicates that once an API is published, developers are forced to maintain it.

When the API of a component is changed, the changes and suggested ways to adapt any existing application code to them are usually documented in the new release of the component, such as the Eclipse “What is New” [119] or the Microsoft Visual C++ “Migrating from Previous Versions” [121]. This approach requires a significant amount

of effort by the component developers to produce the documentation in the first place and to consistently maintain it over time. However, more often than not, the documentation does not tell the whole truth. It may discuss only a subset of the actual API changes that the component developers consider important [104]. Furthermore, it may not always consistently reflect the actual code. Finally, the documentation is sometimes written in very compact – even cryptic – language, not easily understandable by most application developers.

Chow and Notkin [15] developed a system for semi-automatically updating applications in response to library changes, which requires the library developer to specify the interface changes and how existing application code can be transformed to work properly in the face of those changes. The change and transformation specification are distributed along with the evolved libraries, which are then used by application maintainers to update their applications semi-automatically. This approach shifts the burden of upgrading source code for library-interface changes from the application developers to the library developers. Its main shortcoming is that it assumes that the change and transformation are manually specified for each interface change, which requires a substantial effort to write and maintain. Furthermore, the migration process is sensitive to the completeness and correctness of the library developer's specifications.

Henkel and Diwan [42] propose an approach for recording and replaying refactorings, based on the existing refactoring support of modern IDEs, such as Eclipse, to support API evolution. In a similar vein, Borland demonstrated its “team refactoring” support of JBuilder at the JavaOne conference [50]. These approaches relieve the library programmer from manually writing change specifications and delegate the error-prone work of validating and applying changes to industrial-strength IDEs. However, they require that the framework developers and the developers of client applications use same refactoring engines so that the recorded refactorings can be interpreted and replayed on the client applications. Furthermore, the supported changes are limited to those supported by the refactoring engine of a given IDE and these approaches are not aware of the refactorings that are manually performed.

All the above approaches to adapting client applications in response to interface changes of reusable components rely on additional, and potentially substantial, work by

the component-framework developers, whether it is coding to particular standards, or learning some new specification language, or using some specific tool. Although the client-application developers still have the final decision as to update their source code or not, the decision on what can be updated and how is essentially driven by the additional information provided by the component-framework developers that document the changes and advise on how to adapt them. However, it is seldom the case that the change documentation and scripts provided with a large framework are sufficient for a client-application developer to effectively migrate to the changed APIs, in spite of a substantial effort to write and maintain the change documentation or scripts on the part of component-framework developer. All too often, application developers become lost when trying to reuse a changed API, unsure of how to make progress on a migration problem.

To ease the responsibility of the component-framework developers and to help the application developers find their way, we have developed an approach that uses the automatically detected API changes to support the migration of client applications. With our approach, the client-application developers decide what they want to update and how exactly. They run an automated tool to obtain the interface changes of reusable components and the likely ways in which they may update their application in order to properly use the evolved component APIs. Next, they may interactively explore the changes and suggested updates with the support of our visualization component – JDevAn Viewer (see Chapter 7) – so that they are able to better understand the nature of the change, its plausible replacements and how to use them.

2.5 Recommending software artifacts

There has been some work on discovering a component appropriate for reuse within an application. Zaremski and Wing [109] investigated signature matching, such as the comparison of the types of a function's input and output parameters in addition to function's name, for retrieving reusable software components. Ostertag et al. [73] presented an AI-based reusable library system that supports a developer to search library components that best meet the given requirements: they relate the software library components with manually-defined features and terms through domain analysis; they also define manually the weighted subsumer and feature graph over components, based on

which component similarity may be computed. The CodeBroker tool [108] monitors the declaration of method and the insertion of comments in a text editor and queries a library with that information to find components that could be reused instead of a new one being created. To determine matches, a developer must properly format comments in the program being developed in terms similar to that of reusable components in the library. The Strathcona [44] tool avoids the need for writing appropriate comments. Instead, it automatically extracts the structural facts about the context of a code fragment of interest highlighted by a developer and uses this structural context to heuristically search a source-code repository for examples with similar structural context.

On the other hand, many approaches have been proposed to use the artifacts associated with a software project, in addition to source code, to support software maintenance and evolution tasks. The version check-in description (such as the author information) may be used to generate recommendations of people who might have some expertise on a given problem [64]. The developer's communication (such as email, annotations on the code, etc.) may be used to present documents similar to the one currently being edited [78]. Hipikat [17] offers contextual advice during development by extracting similar situations in the history of the system lifecycle. It recommends relevant segments of documentation and/or similar code snippets from which the developers can draw analogies towards solving their current problem, such as suggesting a potential solution to a particular type of bug. We studied software design-mentoring that works at the granularity of design level, providing supports on monitoring and mentoring object-oriented software design and its evolution.

All the above approaches focus on recommending examples to reuse from the current version of the component framework or offering similar documentation, code snippets or design changes from the information stored in a project's archives of a "closed" system to aid the developer's evolution task. In contrast, our API-Evolution catch-up approach addresses the asynchronous API evolution problem in the context of reuse-based software development, which proposes the API replacements and their potential usage examples for upgrading client applications in the face of the evolving APIs of the components that they reuse.

Chapter 3: Model Differencing with *UMLDiff*

The first research problem that this thesis addresses is how to accurately and efficiently recognize the changes in the system's design as it evolves from one version to the next. As discussed in Section 2.1, there exist a number of software differencing techniques [3,20,45,47,82,107,118,128], but they rely on program representations that are designed for purposes other than capturing higher-level logical design changes of software system.

We have developed *UMLDiff*, a heuristic algorithm for automatically detecting the changes that the logical design of an object-oriented software system has gone through, as the subject system evolved from one version to the next. This algorithm takes as input two logical models corresponding to two versions of an object-oriented software system. It traverses the two models in parallel, moving from one type of model elements to the types they contain; as it does so, it identifies corresponding elements, i.e., model elements that correspond to the same conceptual design entity, based on their *lexical* and *structural similarity*. *UMLDiff* produces as output a set of *design-change facts* reporting the various types of design changes it has discovered when comparing the two models, i.e., additions, removals, moves, and renamings of subsystems, packages, classes, interfaces, attributes and operations, and changes to the attributes and relations of these model elements.

In this chapter, we describe the meta-model assumed by *UMLDiff* as the underlying representation of its input logical models and the process by which these models are extracted from Java software. We discuss in detail the algorithm, and the similarity heuristics on which it relies to recognize correspondences between the design elements of the two compared models. Finally, we report on our experimental evaluation of the effectiveness and robustness of the *UMLDiff* algorithm with the JFreeChart [126] system.

3.1 The meta-model

UMLDiff compares logical models of object-oriented software systems. The underlying meta-model is defined according to the semantics of the UML metamodel [69]. We summarize the UML profile in terms of metaclasses and metarelations, of concern to *UMLDiff*, in Appendix A.

instances of *[owner - feature]* meta-composition). The operation `HTMLStatement.printFooter(Customer)` is associated with the class `String` as its return type (an instance of *[typedParameter - type]* meta-association). The operation `Customer.htmlStatement()` instantiates (an instance of *Usage<<instantiate>>*.) the object `HTMLStatement`. The class `PlainStatement` is a subclass of the class `Statement` (an instance of *generalization*). The operation `HTMLStatement.printEachRental(Rental)` implements (an instance of *Abstraction<<realize>>*) the abstract operation `Statement.printEachRental(Rental)`.

3.2 UML model reverse engineering in JDEvAn

The *UMLDiff* algorithm has been implemented in the JDEvAn tool [125], which also implements a Java fact extractor based on the Eclipse Java DOM/AST model. JDEvAn's Java fact extractor reverse engineers UML models in the form expected by *UMLDiff*, from Java source code. The mapping of the Java language constructs to UML metaclasses and metarelations is described in Appendix B. Our current focus on Java is pragmatic; the *UMLDiff* is not restricted to any specific object-oriented programming language, since its meta-model is essentially defined according to the UML semantics. Its design and implementation are extendible to software systems developed in other object-oriented programming languages, assuming appropriate fact extractors that are able to map programming language constructs into UML model elements and relations expected by *UMLDiff*.

Java software subsystem is not really a Java construct; it is a conceptual element. The top-level subsystem corresponds to the model of a system as a whole. Each *Operation* is associated with a *Method* element, which contains the body of the corresponding Java method, constructor, or class initializer. A *Method* element is not contained in the declaring class of its specification operation. The return type of a Java method is treated as a special *Parameter*, whose *name* and *kind* attributes are *return*. A field's initializer is modeled as the *initValue* attribute associated with its corresponding *Attribute*. Although Java requires exceptions to be subclasses of `java.lang.Throwable`, other programming languages, such as C++, allow exceptions to extend arbitrary classes. Therefore, to avoid restricting *UMLDiff* to the Java particulars, the fact extractor does not explicitly model exceptions; instead, at the end of fact extraction process, it marks the classes that appear

in *Usage*_{<<send>>}, *[context – raisedSignal]*, and *[reception – signal]* relations as exceptions. The fact extractor does not model *Receptions* either, since operations are normally receptions that handle the signals, such as exceptions, in most modern object-oriented programming languages. Instead, at the end of the fact extraction process, it marks the operations that appear in *[reception – signal]* relations as receptions. Finally, the fact extractor ignores three Java specific modifiers, *volatile*, *native*, and *strictfp*, and assumes that the classes and interfaces that belong in Java libraries are contained in the top-level subsystem.

Each extracted model element is described in terms of its name, the type of its corresponding UML metaclass (as described in Table A-1 and Table B-1), its corresponding visibility and attribute(s) (as described in Table B-3), and its attached *UMLDiff*-specific tagged values (as described in Table A-4 and Table B-4). The relations between model elements are described in tuples of the form *(relation, e₁, e₂)*, where *e₁* and *e₂* are model elements and *relation* is a type of UML metarelation, as described in Table A-2, Table A-3 and Table B-2, that applies between *e₁* and *e₂*. The number of times that a field is read/written, a method/constructor is called, a class is created, and a class/interface is used is recorded as the *count* tag, attached to the corresponding usage dependency.

The name of array types is in the form of “BasetypeQualifiedname.Dimension”. The name of packages, classes, interfaces and fields is their declared identifier. The name of methods and constructors is in the form of “identifier(paramtype_list)”. JDEvAn’s fact extractor also assigns names to anonymous classes, “new supertype_identifier\$number”; class initializers, “{class_identifier.\$number}”; and field initializers, “{field_identifier=...}”. The “number” is the ordinal number of the anonymous class or the class initializer within the enclosing Java class. Finally, a fully qualified prefix is added in front of the names of model elements that belong in Java libraries.

Anonymous classes are a special type of nested classes. They do not explicitly have declared identifiers. They are specified along with class creation expression within blocks and are then generated by the compiler when parsing expression as the nested classes of the class that declares the corresponding block. Thus, an anonymous class is modeled as a class whose *name*=“new supertype_identifier\$number”, *visibility*=*private*,

isAbstract=false, and *isLeaf=true*, which is contained by the corresponding enclosing class. It is also associated with the corresponding *Method* element of its declaring operation. The *Usage*`<<instantiate>>` dependency between the anonymous class and its declaring operation is not modeled. Instead, it is mapped to the direct super type of the anonymous class. The fact extractor does not model Java local classes/interfaces, which are declared within methods, constructors, or class initializers, because they are rarely used in practice.

The extracted UML models are stored in a PostgreSQL relational database, extended with Simon's transitive closure algorithm [90] for computing transitive containment and inheritance relations, field read/write, method call, and class/interface usage relations. The relational database enables the *UMLDiff* implementation to work on large-scale software projects, such as Eclipse [116]. It also provides the flexibility to infer derivable information about model elements and their relations.

3.3 Comparing logical models of object-oriented software

In this section, we discuss in detail how *UMLDiff* maps model elements and their relations and compare their attributes.

3.3.1 The running example

We will demonstrate how *UMLDiff* works with a small running example, adapted from the versions 23, 27 and 28 of the extended refactoring example at [129]. When the system evolves from version 23 to 27, the nested class *PlainStatement* is extracted from the class *Customer*. The main responsibility of this class is to print out the customer's movie rental information in plain text format, which is originally performed by the operation *Customer.statement()* in version 23. In version 27, the operation *Customer.statement()* instantiates a *PlainStatement* object, to which it delegates this task. Similar changes are also made to *Customer.htmlStatement()*₂₃² and the newly introduced top-level class *HTMLStatement*₂₇ contained in *default*₂₇ package. Furthermore, the *Customer.statement()*₂₃ is

² Denotes the model element contained in a particular version.

renamed to `plainStatement()`²⁷ in order to be consistent with `htmlStatement()` and to more clearly convey the intention of the method.

The main change between versions 27 and 28 is to pull up the operation `value()` from `PlainStatement` and `HTMLStatement` to their superclass, `Statement`. However, to demonstrate several *UMLDiff* key features, we intentionally complicated versions 27 and 28 by including the following changes:

- We renamed the operations `getTotalChange()/getTotalFrequentRenterPoints()` of the class `Customer` to `getAllCharge()/getAllFrequentRenterPoints()` respectively and changed their visibilities from package to public;
- We renamed the operations `headerString()/eachRentalString()/footerString()` of the classes `Statement/PlainStatement/HTMLStatement` to `printHeader()/printEachRental()/printFooter()` respectively;
- In version 27, the class `PlainStatement` is a nested class of the class `Customer`, while, in version 28, it is moved out from the class `Customer` and becomes a top-level class contained in the default package;
- In version 27, the operation `PlainStatement.footerString()` uses `String.value(double)/value(int)` to convert the double and int values to `String`, while, in version 28, it changes to use `Double.toString()` and `Integer.toString()`; and
- In version 28, the operation `Customer.plainStatement()` is deprecated.

3.3.2 *UMLDiff* overview

UMLDiff is an UML-semantics-aware differencing algorithm. As per the adopted meta-model, the software system is modeled as a directed graph $G(V, E)$, where the vertex set V contains model elements and the edge set E contains relations among them. Given two versions, “before” and “after”, of a UML model and their corresponding graphs $G_{before}(V_{before}, E_{before})$ and $G_{after}(V_{after}, E_{after})$, *UMLDiff* essentially maps the two model graphs by computing the intersection and margin sets between (V_{before}, V_{after}) and (E_{before}, E_{after}) , in terms of $(V_{before} - V_{after})$ for the removed model elements, $(V_{before} \cap V_{after})$ for the mapped (i.e., matched, renamed, and moved) elements, $(V_{after} - V_{before})$ for the added model elements, $(E_{before} - E_{after})$ for the removed relations, $(E_{before} \cap E_{after})$ for the matched relations, and $(E_{after} - E_{before})$ for the added relations.

Table 3-1. The containment hierarchy of UML model elements

| Type of model element | Type of the children |
|-----------------------|---|
| Top-level Subsystem | Subsystem and Package ProgrammingLanguageDataType Class and Interface whose isFromModel=false |
| Subsystem | Subsystem and Package |
| Package | Package, Class and Interface |
| Class | Class and Interface Attribute, Operation, Operation<<create>>, Operation <<initialize>> |
| Interface | Class and Interface Operation |
| Attribute | N/A |
| Operation | Parameter |

UMLDiff first attempts to map the model element sets V_{before} and V_{after} . It relies on the composition relations to traverse in a breadth-first fashion³ the vertices (model elements) of the directed graph of the UML model. The composition relations (instances of three meta-compositions – see Table A-3) induce a strict spanning tree of the containment subgraph of the system model. The UML semantics guarantees that all model elements can be visited by traversing the containment hierarchy starting from the top-level subsystem corresponding to the system version and the children of their containing parent are unique in terms of their names. The meta-composition defines four logical levels over types of model elements: subsystem (including the top-level subsystem) > package > (class, interface) > (attribute, operation). The model elements of type subsystem, package, class and interface may contain the nested same-type elements. Table 3-1 summarizes the containment hierarchy of the UML model elements. Table 3-6 shows the partial containment hierarchy of versions 23 and 27 of the model of our running example.

UMLDiff traverses the containment-spanning trees of the two compared models, descending from one logical level to the next, in both trees at the same time. It starts at the top-level subsystems that correspond to the two system versions and progresses down to subsystems, packages, classes and interfaces, and finally, attributes and operations.

³ In the rest of this section, all references to “traversals” are implied to be “breadth-first traversals”. That is, the model elements are processed in First-In-First-Out order.

UMLDiff recognizes that a model element e_1 in the “before” version and an element e_2 of the same type in the “after” version are the “same”, i.e., they correspond to the same conceptual model element, when (a) they have the same or similar name and comment (*lexical-similarity heuristic*), and (b) they have similar relations to other model elements, that have the same name and type or have already been established to be mapped (*structure-similarity heuristic*).

Name similarity is a “safe” indicator that e_1 and e_2 are the same entity: in our experience with several case studies, very rarely is a model element removed and a new element with the same name but different element type and different behavior is added to the system. *UMLDiff* recognizes same-name model elements of the same type first and uses them as initial “landmarks” to subsequently recognize renamed and moved elements. When a model element is renamed or moved, as is frequently the case with refactorings, its relations to other elements, such as the children elements it contains, the attributes it reads/writes, the operations it calls or is called by, etc., tend to remain much the same. Therefore, by comparing the relations of two same-type model elements renamings and moves can be inferred: if they share “enough” relations to known-to-be-same or same-name elements of the same type they are the “same”, even though their names and/or their parent (containing) model elements are different. Whenever two model elements are identified as renamings or moves, this knowledge is added to the current landmarks’ set and is used later on to further match as not-yet-mapped elements. This process continues until it reaches the logical-leaf level of the spanning trees and all possible corresponding pairs of model elements have been identified.

Given two renaming or move candidates, *UMLDiff* computes their structural similarity in terms of the intersection of their two related-element sets, i.e., the sets that contain same-name of the same type or established-to-be-mapped model elements that are related to the two compared candidates with a given type of relation. Therefore, if all or most the model elements related to two candidates were also renamed and/or moved and cannot be established as “same”, the *UMLDiff* structure-similarity heuristic would fail. If, on the other hand, a set of related elements were renamed or moved but enough model elements related to the affected set remained the “same”, it would be possible to recognize this systematic change.

UMLDiff applies two techniques, i.e., multiple-rounds-of-renaming-and-move-identification and propagating-operation-renamings-along-inheritance-hierarchy, to propagate the knowledge of established renamings and moves along their usage and inheritance relations (see Section 3.3.4.4 and Section 3.3.4.5). Finally, global renamings, such as renamings to meet a new naming convention, for example, may be recovered, by enabling the user to specify a string transformation – introducing a prefix or appending a suffix, or replacing a certain substring – that should be applied to the names of the model elements of one of the compared versions, before the differencing process.

Table 3-2. The summary of design changes reported by *UMLDiff*

| Element type | Categories and types of <i>UMLDiff</i> changes |
|---------------------|---|
| Subsystem & Package | Added, removed, renamed, or moved |
| Class & Interface | Added, removed, renamed, or moved Generalization change of class and interface, and no-longer or new interface implementation of class No-longer or new outgoing and incoming usage dependencies Visibility, modifier, deprecation-status change |
| Operation | Added, removed, identifier-changed, moved, extracted, or inlined Parameter added, removed; parameter type changed No-longer or new outgoing attribute read/write, operation call, class instantiation; no-longer or new incoming call No-longer or new declared, thrown, and caught exception Return type change Visibility, modifier, deprecation-status change |
| Attribute | Added, removed, renamed, or moved No-longer or new read-by and written-by dependencies Data type change Visibility, modifier, deprecation-status change |

Once *UMLDiff* has completed mapping the vertex (model element) sets V_{before} and V_{after} , *UMLDiff* proceeds to map the edge (relation) sets E_{before} and E_{after} , by comparing the relations of all pairs of model elements (v_{before}, v_{after}) , where $v_{after} = \text{null}$ if v_{before} is removed and $v_{before} = \text{null}$ if v_{after} is added. The relations from (to) a removed model element are all removed and the relations from (to) an added model element are all added. For a pair of mapped elements (v_{before}, v_{after}) , they may have matched, newly added, and/or removed

relations. Note that a removed (added) relation between two model elements does not indicate any of the elements it relates being removed (added).

Next, *UMLDiff* detects the redistribution of the semantic behavior among operations, in terms of usage dependency changes, and finally computes the changes to the attributes of all pairs of mapped model elements. Table 3-2 summarizes the categories and types of design changes reported by *UMLDiff*.

The *UMLDiff* differencing process is configured through a set of parameters.

- The *LexicalSimilarityMetric* specifies which one of three lexical-similarity metrics (Char-LCS, Char-Pair, and Word-LCS) will be used by *UMLDiff*.
- The *RenameThreshold* and *MoveThreshold* are the minimum similarity values between two model elements in the two compared versions in order for them to be considered as the same conceptual element renamed or moved. *UMLDiff* allows multiple rounds (*MaxRenameRound* and *MaxMoveRound*) of renaming and move identification in order to recover as many renamed and moved entities as possible.
- The similarity of the comments of the model elements (*ConsiderCommentSimilarity*) may also be taken into account when comparing two elements, if the compared elements have an initial overall similarity value above the *MinThreshold*; this prevents model elements with very low name- and structure-similarity from qualifying as renamings or moves just because of their similar comments.
- The similarity of transitive usage dependencies (*ConsiderTransclosureUsageSimilarity*) between two compared operations may also be used to assess their structural similarity.
- At the end of the *UMLDiff* differencing process, it can be instructed whether or not to compute the usage dependency changes for all model elements and analyze the redistribution of operation behavior.

3.3.3 Similarity metrics

UMLDiff relies on two heuristics – lexical and structure similarity – for recognizing the conceptually same model elements in the two compared versions of the system model, in spite of the fact that they may have been renamed and/or moved. In the following

discussion, the term “matched elements” refers to same-name model elements of the same type, while “mapped elements” refers to matched, renamed, and moved elements.

3.3.3.1 Lexical similarity

The term “lexical similarity” refers to the string similarity between the names of and the comments associated with two compared model elements. *UMLDiff* integrates three metrics of string similarity: (a) the longest common character subsequence (Char-LCS); (b) the longest common token subsequence (Word-LCS); and (c) the common adjacent character pairs (Char-Pair). All these metrics are computationally inexpensive to calculate, given the usual length of the names and comments of model elements. They are also case insensitive, since it is common to misspell words with the wrong case or to modify them with just case changes. They are all applicable to name similarity, while only Char-LCS and Word-LCS may be applied to compute comment similarity.

The name similarity of operations is calculated as the product of their identifier similarity and their parameter-list similarity, which is computed as one type of structure similarity for operations. The name similarity of packages is computed based on their dot-removed names. The comment similarity between two model elements is only consulted when both elements have associated comments, the *UMLDiff* parameter *ConsiderCommentSimilarity* is true, and the initial overall similarity metric between these elements is greater than the *UMLDiff* parameter *MinThreshold*.

The longest common character subsequence (Char-LCS) algorithm [98] is frequently used to compare strings. Word-LCS applies the same LCS algorithm, using words instead of characters as the basic constituents of the compared strings. The names of model elements are split into a sequence of words, using dots, dashes, underscores and case switching as delimiters. Comments are split into words using space as delimiters. The actual metric used for assessing LCS-similarity is shown in the following equation:

$$\text{Char/Word-LCS}(s_1, s_2) = 2 * \text{length}(\text{LCS}(s_1, s_2)) / (\text{length}(s_1) + \text{length}(s_2)),$$
 where $\text{LCS}()$ and $\text{length}()$ is based on either characters or words.

LCS reflects the lexical similarity between two strings, but it is not very robust to changes of word order, which is common when renaming a model element. To address this problem, we have defined the third lexical-similarity metric in terms of how many

common adjacent character pairs are contained in the two compared strings. The $pairs(x)$ function returns the pairs of adjacent characters in a string x . By considering adjacent characters, the character ordering information is, to some extent, taken into account. The Char-Pair similarity metric, which is a value between 0 and 1, is computed according to the following equation:

$$\text{Char-Pair}(s_1, s_2) = 2 * |\text{pairs}(s_1) \cap \text{pairs}(s_2)| / (|\text{pairs}(s_1)| + |\text{pairs}(s_2)|).$$

3.3.3.2 Structure similarity

Table 3-3 lists the relations that *UMLDiff* examines to compute the structure similarity between two model elements of the same type. The top-level subsystems, corresponding to the two compared versions of a UML logical model, are always assumed to match. The structure similarity of subsystems, packages, classes and interfaces is determined by the elements they contain, the elements they use, and the elements that use them. The structure similarity of attributes is determined by the operations that read and write them and their initialization expressions. The structure similarity of operations is determined by the parameters they declare, their outgoing usage dependencies (including the attributes they read and write, the operations they call, and the classes/interfaces they create), and their incoming usage dependencies (including the attributes (through their *initValue*) and the operations that call them).

Table 3-3. The UML relations for computing structure similarity

| Type of model element | Type of relations |
|-----------------------|---|
| Top-level subsystem | Always match |
| Subsystem | [namespace – ownedElement] Incoming and outgoing usage |
| Package | [namespace – ownedElement] Incoming and outgoing usage |
| Class and Interface | [namespace – ownedElement] and [owner – feature] Incoming and outgoing usage |
| Attribute | Usage<<read>> Usage<<write>> and inherent Attribute.initValue |
| Operation | [BehaviorFeature – parameter] and [typedParameter – type] Outgoing usage: Usage<<read>>, <<write>>, <<call>>, <<instantiate>> Incoming usage: Usage<<call>> |

The structure similarity of two compared elements is a measure of the overlap between the sets of elements to which the compared elements are related, according to a given relation type. The intersection of the two related-element sets contains the pairs of model elements that are related to the compared elements and have already been established to be mapped or have the same name and element type. This intersection set effectively incorporates knowledge of any “known landmarks” to which both compared model elements are related. Given two model elements of the same type, e_1 and e_2 , let Set_{before} and Set_{after} be their related-element sets, the structure similarity between e_1 and e_2 according to a given group of relations is a normalized value (between 0 and 1) as computed in the following equation:

StructureSimilarity = matchcount / (matchcount + addcount + removecount), where the matchcount, addcount, and removecount are the cardinality of $[Set_{before} \cap Set_{after}]$, $[Set_{after} - Set_{before}]$, $[Set_{before} - Set_{after}]$ respectively.

For a usage dependency, its *count* tag, which indicates the number of times that it appears between the client and supplier elements, is used to compute its matchcount, addcount, and removecount.

The similarity of the parameters of two compared operations is based on the names and types of their parameters. The computation of parameter-list similarity is insensitive to the order of parameters. For non-return parameters, if none of the two operations is overloading, the matchcount for a pair of same-name parameters is 1. If any of the two compared operations is overloading, the types of the two same-name parameters is further examined, in order to distinguish the overloading methods from each other, which often declare the same name parameters but with different parameter types. In the case of overloading, if the same-name parameters have the mapped types, their matchcount is 1; otherwise, their matchcount is set at 0.5. For the return parameters, if their types map, the matchcount is 1. Otherwise, it is set at 0. If the type of the return parameter of both operations is void, the matchcount for the return parameter is set at 0.

The similarity of the *initValue* of two compared attributes is computed in the same way as the outgoing usage similarity between two operations. The *initValue*-similarity value is added to the overall matchcount of the *Usage*_{<<write>>} similarity between two attributes.

Take the operations `Customer.statement()`₂₃ and `Customer.plainStatement()`₂₇ as an example. Let us assume that *UMLDiff* has identified the matched model elements and is in the process of identifying renamings. It collects [`Customer.statement()`₂₃, `Customer.plainStatement()`₂₇] as a pair of renaming candidates. Table 3-4 shows the two related-element sets of `Customer.statement()`₂₃ and `Customer.plainStatement()`₂₇. Note that all the incoming and outgoing usages of these two operations, except for `Customer.statement()`₂₃ calling `String.valueOf(double)`₂₃ twice, happen to be one. We omit the *count* tag attached to such usage dependencies. If a usage dependency appears more than once, it is indicated at the end of the usage dependency, such as `String.valueOf(double)` [2]. In the case of comparing `Customer.statement()`₂₃ and `Customer.plainStatement()`₂₇, the similarity of their parameters is one, their incoming usage similarity is also one, and their outgoing usage similarity is zero.

Table 3-4. The related model-element sets of `Customer.statement()`₂₃ and `Customer.plainStatement()`₂₇

| Type of relations | | Customer.statement() | Customer.plainStatement() |
|-------------------|-------------|--|---------------------------|
| Parameter | | return : String | return : String |
| Outgoing | read | Customer. rentals | null |
| | write | null | null |
| | call | Customer.getName() Customer.getTotalCharge() Customer.getTotalFreq...Points() Rental.getMovie() Rental.getCharge() Movie.getTitle() String.valueOf(double) [2] String.valueOf(int) Vector.elements() Enumeration.hasMoreElements() Enumeration.nextElement() | PlainStatement.value() |
| | instantiate | null | PlainStatement |
| Incoming | call | Vids.main(String[]) | Vids.main.String[]) |

When computing incoming and outgoing usage similarity between two operations, if the two compared operations are related to some other model elements but the intersection of the two related-element sets is empty, such as the case for the outgoing usage of `Customer.statement()`₂₃ and `Customer.plainStatement()`₂₇, *UMLDiff* proceeds to

compute the transitive usage similarity between the two compared operations, if its *ConsiderTransclosureUsageSimilarity* parameter is set to true. The transitive usage similarity takes into account the model elements related through the transitive-closure of the given relation, in addition to the directly related elements.

Table 3-5 shows the transitive outgoing usage of `Customer.statement()`₂₃ and `Customer.plainStatement()`₂₇. The transitive usage similarity is still computed as per the above structure-similarity equation, but without considering the *count* tag. The matchcount, addcount, and removecount for the transitive outgoing usage similarity between `Customer.statement()`₂₃ and `Customer.plainStatement()`₂₇ is 23, 6, and 0 respectively. Thus, the transitive outgoing usage similarity is $23/(23+6+0)=0.79$.

Table 3-5. The transitive outgoing usage of `Customer.statement()`₂₃ and `Customer.plainStatement()`₂₇

| Type of relations | | Customer.statement() | Customer.plainStatement() |
|-------------------|-------------|---|---|
| Outgoing | read | Customer._rentals Customer._name Movie._title Movie._daysRented Price._price Rental._movie | Customer._rentals Customer._name Movie._title Movie._daysRented Price._price Rental._movie |
| | write | null | null |
| | call | omit 17 matched operations | omit 17 matched operations PlainStatement.value PlainStatement.headerString() PlainStatement.each...String() PlainStatement.footerString() Customer.getRentals() |
| | instantiate | null | PlainStatement |

Determining the similarity when both related model element sets are empty is challenging. This case arises, for example, when the operations are not called by any other operations. In such cases, setting the structure similarity to be by default 0 or 1 is not desirable: without any explicit evidence of similarity, to assume that the structure is completely the same or completely different may skew the subsequent result. Therefore, in such cases, *UMLDiff* uses the name similarity with an increasing exponent. The effect is dampened as more empty sets are encountered. For example, when computing the

structure similarity of two operations in the order of their parameter-list, outgoing usage and incoming usage similarities, if the two compared operations declare no parameters, have return type void, and have no outgoing and incoming usage dependencies, *UMLDiff* returns *name-similarity*¹ for comparing parameter-list similarity, *name-similarity*² for outgoing usage similarity, and *name-similarity*³ for incoming usage similarity.

3.3.3.3 Overall similarity assessment

Given two model elements e_1 and e_2 of the same type, their overall similarity metric, used for determining potentially renamed and moved model elements, is computed according to the following equation:

SimilarityMetric = (lexical-similarity + Σ_N structure-similarity) / (lexical-similarity + N), where lexical-similarity = name-similarity + comment-similarity, and N is the number of different types of structure similarities computed for a given type of model elements as defined in Table 3-3.

The value of Σ_N structure-similarity is adjusted in the following cases. When comparing two operations, if anyone of them is overloaded, Σ_N structure-similarity is multiplied by the parameter-list similarity of the compared operations in order to distinguish the overloading operations from each other, which often have similar usage dependencies but with different parameters. Furthermore, when determining the potential moves of attributes and operations, if the declaring classes/interfaces of the compared attributes/operations are not related through inheritance, containment, or usage relations, the value of Σ_N structure-similarity is multiplied by the overall similarity metrics of the classes in which the compared attributes/operations are declared and then divided by the product of the numbers of all the not-yet-mapped model elements with the same name (same identifier for operation) and type as the two compared elements. This is designed to improve the precision when identifying attribute and operation moves.

UMLDiff uses two user-defined thresholds (*RenameThreshold* and *MoveThreshold*): two model elements are considered as the “same” element renamed or moved when their overall similarity metric is above the corresponding threshold. If, for a given element in the “before” version, there are several potential mappings above the user-specified threshold in the “after” version, the one with the highest similarity score is chosen. The

higher the threshold is, the stricter the similarity requirement is. The smaller the threshold is, the riskier the reported renamings and moves are.

3.3.4 Mapping model elements

Table 3-6. The partial model-element sets V_{23} and V_{27}

| Version23 : Top-level subsystem | | | Version27 : Top-level subsystem | | |
|---------------------------------|------------------------|--------------------|---------------------------------|------------------------|-------------------------|
| Element | Children | | Element | Children | |
| Version23 | Subsystem | null | Version27 | Subsystem | null |
| | Package | default | | Package | default |
| | DataType | String[] | | DataType | String[] |
| default | Package | null | default | Package | null |
| | Class | Vids | | Class | Vids |
| | | Rental | | | Rental |
| | | Movie | | | Movie |
| | | Customer | | | Customer |
| Interface | null | Interface | HTMLStatement | | |
| Vids | Class | null | Vids | Class | null |
| | Interface | null | | Interface | null |
| | Operation | main(String[]) | | Operation | main(String[]) |
| | Operation _c | null | | Operation _c | null |
| | Attribute | null | | Attribute | null |
| Customer | Class | null | Customer | Class | PlainStatement |
| | Interface | null | | Interface | null |
| | Operation | getName() | | Operation | getName() |
| | | getTotalCharge() | | | getTotalCharge() |
| | | getTotalPoints() | | | getTotalPoints() |
| | | htmlStatement() | | | htmlStatement() |
| | | <i>statement()</i> | | | <i>plainStatement()</i> |
| | | | | | <i>getRentals()</i> |
| | Operation _c | Customer(String) | | Operation _c | Customer(String) |
| | Attribute | name | | Attribute | name |
| rentals | | rentals | | | |

Given two versions, “before” and “after”, of a system model and their corresponding directed graphs $G_{before}(V_{before}, E_{before})$ and $G_{after}(V_{after}, E_{after})$, *UMLDiff* starts with the original vertex sets V_{before} and V_{after} that contain all the model elements and the initially empty mapped element set. After it finishes mapping the model elements, the mapped element set contains all the identified matched, renamed, and moved model elements, and

the V_{before} contains all the elements that have been removed and the V_{after} contains all the elements that have been added when the system model evolves.

Table 3-6 presents the partial model-element sets, V_{23} and V_{27} , of our running example, organized according to their containment hierarchy. “null” entries indicate that there is no model element of a given type contained in a particular model element. In the remainder of this subsection, we present how *UMLDiff* identifies same-name (i.e., matched), renamed, moved model elements using the running example presented in Section 3.3.1.

3.3.4.1 “Matched” elements

UMLDiff assumes that enough model elements remain “matched” between two compared versions of the system, which serve as the “initial landmarks” set for recognizing renamed and moved elements. The term “matched” refers to two corresponding model elements, of the same UML type, contained in a pair of mapped elements, which have the same names, although their children, attributes, and relations with other elements may be different.

The very first step of *UMLDiff* is to identify as many matched model elements as possible. It starts at the top-level subsystems of the two compared versions of the system model, which are always assumed to match. The pair of the matched top-level subsystems is added into the mapped element set as the first pair of mapped elements. *UMLDiff* then progresses along the containment hierarchy of the models, moving from one logical level to the next at the same time, from subsystems, to packages, classes and interfaces, and finally attributes and operations. Given a pair of mapped model elements of the current logical level in the mapped element set, *UMLDiff* identifies all their children of the same type with same names, adds them to the mapped element set as new pairs of matched elements, and removes them from the set V_{before} and V_{after} respectively. The pairs of matched children may be of the current logical level or one level below. The process continues until there are no more unprocessed pairs of matched elements of the current logical level in the mapped element set and *UMLDiff* then progresses down to the next logical level.

Consider, for example, our running example. The matched model elements are of regular font and left justified in Table 3-6. In this example, given the matched top-level

subsystems, *UMLDiff* adds their contained same-name default packages into the mapped element set. Next, it maps the four same-name classes contained in the default package. Given the matched class Customer, it maps the same-name attributes, constructor, and operations it declares. Note that there is no mapped nested model element in this simple running example. *UMLDiff* proceeds directly from the subsystem, to the package level, to the class/interface level, and finally to the attribute/operation level.

UMLDiff may not recover all the pairs of the matched model elements in this round: same-name, same-type model elements contained in renamed and moved parent elements are also considered as matched, but, at this stage, the renamed and moved model elements have not yet been recovered. As the pairs of renamed or moved elements are added to the mapped element set, *UMLDiff* attempts again to identify the same-name, i.e. matched, children they contain recursively, starting at the given pair of renamed/moved model elements. The only difference is that, instead of traversing the whole containment hierarchy from the top-level subsystem, it traverses the subtree of the containment hierarchy rooted at the given pair of renamed or moved model elements.

3.3.4.2 Renamed⁴ elements

After *UMLDiff* has completed its recognition of matched model elements, it proceeds to recover the renamed model elements. *UMLDiff* only considers potential renamings within the context of two mapped elements, such as the renaming of an operation within a mapped class. Identifying renamings between two arbitrary elements of the same type, such as the renaming of an operation that was moved from one class to another and then had its identifier renamed, is computationally expensive, since it requires the comparison of all the pairs of not-yet-mapped model elements of the same type. Again, *UMLDiff* starts at the matched top-level subsystems of the two compared versions of the system model and it traverses all the mapped model elements along the containment-spanning trees of the compared model graphs to identify pairs of renamed elements, moving from

⁴ The renamings of operations include the changes to their identifiers and/or parameter lists. Furthermore, *UMLDiff* does not consider parameter renamings.

one logical level to the next when it has completed traversing all the model elements of the current logical level in both spanning trees.

Note that *UMLDiff* may not recover all the pairs of the renamed model elements in its first round of recognizing the renamed elements due to two reasons. It may miss the pairs of renamed elements because their related elements have undergone renamings and/or moves as well. Some of these misses may be recovered in the following rounds of renaming identification (see Section 3.3.4.4). Furthermore, renamed model elements may be contained within moved elements or other not-yet-identified renamed elements. Once the pairs of such elements have been added to the mapped element set, *UMLDiff* attempts again to identify the pairs of renamed model elements they contain recursively starting at the given pair of moved or newly-identified renamed model elements.

Given a pair of mapped model elements of the current logical level in the mapped element set, *UMLDiff* first collects all their not-yet-mapped children of the same type and formulates sets of renaming candidate pairs. Suppose there are N not-yet-mapped elements of a particular type contained in the “before” version of the mapped elements and M in the “after” version: *UMLDiff* generates N sets of renaming candidate pairs, each of which contains M pairs. It then identifies the renamed model elements based on their lexical and structure similarities, adds the newly identified pairs of renamed elements to the mapped element set, and removes them from the V_{before} and V_{after} sets. Adding a pair of renamed elements to the mapped element set triggers *UMLDiff* to recursively recognize the matched descendants they contain. The pairs of newly identified renamed and matched children may be of the current logical level or one level below. The process continues until there are no more unprocessed pairs of mapped elements of the current logical level in the mapped element set and *UMLDiff* then progresses down to the next logical level.

For example, when comparing the version 23 and 27 of our running example, the matched top-level subsystems contain only a pair of matched default packages. The matched default packages contain four matched classes and one not-yet-mapped class, `HTMLStatement27`. Thus, at this point, there are no potential renaming candidate pairs. However, when *UMLDiff* reaches the matched class `Customer`, it collects the following not-yet-mapped children: operation `Customer.statement()23`, and operations

Customer.plainStatement()₂₇ and Customer.getRentals()₂₇ and nested class PlainStatement₂₇. Since there are no not-yet-mapped nested classes contained in the class Customer₂₃, there is no need yet for renaming identification of Customer's nested classes. However, *UMLDiff* formulates one set of operation-renaming candidate pairs (italic font and right justified in Table 3-6 and Table 3-8), which contains two pairs of renaming candidates: [statement()₂₃, plainStatement()₂₇] and [statement()₂₃, getRentals()₂₇].

The overall similarity of each of these pairs is computed according to the equations shown in Section 3.3.3.3, based on their lexical and structure similarities. When comparing Customer.statement()₂₃ and Customer.plainStatement()₂₇, *UMLDiff* computes three types of structure similarity between them, i.e. parameter list, outgoing usage, and incoming usage, which are 1, 0.793, 1 respectively. Their identifier similarity using Word-LCS⁵ is 0.5. Thus, their overall similarity metric is 0.941. The overall similarity metric between [statement()₂₃, getRentals()₂₇] is similarly computed to be 0.139. Thus, plainStatement()₂₇ is much more similar to statement()₂₃ than getRentals()₂₇. Assuming that the *RenameThreshold* is less than 0.941, the pair [statement()₂₃, plainStatement()₂₇] is recognized as an instance of operation renaming.

Table 3-7. The sets of renaming candidate pairs

| HTMLStatement | | | |
|--------------------------------------|------------------------|---|----------------------|
| header? ⁶ () - ?Header() | footer?() - ?Header() | each...?() - ?Header() | value() - ?Header() |
| header?() - ?Footer() | footer?() - ?Footer() | each...?() - ?Footer() | value() - ?Footer() |
| header?() - ?Each...() | footer?() - ?Each...() | each...?() - ?Each...() | value() - ?Each...() |
| Customer | | | |
| getTotalCharge() - getAllCharge() | | getTotal...Points() - getAllCharge() | |
| getTotalCharge() - getAll...Points() | | getTotal...Points() - getAll...Points() | |

Let us now look at the versions 27 and 28 of our running example. They involve much more complex changes, including many renamings and moves. Let us first examine

⁵ Word-LCS is used for all the lexical-similarity computation in Section 3.3.4.

⁶ Replace the suffix "String" and the prefix "print" with "?" to fit these operations in table.

renamings. Similar to the comparison of versions 23 and 27, *UMLDiff* first identifies all the matched model elements starting from the top-level subsystems. Then it proceeds to collect potential renaming candidates and to formulate the sets of renaming candidate pairs, such as those shown in Table 3-7 for the matched class `HTMLStatement` and `Customer`.

Note that `HTMLStatement.value()`₂₇ is collected as a renaming candidate at this stage of *UMLDiff* process. It is compared against three `HTMLStatement.printXXX()`₂₈ operations but it is not found similar to anyone of them; therefore, it will be finally collected as one of the potential move candidates (bold font and left justified in Table 3-8). Furthermore, the abstract operations, such as those of the class `Statement`, are not collected as renaming or move candidates: since they have no outgoing usage, the identification of their renamings or moves tends to be error-prone. *UMLDiff* ignores them in its renaming and move identification process. However, the renamings of the abstract operations may be recovered by propagating the knowledge of the identified renamings of their implementation operations along the inheritance hierarchy as discussed in Section 3.3.4.5. Finally, at this stage, the not-yet-mapped operations of the `PlainStatement` class are not collected as renaming candidates, since the move of the class `PlainStatement` has not yet been identified. However, they will be processed when the move of `PlainStatement` is identified and added to the mapped element set.

UMLDiff computes the overall similarity metrics of all the pairs of renaming candidates contained in a given pair of mapped model elements and selects the pair with the highest similarity metric (above the *RenameThreshold*) to be added to the mapped element set as a renaming. It then removes from the candidate sets all other pairs that contain the elements of the selected pair. This process continues until there is no pair left.

For example, if the *UMLDiff RenameThreshold* parameter is 0.3, then all 12 pairs of operation-renaming candidates of the matched `HTMLStatement` class have sufficiently high similarity metric to be qualified for further examination. The pair [`eachRentalString()`₂₃, `printEachRental()`₂₇] ranks highest and is selected as a pair of renamed elements; then all other pairs that contain either `eachRentalString()`₂₃ or `printEachRental()`₂₇ are removed from the list. *UMLDiff* then selects the pair with the highest similarity-metric value in the current list until the pair list is empty.

3.3.4.3 Moved elements⁷

Finally, *UMLDiff* proceeds to examine those model elements that have not yet been identified as matches or renamings and to consider whether they may have been moved from one part of the system to another. It first starts at the top-level subsystem of V_{before} and traverses all the not-yet-mapped model elements along the containment hierarchy of the model, moving from one logical level to the next when there are no more unprocessed elements of the current logical level. Thus, *UMLDiff* first identifies all the potential subsystem moves, and then progresses down to package moves, class and interface moves, and finally attribute and operation moves.

When it encounters a not-yet-mapped model element e_{before} , *UMLDiff* collects all the not-yet-mapped same-type and same-name (same-identifier for operation) model elements in V_{after} and forms a set of move candidate pairs, if such elements exist. It then computes the overall similarity metrics for all these candidate pairs, selects the pair with the highest similarity metric (above the *MoveThreshold*), and adds it to the mapped element set as a pair of moved model elements. Adding a pair of moved elements to the mapped element set triggers *UMLDiff* to recursively recognize their matched and renamed descendants. This process continues until all the not-yet-mapped model elements of the current logical level have been processed; then *UMLDiff* proceeds to identify the potential moves at one logical level below.

Note that for operations, *UMLDiff* uses their identifiers instead of their full signatures to collect move candidates, which enables the identification of changes involving operation moves with simultaneous parameter-list modifications. Furthermore, the set of not-yet-mapped elements may change as the process goes on, because the descendants of the newly identified moved elements might be identified as matches and renamings when the pairs of moved elements are added to the mapped element set, as discussed for the moved `PlainStatement` class below. Finally, the identified moved elements are only removed from V_{before} and V_{after} after the whole move recognition step is complete. After all the not-yet-mapped elements in V_{before} have been processed, *UMLDiff* starts at the top-

⁷ *UMLDiff* does not consider moves of constructors (i.e., model elements of UML type `Operation<<create>>`), since it makes no sense to do so.

level subsystem of V_{after} and performs the same tasks as above. This step, together with not-immediately-remove-moved-elements-from-element-sets, enables *UMLDiff* to identify many-to-one and one-to-many mapping between moved elements.

Table 3-8. The initial not-yet-mapped model elements after the match/renaming reorganization steps

| Version27 : Top-level subsystem | | | Version28 : Top-level subsystem | | | | |
|---------------------------------|------------------------|------------------------|---------------------------------|------------------------|-----------------------|-----------------------|--|
| Element | Children | | Element | Children | | | |
| Version27 | Subsystem | null | Version28 | Subsystem | null | | |
| | Package | null | | Package | null | | |
| | DataType | null | | DataType | null | | |
| default | Package | null | default | Package | null | | |
| | Class | null | | Class | PlainStatement | | |
| | Interface | null | | Interface | null | | |
| Customer | Class | PlainStatement | Customer | Class | null | | |
| | Interface | null | | Interface | null | | |
| | Operation | null | | Operation | null | | |
| | Operation _c | null | | Operation _c | null | | |
| | Attribute | null | | Attribute | null | | |
| PlainStmt | Class | null | PlainStmt | Class | null | | |
| | Interface | null | | Interface | null | | |
| | Operation | <i>headerString()</i> | | | Operation | <i>printHeader()</i> | |
| | | <i>each...String()</i> | | | | <i>printEach...()</i> | |
| | | <i>footerString()</i> | | | | <i>printFooter()</i> | |
| | | value() | | | | | |
| | Operation _c | null | | Operation _c | null | | |
| Attribute | null | Attribute | null | | | | |
| HTMLStmt | Class | null | HTMLStmt | Class | null | | |
| | Interface | null | | Interface | null | | |
| | Operation | value() | | Operation | null | | |
| | Operation _c | null | | Operation _c | null | | |
| | Attribute | null | | Attribute | null | | |
| | | | Statement | Class | null | | |
| | | | | Interface | null | | |
| | | | | Operation | value() | | |
| | | | | Operation _c | null | | |
| | | | | Attribute | null | | |

Table 3-8 lists all the remaining not-yet-mapped model elements that are still in V_{27} and V_{28} of our running example, after *UMLDiff* has completed the match and renaming recognition steps. Note that the three abstract operations of the Statement class are not listed in Table 3-8, since *UMLDiff* does not consider the moves of the abstract operations,

which tends to be error-prone due to the non-existence of outgoing usage dependency from them. The top-level subsystem, Version₂₇, and its default₂₇ package do not contain any not-yet-mapped subsystems or packages. Thus, *UMLDiff* proceeds to the class/interface logical level. When traversing the classes and interfaces, it encounters a not-yet-mapped class Customer.PlainStatement₂₇ (bold font and left justified in Table 3-8 for move candidates). *UMLDiff* then searches the remaining not-yet-mapped classes contained in V_{28} and retrieves all the classes with the same name. It finds the not-yet-mapped class PlainStatement₂₈ in the default₂₈ package. Given the moving candidates [Customer.PlainStatement₂₇, default.PlainStatement₂₈], *UMLDiff* computes their similarity metric to be 0.6. If the *MoveThreshold* is lower than 0.6, the pair of [Customer.PlainStatement₂₇, default.PlainStatement₂₈] is added to the mapped element set as a moved class.

Adding the moved class [Customer.PlainStatement₂₇, default.PlainStatement₂₈] to the mapped element set triggers *UMLDiff* to recognize the matched and renamed descendants they contain. The class PlainStatement has no matched children, but it has four and three not-yet-mapped operations in version 27 and 28 respectively. *UMLDiff* collects them as renaming candidates and identifies three operation renamings (italic font and right justified in Table 3-8) that are added to the mapped element set.

After processing the class PlainStatement, there aren't any not-yet-mapped classes or interfaces and *UMLDiff* proceeds to the attribute/operation level. It encounters the not-yet-mapped operation HTMLStatement.value()₂₇ and retrieves from Version₂₈ the not-yet-mapped operation Statement.value()₂₈. *UMLDiff* computes the overall similarity of the candidate move pair [HTMLStatement.value()₂₇, Statement.value()₂₈] to be 0.71. The [HTMLStatement.value()₂₇, Statement.value()₂₈] pair is added to the mapped element set as a moved operation, assuming that the *MoveThreshold* is lower than 0.71. After that, *UMLDiff* encounters the not-yet-mapped operation PlainStatement.value()₂₇. Similarly to HTMLStatement.value()₂₇, the pair [PlainStatement.value()₂₇, Statement.value()₂₈] is also identified as an operation move. Note that the operations headerString(), footerString(), and eachRentalString() of PlainStatement are not encountered as not-yet-mapped elements: they are identified as operation renamings when the move of PlainStatement class is recognized, which results in them being removed from the initial remaining not-yet-mapped model

element sets. After processing all the not-yet-mapped elements in V_{27} , *UMLDiff* starts over at the top-level subsystem of V_{28} , which contributes no more moves in this running example.

When examining attribute/operation move candidates, if their declaring classes (interfaces) are related through inheritance, containment, or usage relations, their non-adjusted structure similarities (see Section 3.3.3.3) are used in the computation of their overall similarity. Otherwise, *UMLDiff* computes the overall similarity metric of their declaring classes (parent-similarity) and calculates the product (amount-potential-moves) of the numbers of the not-yet-mapped, same-type, same-name model elements as the two compared elements in the two compared versions. In this case, the structure similarity of the two compared attributes/operations is adjusted as $\Sigma_N \text{structure-similarity} * \text{parent-similarity} / \text{amount-potential-moves}$. Intuitively, if the source and target classes of the moved attribute/operation have no special relation, *UMLDiff* takes into account the contexts from and to which the attributes/operations move: they must be similar enough in order for the moves of attributes/operations to make sense. Furthermore, the more the potential moves of the same kind are, the less likely it is that any of them will be recognized as a valid move.

Take the [HTMLStatement.value()₂₇, Statement.value()₂₈] as an example: since Statement is the superclass of HTMLStatement, the original structure similarity 1.83 is used to compute the overall similarity metric, which is 0.71. Let us assume that there is no special relation between HTMLStatement and Statement. The overall similarity metric (parent-similarity) of [HTMLStatement₂₇, Statement₂₈] is 0.7 and there are two potential moves of the value() operation. Thus, the structure similarity of [HTMLStatement.value(), Statement.value()] that is used to compute the overall similarity metric becomes $1.83 * 0.7 / 2 = 1.3$, which brings the overall similarity metric down to 0.58.

This technique is designed to improve the precision of attribute/operation moves. For example, in an interactive system, many classes implement the ActionListener interface and its actionPerformed() operation; in general, these implementations handle different user actions and are used in different contexts. However, when some actionPerformed() method disappears (usually because its class is removed or has stopped implementing the ActionListener interface) and new ones appear between two compared versions of a model,

they tend to be reported as pairs of moves, which usually does not make sense. The *UMLDiff* algorithm integrates the above technique to filter out such moves.

3.3.4.4 Propagating knowledge of identified renamings and moves along usage dependency

UMLDiff computes the structure similarity of two compared model elements in terms of the intersection of their two related-element sets. It is sensitive to the order that a set of renamed and/or moved model elements are examined, which may result in some renamings and moves being missed during a particular round of renaming/move identification. On the other hand, the more renamings and moves *UMLDiff* recovers, the larger the current “landmarks” set (i.e., the mapped element set) becomes, and the more likely it becomes that *UMLDiff* may recover further related renamings and moves.

Let us look at versions 27 and 28 of our running example. The operations `Customer.getTotalCharge()`₂₇ and `Customer.getTotalFrequentRenterPoints()`₂₇ and their caller operation `HTMLStatement.footerString()`₂₇ and `PlainStatement.footerString()`₂₇ are renamed to `Customer.getAllCharge()`₂₈, `Customer.getAllFrequentRenterPoints()`₂₈, `HTMLStatement.printFooter()`₂₈ and `PlainStatement.printFooter()`₂₈ respectively.

First, the renaming candidate pair [`PlainStatement.footerString()`₂₇, `PlainStatement.printFooter()`₂₈] is examined after the `PlainStatement` class has been recognized as moved. Furthermore, the renaming candidates [`Customer.getTotalCharge()`₂₇, `Customer.getAllCharge()`₂₈] may be compared before [`HTMLStatement.footerString()`₂₇, `HTMLStatement.printFooter()`₂₈]; even if the order is reverse, the renaming of [`HTMLStatement.footerString()`₂₇, `HTMLStatement.printFooter()`₂₈] may not be recovered if the *RenameThreshold* is greater than 0.5 (see below). Therefore, at the time of determining the mapping between [`Customer.getTotalCharge()`₂₇, `Customer.getAllCharge()`₂₈], their incoming usage relations may be substantially different to *UMLDiff* and their incoming usage similarity may be 0. However, the operations [`getTotalCharge()`₂₇, `getAllCharge()`₂₈] declare the same parameters and they use the same sets of other model elements; their parameter-list similarity and outgoing usage similarity are 1, which brings their overall similarity to 0.714, which is sufficiently high. Thus, even without the knowledge of the renaming [`HTMLStatement.footerString()`₂₇, `HTMLStatement.printFooter()`₂₈] and

[PlainStatement.footerString()₂₇, PlainStatement.printFooter()₂₈], getTotalCharge()₂₇ and getAllCharge()₂₈ may still be recovered as a renamed pair, at a fairly high *RenameThreshold*. The case of the renaming candidates [getTotalFrequentRenterPoints()₂₇, getAllFrequentRenterPoints()₂₈] is similar.

On the other hand, the outgoing usage similarity of [HTMLStatement.footerString()₂₇, HTMLStatement.printFooter()₂₈] is 0.33, when the renaming pairs [getTotalCharge()₂₇, getAllCharge()₂₈] and [getTotalFrequentRenterPoints()₂₇, getAllFrequentRenterPoints()₂₈] have not yet been recovered, but it increases to 1 if these pairs have already been established as renamings before *UMLDiff* considers the renaming candidate pair [HTMLStatement.footerString()₂₇, HTMLStatement.printFooter()₂₈]. The corresponding overall similarity metric of the pair [HTMLStatement.footerString()₂₇, HTMLStatement.printFooter()₂₈] increases from 0.5 to 0.7, which may push the pair above the *RenameThreshold*.

It is computationally expensive to keep track of all related not-yet-mapped model elements. Furthermore, as shown in the example, it is not necessary to update the similarity metric of two not-yet-mapped model elements as each of its related renamings and/or moves is recovered. For example, we only need to re-compute the similarity metric of the renaming candidate pair [HTMLStatement.footerString()₂₇, HTMLStatement.printFooter()₂₈] once after both the renamings of [getTotalCharge()₂₇, getAllCharge()₂₈] and [getTotalFrequentRenterPoints()₂₇, getAllFrequentRenterPoints()₂₈] are identified. Therefore, at the end of each round of renaming and move identification, *UMLDiff* collects the pairs of not-yet-mapped renaming and move candidates that are related, through usage dependencies, to the newly identified renamed and moved model elements in the last round and updates their similarity metrics to see if they may be qualified this time.

UMLDiff may be configured to perform up to *MaxRenameRound* and *MaxMoveRound* of renaming and move recognition or to continue until there is no more affected potential renaming and move candidates that are related to the new instances of renamings and moves identified in the last round. Allowing multiple rounds of renaming and move identification relieves the impact of the order of the model elements being processed by *UMLDiff* on its final mapping results.

3.3.4.5 Propagating identified operation renamings along inheritance hierarchy

When the move of PlainStatement class is identified, *UMLDiff* attempts to recover its operation renamings (see Section 3.3.4.3). When the pair [PlainStatement.footerString()₂₇, PlainStatement.printFooter()₂₈] is examined, the related operation renamings [getTotalCharge()₂₇, getAllCharge()₂₈] and [getTotalFrequentRenterPoints()₂₇, getAllFrequentRenterPoints()₂₈] have already been identified. However, the overall similarity of the pair [PlainStatement.footerString()₂₇, PlainStatement.printFooter()₂₈] is 0.475, still not sufficiently high, in comparison with the similarity (0.7) of the operation pair [HTMLStatement.footerString()₂₇, HTMLStatement.printFooter()₂₈], since we intentionally introduced more changes to the PlainStatement.printFooter()₂₈ (see Section 3.3.1). It is therefore possible that the renaming of [PlainStatement.footerString()₂₇, PlainStatement.printFooter()₂₈] is not recognized when the renaming of [HTMLStatement.footerString()₂₇, HTMLStatement.printFooter()₂₈] is, if, for example, the *RenameThreshold* is 0.5.

However, *UMLDiff* knows that both HTMLStatement and PlainStatement extend the Statement class and their corresponding footerString() and printFooter() operations implement the abstract Statement.footerString() and Statement.printFooter() operations in the two compared versions. Implementing (or overriding) operations must have the same signature (i.e., the same identifier and parameter list) as the operations they implement (override). Therefore, if any one of them is renamed, all the rest must be renamed as well. Based on this definition, *UMLDiff* propagates the knowledge of the identified operation renamings along (both up and down) their implementation (overriding) hierarchy, which may result in recognizing the renamings of abstract operations (which are not explicitly compared) and the renamings of other implementation (overriding) operations, as yet missed.

For example, based on the identified renaming [HTMLStatement.footerString()₂₇, HTMLStatement.printFooter()₂₈], *UMLDiff* first searches up to the mapped top-most ancestor class or interface (Statement in this case) and collects the pair of not-yet-mapped operations (the abstract operations [Statement.footerString()₂₇, Statement.printFooter()₂₈]) that are implemented (or overridden) by the identified pair of renamed operations and asserts them as a pair of renamed operations. And then based on the recovered operation

renaming of the top-most ancestor class ([Statement.footerString()]₂₇, Statement.printFooter()]₂₈), *UMLDiff* searches down all the pairs of the not-yet-mapped operations ([PlainStatement.footerString()]₂₇, PlainStatement.printFooter()]₂₈) that implement (override) them and asserts all of them as pairs of renamed operations.

3.3.5 Mapping relations

In Section 3.3.4, we discussed the *UMLDiff* process for mapping the elements of two UML models corresponding to two versions of an evolving software system. This process produces three sets that contain (a) the model elements for which mappings have been identified in the two compared versions (i.e., matched, renamed, and moved), (b) the removed elements, and (c) the newly introduced elements respectively. *UMLDiff* then proceeds to map the relations between the model elements, i.e., to map the edge set (E_{before} , E_{after}) of the model graphs. This process step also produces three relation sets that contain (a) the matched relations between the two model elements, (b) the removed relations, and (c) the newly introduced relations respectively.

The UML relations are defined by their types (see Table A-2 and Table A-3) and the UML model elements they relate. Given two model elements (v_{before} , v_{after}), where $v_{after} = \text{null}$ if v_{before} is removed and $v_{before} = \text{null}$ if v_{after} is added, *UMLDiff* collects all their relations in the two compared models. Two same-type relations of the model elements v_{before} and v_{after} in the two compared versions are matched, if the model elements they relate are contained in the mapped model element set, i.e., they map to each other. After *UMLDiff* finishes comparing the relations of all the pairs of the model elements, all unmatched relations that are still contained in E_{before} are assumed to have been *removed* and all unmatched relations in E_{after} are assumed to have been *added* when system evolves from the “before” version and the “after”.

Note that the removed model elements contained in ($V_{before} - V_{after}$) and the newly added model elements contained in ($V_{after} - V_{before}$) have no counterpart in the compared models. The relations from (to) a removed model element are all removed and the relations from (to) an added model element are all newly added. For a pair of mapped elements (v_{before} , v_{after}), they may have matched, newly added, and/or removed relations. A removed (added) relation between the two model elements does not indicate that any of

the elements it relates has been removed (added). For usage dependencies, *UMLDiff* also compares their *count* tag and reports the changes to the number of times they appear between the model elements.

**Table 3-9. Mapping relations of the renamed
[Customer.statement()₂₃, Customer.plainStatement()₂₇]**

| Version23 | | Version27/Version28 | |
|---------------------------|--|---------------------------|----------------------------------|
| Relation type | Instances of relation | Relation type | Instances of relation |
| owner – feature | Customer – statement() | owner – feature | Customer – plainStmnt() |
| Usage<<read>> | statement(), _rental | Usage<<read>> | null |
| Incoming Usage<<call>> | Vids.main(), statement() | Incoming Usage<<call>> | Vids.main(), plainStmnt() |
| Outgoing Usage<<call>> | statement(), getName() statement(), getTotalCh...() statement(), getTotalFr...() statement(), getMovie() statement(), getCharge() statement(), getTitle() statement(), valueOf(d) [2] statement(), valueOf(int) statement(), elements() statement(), hasMore...() statement(), nextEle...() | Outgoing Usage<<call>> | <u>plainStmnt(), value()</u> |
| Usage<<instantiate>> | null | Usage<<instantiate>> | <u>plainStmnt(), PlainStmnt</u> |
| Usage<<write>> | null | Usage<<write>> | null |
| Usage<<send>> | null | Usage<<send>> | null |
| Parameter | statement() – return:Str | Parameter | <u>plainStmnt() – return:Str</u> |
| raisedSignal | null | raisedSignal | null |
| Reception | null | Reception | null |

Table 3-9 lists all the relations of the renamed operation [Customer.statement()₂₃, Customer.plainStatement()₂₇], grouped according to their types. Consider the incoming-call relation as an example: the statement()₂₃ and plainStatement()₂₇ operations are called by the operations Vids.main(String[])₂₃ and Vids.main(String[])₂₇ respectively, which are matched. Thus, the renamed operation [Customer.statement()₂₃, Customer.plainStatement()₂₇] has a matched (regular font and right justified in Table 3-9) incoming-call relation. Similarly, they have a matched composition relation (both are declared in the matched class Customer) and a matched [*BehaviorFeature* – *parameter*] relation (both declare a return parameter of type String). All the removed relations are highlighted with strikethrough lines, while all the newly introduced relations are underlined with dash lines. The

renamed operation [Customer.statement()₂₇, Customer.plainStatement()₂₈] no longer uses the attribute Customer._rentals (a removed *Usage*_{<<read>>} relation), but the attributes Customer._rentals exist in both versions 23 and 27, i.e., they are matched. Furthermore, two (indicated by [2] at the end of the usage dependency) operation calls to String.valueOf(double) are removed (a removed outgoing *Usage*_{<<call>>} relation with tag *count=2*) when Customer.statement()₂₃ evolves to Customer.plainStatement()₂₇.

3.3.6 Recognizing behavior redistribution

Developers, sometimes, redistribute the behavior in the system in order to reorganize the inheritance hierarchy, restructure the usage dependencies between objects, or refactor a long method. After *UMLDiff* finishes mapping the model elements and their relations, it attempts to detect the redistribution of the behavior among operations, by analyzing the removals and additions of *Usage*_{<<read>>/<<write>>/<<call>>/<<instantiate>>} dependencies of the mapped operations and the related removed or added operations along their transitive usage and generalization/abstraction relations.

Behavior redistribution is reported in terms of *Extract operation* and *Inline operation* changes. Note that our concepts of *Extract operation* and *Inline operation* are broader than the *Extract Method* and *Inline Method* introduced in Fowler's refactoring catalog [32], which are limited to refactoring only class internals.

We discuss in detail how *UMLDiff* detects *operation extraction – operation inlining* is detected in the same manner. Given two mapped operations [O_{before} , O_{after}] with some removed outgoing *Usage*_{<<read>>/<<write>>/<<call>>/<<instantiate>>} relations originated from O_{before} , *UMLDiff* identifies the candidate targets (O_{target}) of the behavior redistribution, as all the newly added operations that have a transitive relation with O_{after} through the relations of *Usage*_{<<call>>} (incoming and outgoing), *Generalization* (overriding or overridden), *Abstraction*_{<<realize>>} (implemented by), or a combination of them. We consider a removed outgoing usage relation [O_{before} , e_{before}] from O_{before} as equal (not equivalent to relation match) to a newly added relation [O_{target} , e_{after}] from one candidate target operation, if they are of the same type and the elements [e_{before} , e_{after}] have been mapped. If the set of the outgoing usage relations from a candidate target operation O_{target} is a subset of the removed outgoing usage relations from O_{before} , or their intersection set is

greater than the user-specific threshold, then *UMLDiff* asserts that o_{target} is extracted from O_{before} .

Table 3-10. Redistribute semantic behavior among operations

| Version23 | | Version28 | |
|----------------------|--|----------------------------------|---|
| Relation type | Instances of relation | Relation type | Instances of relation |
| Customer.statement() | | Customer.getRentals() | |
| Usage<<read>> | statement(), _rental | Usage<<read>> | getRentals(), _rental |
| Usage<<call>> | statement(), getName() statement(), getTotalCh...() statement(), getTotalFr...() statement(), getMovie() statement(), getCharge() statement(), getTitle() statement(), valueOf(d) statement(), elements() statement(), hasMoreElements() statement(), nextElement() statement(), valueOf(d) statement(), valueOf(int) | PlainStatement.printHeader() | |
| | | Usage<<call>> | printHeader(), getName() |
| | | PlainStatement.printFooter() | |
| | | Usage<<call>> | printFooter(), getAllCharge() printFooter(), getAllFreq...() printFooter(), Double.toString() printFooter(), Integer.toString() |
| | | Usage<<instantiate>> | printFooter(), Double printFooter(), Integer |
| | | PlainStatement.printEachRental() | |
| | | Usage<<call>> | printEach(), getMovie() printEach(), getCharge() printEach(), getTitle() printEach(), valueOf(d) |
| | | PlainStatement.value() | |
| | | Usage<<call>> | value(), elements() value(), hasMoreElements() value(), nextElement() value(), printHeader() value(), printFooter() value(), printEachRental() |

Let us now compare the versions 23 and 28 of our running example. *UMLDiff* identifies the renamed operation [Customer.statement()₂₃, Customer.plainStatement()₂₈] and reports the relation differences as shown in Table 3-9. Since the operation Customer.plainStatement() has not changed between versions 27 and 28, Table 3-9 reflects the relation changes between its versions 23 and 27 as well as its versions 23 and 28. Given the renamed operation [Customer.statement()₂₃, Customer.plainStatement()₂₈],

Customer.statement()₂₃ is *o_{before}* in this example and Customer.plainStatement()₂₈ is *o_{after}*, and they have some removed outgoing usage relations. *UMLDiff* then collects all the newly added operations (the candidate targets) that have transitive usage and/or generalization/abstraction relations with Customer.plainStatement()₂₈. The operation Customer.plainStatement()₂₈ calls Statement.value()₂₈, which calls Customer.getRentals()₂₈ and the three abstract operations Statement.printHeader()₂₈/printFooter()₂₈/printEachRental()₂₈, which are implemented by PlainStatement.printHeader()₂₈/printFooter()₂₈/printEachRental()₂₈ respectively. Furthermore, Statement.value()₂₈ are implemented by PlainStatement.value()₂₈. All these operations are newly introduced in version 28. Thus, the candidate targets of the behavior redistribution include Customer.getRentals()₂₈ and PlainStatement.printHeader()₂₈/printFooter()₂₈/printEachRental()₂₈/value()₂₈. Note that *UMLDiff* ignores the abstract operations, since they have no outgoing usage.

Table 3-10 lists the removed outgoing usages of the operation Customer.statement()₂₃ and the candidate target operations and their newly added outgoing usages, when the model evolves from version 23 to 28. The outgoing usage relations of the target operations Customer.getRentals()₂₈ and PlainStatement.printHeader()₂₈/printEachRental()₂₈ are the subset of the removed outgoing usage relations of the Customer.statement()₂₃. *UMLDiff* asserts that these three target operations have been extracted from the Customer.statement()₂₃. On the other hand, the intersection of the new outgoing usages of the target operations PlainStatement.value()₂₈/printFooter()₂₈ and the removed outgoing usages of the Customer.statement()₂₃ is not empty. Depending on the user-specific threshold, the target operations PlainStatement.value()₂₈/printFooter()₂₈ may or may not be asserted as being extracted from the Customer.statement()₂₃. Clearly, as the system evolved from version 23 to 28, the behavior of the operation Customer.statement()₂₃ has been redistributed and encapsulated in a separate strategy object in version 28, which is defined by the abstract class Statement and its two implementation classes PlainStatement and HTMLStatement.

3.3.7 Comparing attributes of mapped model elements

Finally, *UMLDiff* compares the inherent attributes and the tagged values of the mapped UML model elements. For the visibility attribute, *UMLDiff* reports the changes as either

of the following: (a) *up*: the access modifier has become less restrictive in the “after” version; (b) *down*: the access modifier has become more restrictive in the “after” version; and (c) *match*: visibility has not changed. The access modifiers can be private, package, protected, or public, in decreasingly restrictive order. For all other attributes and tagged values, *UMLDiff* simply reports whether they are of the same value or not. For example, *UMLDiff* reports that the *visibility* of the two renamed operations [`getTotalCharge()`₂₇, `getAllCharge()`₂₈] and [`getTotalFrequentRenterPoints()`₂₇, `getAllFrequentRenterPoints()`₂₈] has been changed *up* from package to public. Furthermore, the deprecation status of the matched operation [`Customer.plainStatement()`₂₇, `Customer.plainStatement()`₂₈] has been changed from false to true.

3.4 Evaluation

In this section, we used JFreeChart [126] as the subject system of an extensive case study that we conducted to evaluate the run-time performance and the effectiveness of *UMLDiff* algorithm. JFreeChart is a class library of a realistic size that has been under active development for a long time and has suffered a substantial amount of design changes. At the same time, it is of a manageable size, possible to inspect “manually” to establish the ground truth for the algorithm’s results. Table C-1 report the numbers of model element and relation facts extracted by JDevAn’s fact extractor during the system lifecycle. Table C-2 reports the summary of most interesting design changes (including the changes correctly identified by *UMLDiff* and the missed ones manually added through the inspecting session of *UMLDiff* results with the JDevAn tool) in the evolution of JFreeChart system, which serves as the ground truth, i.e., M_{actual} , for evaluating the effectiveness of *UMLDiff* algorithm and the impacts of various factors that can affect the *UMLDiff* quality.

3.4.1 *UMLDiff* effectiveness

First, let us report on the effectiveness of *UMLDiff* in identifying renamed, moved and otherwise changed mode elements on the basis of their lexical-similarity and structure-similarity to other entities that have been identified to be the “same” across the two compared system versions. In principle, the precision and recall metrics are used to

evaluate the quality of such tasks. In the *UMLDiff* context, given the total number of design changes that have occurred between two versions (M_{actual}) and the number of changes reported by *UMLDiff* (M_{reported}), precision is the percentage of the correctly reported changes $(M_{\text{actual}} \cap M_{\text{reported}})/M_{\text{reported}}$ and recall is the percentage of changes reported $(M_{\text{actual}} \cap M_{\text{reported}})/M_{\text{actual}}$.

Precision is easier to evaluate than recall. Once all pairs of successive versions are *UMLDiffed*, we manually inspected, with the support of JDevAn tool [125], the correctness of each instance of changes reported by *UMLDiff* against the JFreeChart source code, the accompanying Javadocs and source-code comments, and a textual change log shipped with each major release version. Table 3-11 and Table 3-12 summarize the changes reported by *UMLDiff* when the renaming and move threshold is set to 0.3 and 0.35 respectively. The second column of these two tables reports the numbers of changes of each type reported by *UMLDiff* for JFreeChart. The number of correctly identified changes is reported in the third column and the precision percentage is reported in the fourth column.

Recall is harder to assess since it requires knowledge about the total number of changes of each type that have actually occurred. To develop an intuition about how good the *UMLDiff* recall is, we first run *UMLDiff* with a very low threshold for renamings, i.e., 0.01. With such a low threshold, *UMLDiff* is very eager to recognize mode elements as renamed and thus we expected to collect all instances of renamings to use as the “set of actually renamed elements” to assess renaming recalls in other configurations. With the renaming threshold set at 0.01, *UMLDiff* reports 2945 instances of renamed elements; after inspecting each one of them, we established that 2154 are correctly identified. In addition, some missed instances of renamings have been manually added through the inspecting session of *UMLDiff* results with JDevAn tool. Finally, we obtained 2180 instances of renamed mode elements. At renaming threshold 0.3, *UMLDiff* reports 2077 correct renaming instances, i.e. adding up the number of correct instances of renaming package, renaming class and interface, and renaming field and method. This implies that at renaming threshold 0.3, the renaming recall is $2077/2180$ (95.3%). What is also interesting to note, is that, even at the extremely low renaming threshold 0.01, the

UMLDiff renaming precision is 2154/2945 (73.1%) is not too low. This robustness is due to *UMLDiff*'s similarity-ranking mechanism.

Next, we set the renaming threshold at 1 and the move threshold at 0.01. Given this very strict criterion for recognizing renamings, only model elements in the same parent context that have the exact same relationships with other known-to-be-same elements are identified as renamed. All other not-yet-mapped model elements are examined against other not-yet-mapped elements in different parent context: given the very low move threshold, *UMLDiff* is eager to recognize moved elements, based on even the most tenuous similarity results. Together with manually-added missed instances of moves, we finally obtained 957 instances of moved model elements. At move threshold 0.3, *UMLDiff* reports 936 correct move instances, i.e. adding up the number of correct instances of moving class and interface, and moving field and method. Therefore the moves recall at the move threshold 0.3 is 936/957 (97.8%).

Furthermore, we also evaluated qualitatively the *UMLDiff* results against the release notes shipped with each major release of JFreeChart. Most of the changes recorded in release change logs can be recovered but with much more detail by *UMLDiff*. In this sense, the design changes reported by *UMLDiff* can be used to re-document the system evolution, which could be very useful to help the developers to capture the API and design changes.

3.4.1.1 When *UMLDiff* gets confused?

Table 3-11. *UMLDiff* results at renaming/move threshold 0.3

| Type of changes | # reported instances | # correct instances | Precision |
|-----------------------------|----------------------|---------------------|--------------|
| Renamed package | 29 | 29 | 100% |
| Renamed class and interface | 128 | 121 | 94.5% |
| Moved class and interface | 306 | 306 | 100% |
| Renamed field and method | 2024 | 1927 | 95.2% |
| Moved field and method | 721 | 630 | 87.3% |
| Data type and return type | 710 | 677 | 95.4% |
| Visibility change | 855 | 845 | 98.8% |
| No-access modifier change | 303 | 299 | 98.7% |
| Generalization change | 185 | 180 | 97.3% |
| Implementation change | 1025 | 970 | 94.6% |
| Total | 6286 | 5984 | 95.2% |

Even though the precision and recall rates are quite good, it is interesting to understand when *UMLDiff* gets confused. Let us, therefore, review the cases of erroneously reported changes. Table 3-11 shows the *UMLDiff* results with the renaming and move thresholds set to 0.3.

Renamed class and interface: 7 out of 128 class/interface renamings were incorrect. 2 involved pairs of JUnit test classes and 5 involved demo classes. All mistakenly recognized pairs of classes were very similar. The JUnit classes shared methods such as `suite()`, `testEqual()`, `testCloning()` and `testSerialization()` and the demo classes had methods like `main()`, `createChart()`, and `createDataset()`. Moreover, the efferent relations of these class pairs are also similar. For example, the `suite()` methods of the JUnit test classes created an instance of `TestSuite` with parameter `TestClass.class` and their `testSerialization()` methods use the `ByteArrayOutputStream`, `ObjectOutputStream`, `ObjectOutput` classes. Finally, none of these classes had any afferent relations: the JUnit classes are launched by the JUnit framework, and the demo classes are stand-alone java applications.

Move class and interface: All reported instances are correct, a few instances are missed though. For example, in version 0.9.5, the interface `CategoryItemRenderer` was moved from package `com.jrefinery.chart` to `com.jrefinery.chart.renderer`. At the same time, 11 new methods were added to its original 10 methods, which also changed signatures. Furthermore, out of 9 classes that use this interface in both versions, only 3 pairs of them were matched. This dramatic change make the moving of interface `CategoryItemRenderer` not recognized by *UMLDiff* at threshold 0.3.

Rename field and method/constructor: 97 out of 2024 field and method renamings were not correctly identified. Most of them involved get and set methods and the fields they access. These methods are simple and short, with few relationships to other entities and present a challenge to *UMLDiff*'s structure-similarity heuristic.

It is important to note here that among the 1927 correctly identified field and method renamings, there exist renamings that had no identifier similarity at all, and therefore would not have been intuitively recognized by a developer, such as for example, the `CategoryPlot.getDataArea()` is correctly identified as renamed to `calculateAxisSpace()` in version 0.9.10.

Move field and method: The precision of recognizing moving fields and methods is the lowest among all the different types of design changes reported by *UMLDiff*. However, its recall is very good. For example, in version 1.0.0, a superclass `AbstractPieItemLabelGenerator` was extracted from class `StandardPieItemLabelGenerator`, and 6 (all possible moves) fields and methods were reported as having moved from the subclass to the new superclass. In addition, `createItemArray()` is also identified as moving to `StandardXYSeriesLabelGenerator`, which, at closer inspection, is an incorrect one.

Other errors: The accuracy of the reported changes to data types and return types, no-access modifiers, visibility, and generalization and abstraction relations is relatively higher than that of renamings and moves. The occasional errors are due to: (a) erroneously identified renamed and/or moved model elements; (b) missed renamings and/or moves; or (c) combined moves and identifier-renamings. If two model elements are mistakenly identified as renamed or moved, their different data (return) types, modifiers, visibility, and generalization/abstraction relations will also be reported as changes. On the other hand, if a renaming and/or move is missed, the model elements referring to the renamed/moved element will mistakenly be reported as changed. For example, the interface `CategoryItemRenderer` was not identified as moving to the new package `com.jrefinery.chart.renderer` in version 0.9.5, and, consequently, the type of field `renderer` and method `getRenderer()` of class `CategoryPlot` were identified as changed. In addition, the interface implementation of 15 `renderer` classes that implemented the `CategoryItemRenderer` interface were also identified as changed. Finally, since *UMLDiff* does not attempt to identify cases of combined identifier-renamings and moves, if a class was renamed and then moved, such as for example, `CrosshairInfo` in package `com.jrefinery.chart` in version 0.9.16 and `CrosshairState` in `com.jrefinery.chart.plot` in version 0.9.17, they will be treated as removed and newly added entities which might also result in the wrong data (return) type, and inheritance and implementation changes being identified.

Summary: As discussed above, there are three typical situations in which *UMLDiff* may get confused:

- *UMLDiff* is based on lexical-similarity and structure-similarity heuristics. If two “irrelevant” model elements have very similar names and relations to other elements, they may be erroneously identified as renamings or moves.
- *UMLDiff* assumes that enough entities remain the “same” between two compared versions. If all or most of the model elements related to two renamed or moved elements were also renamed and/or moved, the structure-similarity heuristic may fail and thus *UMLDiff* may miss the renamings or moves.
- When two renamed or moved model elements have very few relations with other elements, it is difficult for *UMLDiff* to determine whether or not they represent a single conceptual element in the two compared system versions.

3.4.2 *UMLDiff* robustness

Next, let us review several factors that can impact the quality (in terms of precision and recall) of the renamings and moves reported by *UMLDiff*. To discuss the impact of each particular factor, we fix the others at the values that enable the identification of most renaming or move instances. Furthermore, we focus on the renamings and moves of classes/interfaces, attributes, and operations, since in our experience with several case studies, no subsystem/package renamings and moves were ever erroneously reported or missed by *UMLDiff*.

3.4.2.1 Renaming and move threshold

To understand how sensitive *UMLDiff* is to the choice of the “right” renaming and move threshold, we experimented with a few different thresholds. Table 3-12 presents the *UMLDiff* results at renaming and move threshold 0.35. Compared with the results at threshold 0.3, six less instances of renamed classes and interfaces are reported: four of them are actual class renamings that are not recognized at threshold 0.35, while the other two are incorrect instances reported at 0.3 but correctly ignored at threshold 0.35.

UMLDiff misses the moves of three classes at 0.35 *CategoryAxis*, *CategoryPlotConstants*, and *AbstractRenderer* into the corresponding new package in version 0.9.5. For renamed field and method, 79 less instances were reported. 29 of them are incorrect instances being filtered out at threshold 0.35, while 40 are actual renamings missed at 35%. The

other 10 instances are the results of four missed class renamings, also incorrectly ignored at this threshold.

Table 3-12. *UMLDiff* results at renaming/move threshold 0.35

| Type of changes | # reported instances | # correct instances | Precision |
|-----------------------------|-----------------------------|----------------------------|------------------|
| Renamed package | 29 | 29 | 100% |
| Renamed class and interface | 122 | 117 | 95.1% |
| Moved class and interface | 303 | 303 | 100% |
| Renamed field and method | 1945 | 1888 | 97.9% |
| Moved field and method | 686+14 | 608+14 | 88.6% |
| Data type and return type | 682 | 662 | 97.1% |
| Visibility change | 841 | 836 | 99.4% |
| No-access modifier change | 296 | 295 | 99.7% |
| Generalization change | 186 | 178 | 95.7% |
| Implementation change | 1025 | 962 | 93.7% |
| Total | 6129 | 5877 | 95.8% |

22 correct instances of moved fields and methods are missed at threshold 0.35, while 14 fields and methods of missed pairs of renamed and moved classes are identified as move. The precision of renamings and moves at threshold 0.35 are slightly better than that of threshold 0.3, but as expected, the recalls are slightly lower, 94.5% for renamings and 94.6% for moves. Most of disappeared instances for visibility and modifier changes are incorrect instances, and thus they get relative bigger increases in precision. Because there are seven actual class and interface renamings and moves are not recognized by *UMLDiff* at 0.35, model elements that refer to these classes and interfaces are considered to be changed. This directly results in the slight decrease of precision of changes to generalization and abstraction relations.

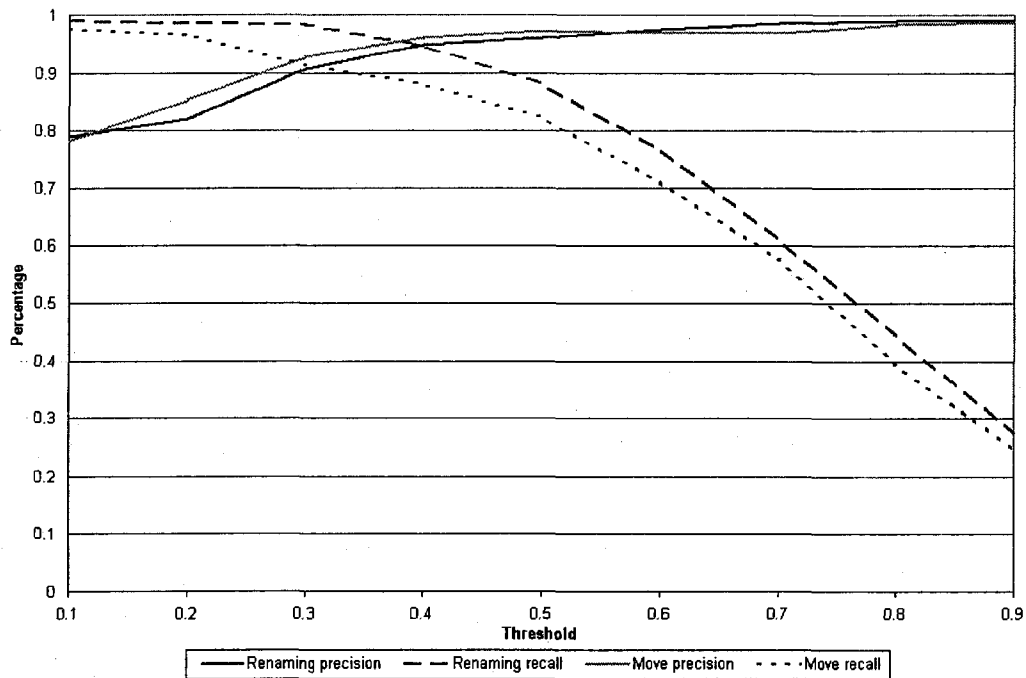


Figure 3-2. The impact of the user-specific renaming and move thresholds

Figure 3-2 summarizes the overall impact of the user-specific renaming and move threshold on the quality of *UMLDiff* results. We run *UMLDiff* on JFreeChart with the renaming and move thresholds set to 0.1 through 0.9, with 0.1 increment (using the CharLCS lexical-similarity metric, with comment-similarity, and transitive-usage-similarity) and computed the precision and recall of renamings and moves at each threshold. We found that a renaming threshold slightly higher than the move threshold, with both being within the 0.3 to 0.5 range, is an effective setting for accurately recognizing both renamings and moves. A threshold higher than 0.5 produces results with tenuous precision improvement but at a significant cost of recall, while a threshold below 0.3 produces results with slightly better recall but much worse precision.

3.4.2.2 Regularity of CVS usage

We also examined the changes that *UMLDiff* reported when comparing major releases and the changes it reported when comparing intermediate versions in order to assess the impact of not having regular and frequent versioning-system updates. To that end, we

examined the release versions documented with the major API changes, such as version 0.8.0 and 0.9.0, or those of lower precision, such as 0.9.16 and 0.9.17.

For example, the class `CombinedXYPlot` in version 0.9.0 was identified by *UMLDiff* as a renaming of class `CombinedPlot` in version 0.8.0. In fact, the `CombinedPlot` was renamed `MultiXYPlot` on April 23, 2002 and subsequently it was renamed again `CombinedXYPlot` on May 23, 2002 just before release 0.9.0. Clearly, when *UMLDiff* only compares the two major releases - 0.8.0 on March 22, 2002 and 0.9.0 on June 7, 2002 - the intermediate renaming is missed. In general, the smaller the distance between two compared versions, the higher the detail of the report is likely to be.

As another example, the precision of reported renamings between version 0.9.16 and 0.9.17 is about 90.6%, which is worse than the overall precision 95.2% at renaming threshold 0.3. 15 of 159 reported renamings are incorrect, among which 6 are related to renaming fields and methods `?ItemLabelGenerator` of class `AbstractCategoryItemRender` and interface `CategoryItemRender` to `?ToolTipGenerator`, which did not reflect what changes were really made, since by checking the source code we know that they were actually renamed to `?LabelGenerator`. This low precision guides us to further investigate the intermediate changes by taking the weekly snapshots between two major releases 0.9.16 and 0.9.17, which resulted in 12 snapshots from January 9, 2004 to March 26, 2004. *UMLDiff* was applied to these 12 weekly snapshots, which produced more accurate results. 5 of 6 wrong instances were corrected, except for `baseItemLabelGenerator` of `AbstractCategoryItemRender` being still identified as renamed to `baseToolTipGenerator`.

Clearly, the quality of *UMLDiff* results is affected by the frequency of saving changes back to versioning system and the time duration between two compared versions. In general, *UMLDiff* will produce better and more accurate results if the changes are properly saved in time and the short time period is used between two compared system versions.

3.4.2.3 *UMLDiff* parameters

We comparatively evaluated the appropriateness of different lexical-similarity metrics for assessing the name similarity of two compared model elements. Table 3-13 summarizes the number of identified renaming instances, with different name-similarity metrics and

the renaming threshold 0.3. The impact of choosing a particular name-similarity metric was most pronounced when recognizing attribute renamings (precision ranging from 87.5% to 90.1% and recall ranging from 94.6% to 99.7%), in contrast to recognizing class/interface renamings and operation renamings with identifier changes, where the choices of different name-similarity metrics were almost indistinguishable. Overall, none of the three used metrics, Char-LCS, Word-LCS and Char-Pair, is significantly better, although the Char-Pair metric seems to produce results with a better balance of precision and recall.

Table 3-13. Recognizing renamings with different name-similarity metrics

| | | Correct | Wrong | Precision | Recall |
|---------------------|-----------|---------|-------|--------------------|--------|
| Class and Interface | Char-LCS | 123 | 48 | 71.9% ⁸ | 98.4% |
| | Char-Pair | 123 | 47 | 72.4% | 98.4% |
| | Word-LCS | 123 | 47 | 72.4% | 98.4% |
| Attribute | Char-LCS | 295 | 42 | 87.5% | 99.7% |
| | Char-Pair | 290 | 33 | 89.7% | 97.9% |
| | Word-LCS | 280 | 31 | 90.1% | 94.6% |
| Operation | Char-LCS | 794 | 126 | 86.3% | 97.2% |
| | Char-Pair | 792 | 116 | 87.2% | 97.0% |
| | Word-LCS | 793 | 118 | 87.0% | 97.1% |
| Overall | Char-LCS | 1212 | 216 | 84.9% | 97.9% |
| | Char-Pair | 1205 | 196 | 86.0% | 97.3% |
| | Word-LCS | 1196 | 196 | 85.9% | 96.6% |

We examined the effectiveness of the two techniques for propagating the knowledge about the identified renamings and moves through usage and inheritance relations. They are both useful in increasing the recall of renamings and moves; the corresponding slight decrease in precision should not be a major concern, since the users should be able to easily recognize and filter out the false positive instances reported. For example, with the renaming and move thresholds set to 0.3, about 1.2% of all the operation renamings and moves were recovered through second and third rounds of renaming and move

⁸ The low precision of class renaming is due to the large amount (31 out of 48) of demo and junit test classes being identified as renamed. Most of them can be prevented with higher renaming threshold.

recognition; about 7.6% of all the operation renamings were recovered through propagating the operation renamings depending on the generalization/abstraction relations. Less than 10 instances were erroneously reported due to the application of these two techniques. Thus, we believe these two techniques are very effective at recovering renamings and moves that would otherwise be missed.

Table 3-14. Recognizing renamings and moves with and without comment-similarity

| | | | Correct | Wrong | Precision | Recall |
|-----------|---------------------|---------|---------|-------|-----------|--------|
| Renamings | Class and Interface | Without | 121 | 36 | 77.0% | 96.8% |
| | | With | 123 | 48 | 71.9% | 98.4% |
| | Attribute | Without | 278 | 26 | 91.5% | 94.0% |
| | | With | 295 | 42 | 87.5% | 99.7% |
| | Operation | Without | 1590 | 62 | 96.3% | 91.9% |
| | | With | 1696 | 129 | 93.0% | 98.3% |
| Moves | Class and Interface | Without | 296 | 0 | 100% | 95.8% |
| | | With | 299 | 0 | 100% | 96.8% |
| | Attribute | Without | 186 | 5 | 97.4% | 83.4% |
| | | With | 200 | 9 | 95.7% | 89.7% |
| | Operation | Without | 343 | 25 | 93.2% | 80.7% |
| | | With | 375 | 58 | 86.7% | 88.3% |

Table 3-15. Recognizing operation renamings and moves with/without transitive usage similarity

| | | Correct | Wrong | Precision | recall |
|---------------------|---------|---------|-------|-----------|--------|
| Operation renamings | Without | 1672 | 74 | 95.8% | 96.6% |
| | with | 1696 | 129 | 93.0% | 98.3% |
| Operation moves | Without | 375 | 55 | 87.2% | 88.3% |
| | With | 375 | 58 | 86.7% | 88.3% |

We evaluated the impact of additional sources of information, i.e. comment and transitive usage dependency, on *UMLDiff*'s accuracy. Table 3-14 and Table 3-15 summarize the impact of comment similarity and transitive usage similarity on the precision and recall of identified renamings and moves of classes/interfaces, attributes, and operations. Overall, the comments of model elements and their transitive usage dependencies can effectively inform the process to further increase its recall, albeit at a small precision cost. Based on the estimated number of changes, the time lapse between two compared versions and the need for the more coverage of changes or the more

precise results with shorter comparison time, the users may turn on or off these additional sources of information when comparing renaming and move candidates.

3.4.3 *UMLDiff* run-time performance

Finally, let us examine the run-time performance of *UMLDiff* algorithm. The run-time complexity of *UMLDiff* is determined by the renaming and move recognition process, which require the pair-wise comparison of the not-yet-mapped model elements in two compared versions of the system model. Through the use of appropriate in-memory data structures and efficient database indexing, the run-time complexity of the renaming and move recognition process is $O(\alpha \cdot N \cdot M)$, where N and M are the number of not-yet-mapped model elements of the same type in two compared versions respectively. *UMLDiff* only attempts to identify the pairs of renamed model elements within the context of pairs of the mapped parent elements, and it only identifies the pairs of moved elements with same names (same identifiers for operations). In the worst-case scenario, when all the not-yet-mapped elements are contained in one pair of mapped parent elements or all the not-yet-mapped elements have the same names, the α is equal to 1; in the best-case scenario, when all the not-yet-mapped elements are contained in different parent elements or all the not-yet-mapped elements have different names, the α is equal to 0. In our experience with several case studies, the α is usually very small.

Table 3-16 summarizes the run-time complexity of *UMLDiff* when comparing the subsequent releases of the JFreeChart system, with *RenameThreshold*=0.3 and *MoveThreshold*=0.3. The column “Versions” indicates that the information summarized in a particular row is collected when the system evolved from the version of one row above to this version. The columns “N” and “M” list the number of not-yet-mapped model elements in the two compared versions. The column “#Comp” lists the number of comparisons that *UMLDiff* performed for identifying the attribute/operation renamings and moves. The α is consequently computed as $\#Comp/(N \cdot M)$. As shown in the Table 3-16, the α is very small. Table 3-16 also presents the recalls of attribute/operation renamings and moves, which indicates that the *UMLDiff* is quite effective at recovering the renamed and moved model elements by comparing only a very small subset of not-yet-mapped candidates.

Table 3-16. The run-time complexity of *UMLDiff* in JFreeChart case study

| Versions | Attribute/Operation Renamings | | | | | Attribute/Operation moves | | | | |
|----------|-------------------------------|-----|-------|----------|--------|---------------------------|------|-------|----------|------------------|
| | N | M | #Comp | α | Recall | N | M | #Comp | α | Recall |
| 0.6.0 | 154 | 357 | 1119 | 0.05 | 0.98 | 203 | 378 | 64 | 0.002 | 0.95 |
| 0.7.0 | 6 | 33 | 36 | 0.36 | 1.0 | 4 | 188 | 2 | 0.004 | 1.0 |
| 0.7.1 | 72 | 169 | 298 | 0.06 | 1.0 | 48 | 190 | 16 | 0.003 | 1.0 |
| 0.7.2 | 56 | 101 | 130 | 0.06 | 1.0 | 51 | 104 | 7 | 0.003 | n/a ⁹ |
| 0.7.3 | 5 | 10 | 6 | 0.12 | 1.0 | 1 | 18 | 2 | 0.13 | 1.0 |
| 0.7.4 | 32 | 34 | 14 | 0.04 | 1.0 | 22 | 57 | 18 | 0.03 | 1.0 |
| 0.8.0 | 42 | 69 | 37 | 0.04 | 1.0 | 83 | 233 | 37 | 0.004 | 1.0 |
| 0.9.0 | 170 | 313 | 1519 | 0.06 | 0.94 | 216 | 842 | 244 | 0.003 | 0.85 |
| 0.9.1 | 2 | 10 | 2 | 0.11 | 1.0 | 0 | 89 | 0 | 0 | n/a |
| 0.9.2 | 99 | 99 | 1168 | 0.29 | 0.97 | 60 | 105 | 9 | 0.004 | 1.0 |
| 0.9.3 | 109 | 207 | 618 | 0.06 | 0.95 | 144 | 1003 | 111 | 0.001 | 1.0 |
| 0.9.4 | 212 | 297 | 1737 | 0.07 | 1.0 | 121 | 406 | 74 | 0.004 | 0.70 |
| 0.9.5 | 564 | 721 | 4842 | 0.02 | 0.97 | 422 | 1303 | 267 | 0.001 | 1.0 |
| 0.9.6 | 10 | 43 | 35 | 0.34 | 1.0 | 1 | 42 | 0 | 0 | n/a |
| 0.9.7 | 138 | 242 | 213 | 0.01 | 1.0 | 190 | 583 | 227 | 0.004 | 0.92 |
| 0.9.8 | 52 | 54 | 24 | 0.01 | 1.0 | 31 | 96 | 85 | 0.04 | 1.0 |
| 0.9.9 | 309 | 625 | 3475 | 0.04 | 0.99 | 406 | 876 | 470 | 0.002 | 0.92 |
| 0.9.10 | 233 | 361 | 3044 | 0.07 | 1.0 | 137 | 213 | 23 | 0.001 | 0.30 |
| 0.9.11 | 16 | 73 | 10 | 0.02 | 1.0 | 8 | 237 | 11 | 0.01 | 1.0 |
| 0.9.12 | 106 | 383 | 690 | 0.03 | 1.0 | 124 | 550 | 345 | 0.008 | 0.90 |
| 0.9.13 | 83 | 174 | 410 | 0.04 | 1.0 | 6 | 161 | 3 | 0.005 | 1.0 |
| 0.9.14 | 168 | 333 | 821 | 0.02 | 0.96 | 89 | 427 | 27 | 0.001 | 0.80 |
| 0.9.15 | 21 | 43 | 23 | 0.04 | 1.0 | 13 | 180 | 2 | 0.001 | 1.0 |
| 0.9.16 | 70 | 76 | 95 | 0.04 | 1.0 | 85 | 202 | 67 | 0.008 | 0.98 |
| 0.9.17 | 272 | 495 | 3389 | 0.06 | 0.94 | 168 | 974 | 282 | 0.003 | 0.92 |
| 0.9.18 | 59 | 119 | 230 | 0.10 | 1.0 | 35 | 191 | 4 | 0.001 | 1.0 |
| 0.9.19 | 219 | 385 | 3444 | 0.08 | 0.98 | 242 | 418 | 108 | 0.002 | 0.92 |
| 0.9.20 | 9 | 21 | 5 | 0.08 | 1.0 | 4 | 64 | 1 | 0.008 | n/a |
| 0.9.21 | 118 | 272 | 574 | 0.02 | 1.0 | 1043 | 349 | 75 | 0.001 | 0.35 |
| 1.0.0 | 129 | 354 | 641 | 0.03 | 0.97 | 53 | 531 | 29 | 0.002 | 1.0 |

Furthermore, the actual time cost of *UMLDiff* is affected by the size of the system and the number of its versions, i.e., the size of JDevAn database. Table 3-17 summarizes the actual time cost of applying *UMLDiff* to pair-wisely compare subsequent system versions of JFreeChart on an Intel Centrino 1.6GHz machine with 768M physical memory. The average time required for *UMLDiffing* two subsequent versions of JFreeChart system is about 10-12 minutes. For those releases that have major changes, such as version 0.9.5,

⁹ "n/a" indicates that there are no moves of model elements in the compared models.

0.9.9, 0.9.19, *UMLDiff* requires about 30-50 minutes; most of this time is used to detect moves and renamings. As shown in Appendix C, *UMLDiff* deals with a very large information database. Therefore, if major changes were made between two compared versions, to determine potential moves and renamings, *UMLDiff* has to query the database for retrieving their corresponding relationships and previously established matched pairs of entities, which is a time-consuming process.

Table 3-17. The actual time cost of *UMLDiff* in JFreeChart case study

| Compared subsequent versions | Time (mins) |
|---|-------------|
| 0.5.6 – , 0.7.0 – 0.6.0, 0.7.2 – 0.7.1, 0.7.3 – 0.7.2, 0.7.4 – 0.7.3, 0.8.0 – 0.7.4, 0.9.1 – 0.9.0, 0.9.6 – 0.9.5 | <1 |
| 0.6.0 – 0.5.6, 0.7.1 – 0.7.0 | 2 |
| 0.9.0 – 0.8.0, 0.9.2 – 0.9.1, 0.9.3 – 0.9.2, 0.9.8 – 0.9.7, 0.9.11 – 0.9.10 | 4 ~ 6 |
| 0.9.7 – 0.9.6, 0.9.15 – 0.9.14, 0.9.16 – 0.9.15 | 8 ~ 10 |
| 0.9.4 – 0.9.3, 0.9.12 – 0.9.11, 0.9.13 – 0.9.12, 0.9.18 – 0.9.17, 0.9.20 – 0.9.19 | 11 ~ 14 |
| 0.9.14 – 0.9.13, 0.9.21 – 0.9.20, 1.0.0 – 0.9.21 | 16 ~ 10 |
| 0.9.5 – 0.9.4, 0.9.10 – 0.9.9 | 23 ~ 25 |
| 0.9.9 – 0.9.8 | 37 |
| 0.9.17 – 0.9.16 | 52 |
| 0.9.19 – 0.9.18 | 58 |
| Total | ~370 |

3.5 Summary

In this chapter, we described the *UMLDiff* algorithm for differencing object-oriented logical-design models. This algorithm is aware of the UML semantics and compares software versions at the design level, so that its results are more directly relevant to the evolutionary-development process than either lexical or code-metrics differencing. Furthermore, we believe – while recognizing that empirical evaluation is required to demonstrate our belief – that its results are more intuitive to developers, compared to other structure-differencing algorithms that rely on low-level program representations such as ASTs, program-dependency graphs or XML.

Our experimentation with the algorithm demonstrated that the algorithm is quite accurate, when the project uses a consistent versioning scheme; it identifies about 96% of the renamed and moved model elements at about 94% of precision in JFreeChart case

study. *UMLDiff* is also robust to the user's choice of parameters, which configure the differencing process; it produces results with both good precision and recall at a wide range of user-specific renaming and move thresholds; its differencing process can be configured according to the estimated amount of design changes, the time lapse between two compared versions and the need for the more coverage or precise results. Finally, the JDEvAn implementation of the algorithm is practically efficient.

Chapter 4: Query-based Change Pattern Detection

An important kind of change pattern to object-oriented software is refactoring [32,51,72]. The goal of refactoring is to improve the design and quality of the software system, without affecting its overall functionality and behavior. Recognizing its beneficial impact to software design, several modern IDEs [79,116,124] support refactoring, albeit usually for simple refactorings.

In addition to supporting refactoring in forward engineering activities, it is also interesting to recognize refactorings, and more generally, structural change patterns, in the history of a software project. Recognizing refactorings is important for two main reasons. First, since different types of design changes aim to improve different aspects and qualities of the system design – e.g., refactorings that collapse class hierarchies reduce the system layers and are likely to increase the system performance, while class extraction has the opposite effect [49] – the history of the actual changes that a software system has suffered is evidence of the qualities relevant to the project that new team members should be aware of. Second, the recognition of refactorings to the published API of reusable component frameworks is an essential prerequisite for applications that need to migrate from earlier versions of the framework to the most recent one.

Recent research on inferring refactorings in the software evolution history has been based on examining change documentation [22], or comparatively analyzing source-code metrics [20], or clone detection [23,36,83], or visualization [37]. However, all these approaches suffer from some non-trivial disadvantages. More frequently than not, there is no consistently maintained change documentation. Aggregate code metrics do not provide sufficient information to precisely pinpoint the elements involved in the refactoring. Clone detection is not very effective in recognizing “non-local” refactorings that involve several entities. Finally, visualization techniques do not scale well as the size of the system increases.

In this chapter, we present our approach to recognizing refactorings. Based on the elementary design changes reported by *UMLDiff*, we have defined queries to automatically detect complex design-change patterns, as compositions of elementary changes, such as the refactorings [32] listed in Fowler’s refactoring catalog. We

evaluated our refactoring-detection queries with a detailed case study on the design evolution of Eclipse [116]. The objective of this case study has been to (a) examine the actual refactoring practice in the context of a realistic framework with substantial evolution history and many client applications and (b) to come up with some requirements and design suggestions for tools purported to support the practice.

4.1 Detecting refactorings with change-pattern queries

The refactorings in Fowler’s catalog [32], henceforth also referred to as “standard” refactorings, are well known and understood. They are also frequently used in practice. Refactorings can be local or global and, sometimes, they may result in many scattered low-level changes to the logical model of the system. Although one may still understand how the software system has been evolved by examining a set of small, elementary changes, such as those reported by *UMLDiff*, we believe that, by combining the relevant elementary changes into change patterns such as refactorings, it becomes easier to understand the specific intent of the design evolution and support the subsequent development. The instances of these refactorings are recognized from their effects on the logical model, in terms of queries of *UMLDiff* design-change facts. They are reported in terms of their types and participants.

We organize these refactorings in four general categories according to their intention and scope, as shown in Table 4-1: those dealing with containment hierarchy, those dealing with inheritance hierarchy, those moving features between objects, and those refactoring class internals. These refactorings are further characterized as *simple* or *composite*, depending on the amount of the elementary change facts they involve. Appendix F reviews the queries for detecting these refactorings. Appendix G summarizes the instance of refactorings these queries reported in the evolution of HTMLUnit [123], JFreeChart [126] and Eclipse [116] respectively.

More generally, our approach is not limited to only the Fowler-catalog refactorings. Software developers can define queries to query the design changes, reported by *UMLDiff* and subsequently produced through the discovery of instances of simple or composite refactorings, for design-change patterns of their interests.

For example, consider the Java deprecation mechanism that is used to evolve APIs by supporting backward compatibility and allowing the system to evolve into a better design. One may be interested in identifying the model elements that have just stopped being deprecated in a latter version, after being tagged as deprecated, since this phenomenon may indicate that the developers have given up on moving to a clearer API. Information about newly or no longer deprecated element is reported directly by *UMLDiff*. One may also be interested in identifying the classes that have been deprecated and their corresponding replacements. A query searching for the class being newly tagged as deprecated and being the source of the refactoring *extract class* may return the candidates for further inspection.

Table 4-1. Fowler's "standard" refactorings

| Category | Simple refactorings | Composite refactorings |
|---------------------------------|--|--|
| Dealing with Containment | Convert inner type to top-level ^{*10} Convert top-level type to inner* Move subsystem/package/class* | Extract subsystem/package Inline subsystem/package |
| Dealing with Generalization | Pull-up method/field* Push-down method/field* Pull-up behavior Push-down behavior Pull-up constructor body | Extract interface* Extract superclass* Extract subclass* Inline superclass Inline subclass Form template method Replace inheritance with delegation* Replace delegation with inheritance* |
| Moving features between objects | Move method/field* Move behavior | Extract class Inline class Die-hard and legacy classes |
| Refactoring class Internals | Rename subsystem/package* Rename class/method/field* Add parameter* Remove parameter* Information hiding Generalize type* Downcast type Extract method* Inline method* | Introduce factory method* Introduce parameter object Encapsulate field* Preserve whole object |

¹⁰ The "*" indicates that the state-of-the-art IDEs, such as Eclipse [116], IntelliJ IDEA [124], support the refactoring.

4.2 Refactoring practice: How it is and how it should be supported

Eclipse is a large-scale industrial framework that has been under development for about six years. In the process, it has acquired a large user base and a multitude of applications have been built on it. Eclipse is built as a plugin-based framework. Its users can simply use it as an IDE, but they can also extend or build their own plugins from the existing ones. Since version 3.0, Eclipse introduced a concept of a rich client platform, which allows its users to build stand-alone applications from a subset of plugins. Therefore, studying the design evolution of Eclipse can help us understand the design requirements for refactoring-based development environment from the perspectives of both the component developers and component users.

Eclipse consists of three subprojects and in this case study, we have focused on the JDT subproject, which defines about half of the classes and interfaces of the whole Eclipse platform. Clearly, the substantial numbers of program entities (407720), relations (2220707) and changes (58973) (see Table E-1, Table E-2 and Table E-3) preclude the existing refactoring-detection approaches from effectively discovering what refactorings have been made in the evolution of Eclipse. In this section, we describe a detailed case study on the design evolution of Eclipse with our automatic refactoring-detection queries, which help us gain insight into the following research questions:

- What proportion of the design changes in the evolution of a system are the results of refactoring?
- What are the typical refactorings applied in practice?
- Which of these types are “safe” to client applications that reuse the refactored system?
- What type of support should modern IDEs provide and how might this support be implemented?

4.2.1 The empirical assessment of the design evolution of Eclipse

First, we describe and assess the empirical data we collect in our study on the design evolution of Eclipse. We will summarize our findings in next subsection.

4.2.1.1 Elementary design changes

First, let us examine seven types of elementary design changes reported by *UMLDiff* algorithm. We start our discussion with renamings and moves, which we expect to be the more benign changes, i.e., changes that are likely to be behavior preserving and therefore relatively easy to propagate their implications to the client applications of the earlier version. We then proceed to examine increasingly “suspect” modifications, such as modifier and visibility changes, data-type changes, inheritance-hierarchy changes, and entity additions and removals.

4.2.1.1.1 Program-entity renamings

There are 4891 renamings of various types of program entities, including packages, classes and interfaces, methods¹¹ and fields. Note that 2 moved classes, 1 moved interface, 42 (5+8+29) moved fields, and 264 (19+162+83) moved methods were renamed (marked with “#”) as well as moved. Renamings of packages, classes and interfaces, and fields involve changes to their identifier. Method renamings may involve changes to the whole method signature (including identifier and/or parameter list). About 24% (984/4170) renamed methods had only their identifiers changed.

Table 4-2. Rename program entities

| | 2.1 – 2.0 | 3.0 – 2.1.3 | 3.1 – 3.0.2 | Total |
|-------------------------------|-----------|-------------|-------------|-------------|
| Rename package | 1 | 1 | 0 | 2 |
| Rename class [#] | 20 | 47 + 2 | 24 | 91 + 2 |
| Rename interface [#] | 2 | 1 + 1 | 0 | 3 + 1 |
| Rename field [#] | 107 + 5 | 274 + 8 | 199 + 29 | 580 + 42 |
| Rename method [#] | 559 + 19 | 1647 + 162 | 1042 + 83 | 3248 + 264 |
| Rename constructor | 120 | 315 | 223 | 658 |
| Total | 833 | 2455 | 1600 | 4891 |

Through inspection, we identified several plausible motivations behind renamings:

1. *Conformance to a consistent naming scheme;*
2. *Reflecting the semantics of an internal implementation change to the entity;*

¹¹ In our discussion the method changes also include similar changes to constructors, except for move. *UMLDiff* does not consider the move of constructor.

3. *Concept merging or splitting; and*
4. *Maintaining backward compatibility with earlier versions.*

A consistent naming scheme improves code readability and understandability, especially when the identifiers allude to the functions of the program entities. 29 renamings were simply to correct spelling or wrong names resulting from code cutting and pasting. Some of the renamings were motivated by the adoption of a more meaningful name for the entity: clearly, `fSelectedCU`, `isOnBuildPath()` and `AccessorClassCreator` reveal the purposes of the program entities much more clearly than their precursors `fCU`, `checkJavaElement()` and `AccessorClass`. In other cases, the renamings were more “syntactic” aiming to simply conform to the adopted naming convention. For example, 79 fields were renamed to remove the prefix “f”; at the same time, it is interesting to note that 11 other fields were renamed by adding the same prefix “f”. This phenomenon may be because different Eclipse plugins adopt different and occasionally contradictory naming conventions. In another case, 12 fields were capitalized because they were declared “static” and/or “final”, while 6 fields were converted to lowercase when they stopped being static final constants.

Renamings also reflect implementation changes. For example, the data (return) type of 723 renamed fields (methods) was also changed. In version 3.0, 7 classes `RenameXXXRefactoring` were renamed to `RenameXXXProcessor`, which corresponds to the introduction of the new concept of processor-based refactoring. In version 3.0, package `org.eclipse.jdt.internal.ui.text.template` was renamed to `org.eclipse.jdt.internal.ui.text.template.preferences` since its two classes related to content-assist features were extracted to a newly created package, named `org.eclipse.jdt.internal.ui.text.template.contentassist`, and it now contained only preferences-related classes.

The member class `ProjectCache` was created to encapsulate two fields `allPkgFragmentRootsCache` and `allPkgFragmentsCache`, whose role was replaced by a field of type `ProjectCache`. Finally, as an example of backward-compatibility renaming, consider class `ASTRewrite`, which was renamed to `OldASTRewrite` that delegates to the new (with totally different implementation) `ASTRewrite` in a new package.

The question then becomes: “how easy is to modify the clients of the renamed entities if they are carried over to the new Eclipse version?”

References to entities with only modified identifiers can be automatically updated with little cost by parsing the source code and scanning the abstract syntax tree (AST). The case of method signature changes (including parameter-list changes) is a bit more intricate. The parameter lists of about 16% (654/4170) renamed methods were changed in some combination of the following three types: (a) a parameter type was renamed or moved; (b) the parameter order was changed; (c) a parameter was removed. The combination of these three types of parameter list changes cannot be handled as easily as simple identifier changes and would require special support by the refactoring IDE and corresponding refactoring-migration tool.

A small fraction (less than 2% of 4170) of renamed methods changed the parameter type to its supertype to make the method more general. Such changes would be transparent to client code at compile time.

In about 58% of method renamings, the parameter list was extended with at least one additional parameter. In 33% (1352/4170) of the cases, there was only newly added parameter(s) without removed parameter(s) (but may have other types of parameter list changes listed above). Such types of changes often indicate that the method delivers some additional functionality by making use of the additional parameter(s). In the case of constructors, additional parameter(s) are frequently used to initialize corresponding newly added field(s). Since additional parameter(s) most commonly indicate new behavior, these renamings are in effect non-behavior-preserving and should not be considered as refactorings.

In 25% (1110/4170) of method-renaming cases, the new parameter lists included newly added parameters as well as removed ones. In some cases, a parameter is replaced by several others. For example, the method `javadocDuplicatedParamTag(JavadocSingleNameReference)` used only a few pieces of information from its parameter object. It was subsequently renamed to `javadocDuplicatedParamTag(char[],int,int)` in version 3.1 to take in just-enough information as parameters, since it was not concerned with the whole `JavadocSingleNameReference` object. There are also cases, such as 6 methods defined in interface `ISourceElementRequestor`, where several parameters were replaced by a single parameter, which may be the result of the *introduce parameter object* refactoring [32].

Finally, there were several types of parameter-type changes, such as replacing a boolean type with an int or long flags, replacing a primitive type with an object type (e.g. int with Integer), replacing a type with a collection of that type. In order to regard these changes as automated refactorings, one would need to invoke a proper wrapper, such as [4], for the relevant parameters and would also need to know how to access the member from the wrapper. However, the relevant methods and classes most likely exhibit other substantial changes, which cannot be expressed in terms of refactorings and would require that the developers of client applications manually modify their software.

4.2.1.1.2 Program-entity moves

There are 2315 move instances of various types of program entities. We identified several kinds of moves with different underlying motivations:

1. *Reorganizing or redistributing the information among different parts of a software system;*
2. *Moving responsibilities to eliminate Law-of-Demeter violations;*
3. *Maintaining backward compatibility with earlier versions;*
4. *“Implicit” moves*

Table 4-3. Move program entities

| | 2.1 – 2.0 | 3.0 – 2.1.3 | 3.1 – 3.0.2 | Total |
|----------------|-----------|-------------|-------------|-------------|
| Move package | 0 | 4 | 0 | 4 |
| Move class | 18 | 62 | 31 | 111 |
| Move interface | 1 | 6 | 3 | 10 |
| Move field | 172 | 318 | 331 | 808 |
| Move method | 196 | 854 | 319 | 1369 |
| Total | 387 | 1244 | 684 | 2315 |

For example, in version 3.0, three packages were moved to the new source folder of jdt.launching plugin; one package org.eclipse.jdt.internal.junit.runner was moved from the jdt.junit plugin to the newly added plugin jdt.junit.runtime. In version 3.0, the abstract class SearchPattern was moved from org.eclipse.jdt.intenal.core.search to org.eclipse.jdt.core.search to replace the role of the deprecated interface org.eclipse.jdt.core.search.ISearchPattern. In version 2.1.3, there were three Util classes scattered in three different packages of the jdt.core plugin; some of the features they provide were duplicate; in 3.0, their features

were moved (merged) into a single Util class. The overall intention of these moves is to reshape the software system so that it is easier to understand and maintain.

The “Law of Demeter” [60] – “only talk to your friends” – is essentially an object-oriented formulation of the general “low coupling” software-engineering principle. Often, moves aim at refactoring entity responsibilities so that this law is not violated. For example, in version 2.0, `JavaBasePreferencePage` used to declare a public static method `doubleClickGoesInto()`, which was only called by `PackageExplorerActionGroup.handleDoubleClick()`; in 3.0, this method was moved to `PackageExplorerActionGroup` and it was made private and no longer declared static. Such moves often involve the fields and methods defined in one class but are mostly used in other classes, which is the exact intention of *move field/method* [32] as described in Fowler’s refactoring catalog. They enhance encapsulation and reduce coupling.

Similarly to renamings, some moves aim at maintaining backward compatibility. For example, when evolving to version 3.0, the class `TextChange` was redeveloped. In order to maintain the backward compatibility, two of its public methods were moved to a new class `TextChangeCompatibility` and were declared as static; they were also given one more parameter of the type `TextChange` to which they delegate their implementation.

In some cases, moves are “implicit” (e.g., deprecation+delegation) when the “old” entity is replaced by the “new” entity but the “old” is not removed, instead it simply delegates to the “new” entity that now implements its logic. For example, in version 3.1, a new class `BasicSearchEngine` was extracted from `SearchEngine`, which has been tagged as deprecated: 13 fields and methods were moved to `BasicSearchEngine`; for the remaining 12 public methods, 12 corresponding same-signature methods were declared in `BasicSearchEngine` that implement the same logic as their counterparts in `SearchEngine`, and `SearchEngine` simply delegates to `BasicSearchEngine` for its functionalities.

Finally, in some cases, moves are the integral part of “bigger” refactorings. For example, 60 (about 50% of 121) class and interface moves are part of 16 “Extract Package” refactorings (see Section 4.2.1.2.1 for detailed discussion on this subject).

Let us now consider again the issue of the support required to carry the clients of the moved entities over to the new version.

In principle, all types of program entities can be moved. About 76% (1759/2315) of entities were moved with no other changes made to them. This is not surprising since the general intention of moving program entities is just to redistribute features in order to enhance encapsulation, understandability and maintainability, instead of modifying entities for other purposes. These entity moves represent true behavior-preserving refactorings and the references to them can be automatically updated (the information about the context of moved entities may be needed).

However, sometimes, moved methods also experienced changes to their parameter lists. They may take the “old” home class as an additional parameter, such as `TextChangeCompatibility` described above. More frequently, moved entities also undergo modifier and visibility changes. About 18% (416/2315) of moved entities had their declared modifiers and/or visibility levels changed (such as `PackageExplorerActionGroup` and `TextChangeCompatibility` discussed above). In 107 cases, their static and/or final status was toggled. In 331 cases, the visibility was modified. Many of the modifier and visibility changes can be easily wrapped. Finally, less than 7% of total 2315 moved entities came with other changes, such as data (return) type change and/or inheritance-hierarchy change. By closer inspection, they are most likely a sequence of separate (not inherently related) changes applied to the same program entity, which often require that the developers of client applications manually update their software.

4.2.1.1.3 Modifier changes

There were 1076 modifier changes (including newly added and removed modifiers) made to 1064 program entities. About 50% of these changes should not cause compilation problem or could be easily wrapped.

Java synchronization operations may incur significant performance overhead, which might affect the applications’ performance and behavior. However, an entity newly declared as synchronized will not cause a compilation failure in its client application. For entities that changed from being synchronized to not being synchronized, an escape analysis [14] can be applied to determine where it is safe to replace a synchronized object with an unsynchronized one. In cases where synchronization cannot be safely removed, a synchronization wrapper (similar to the Java standard library class `java.util.Collections`,

which is an instance of the Decorator pattern [34]) can be inserted around the object, which delegates to the given object, but makes the forwarding method synchronized.

An entity newly declared as final may or may not break client applications, depending on whether the application assigns, overrides, or extends the changed program entity. Entities that used to be, but are no longer, declared as final should not cause compilation failures to the client application.

Fields (methods) newly declared as static may cause compiler warnings such as “The static entity should be accessed in a static way”, but should not cause any compilation problems on client code. For those fields (methods) that are no longer declared as static, a factory method that returns an instance of the declaring class may be inserted; the returned instance can then be used to refer to the corresponding instance entities.

It is interesting to note that about 34% modifier changes were made to the program entities contained in a very small set of entities (about 40). For example, in version 2.1, 26 public fields of `JavadocOptionsManager` were no longer declared with static; in version 3.0, 25 methods of the class `DefaultBindingResolver` were newly declared with synchronized.

4.2.1.1.4 Visibility changes

In our analysis, we found 1842 program entities that changed their visibility: of them, 1091 changed to a less restrictive visibility level and 751 changed to a more restrictive one.

Object-oriented languages provide explicit support for defining the scope of the various design elements of a system. Frequently, developers make elements “too accessible” in the beginning. As the picture of the scope of the valid clients of each element becomes clearer, the element visibility may be restricted. For example, in about 24% (441/1842) of the visibility-change cases, an entity was made private: in about 70% of these cases (299/441) there was no incoming usage from outside their corresponding declaring classes. Most of the others gradually became used only inside their declaring classes and were finally made private in a subsequent release.

For those entities whose visibility is decreased, the changes may be safe within the component (such as Eclipse) itself. However, the client applications that depend on those

entities may break as a result. In that case, a wrapper (similar to the effect of *encapsulate field* refactoring [32]) may be used to provide the access to more restrictive entities.

About 15% (331/2315) of moved entities also changed their visibility. When entities are pulled up to a superclass or moved to helper or delegate classes, their visibility often changes to a less restrictive level in order to allow the subclasses or the original class to access them. When entities are pushed down to a subclass or moved closer to where they actually get used, their visibility often decreases since they can be accessed within the declaring scope of the current class. Sometimes, when converting nested types to top-level, their visibility may increase; on the other hand, when top-level types are converted to nested types, their visibility often decreases.

Similarly to modifier changes, about 30% of visibility changes to a more restrictive one were made to the program entities contained in a very small set of entities (about 20).

4.2.1.1.5 Data-type changes

We found 1524 data-type changes (including field data-type and method return-type). 7% of them (107/1524) were generalizations to a supertype and 6% (85/1524) were specializations to a subtype.

The clients that are now forced to use a supertype may fail to compile successfully, depending on whether they access the members that are not visible through the supertype's interface. In such cases, an explicit downcast that wraps the changed field (method) may be necessary.

Specializations to subtypes may be the result of the *encapsulate downcast* refactoring [32]: for example, when evolving to version 3.1 the return type of `CompilationUnitRewrite.createChange()` was changed from `TextChange` to `CompilationUnitChange` (but `createChange()` returns an instance of `CompilationUnitChange` in both versions 3.0.2 and 3.1). Although using a subtype in terms of its supertype will not cause a compilation failure to the client code, it may behave differently. For example, the data type of `ExceptionBreakpointFilterEditor.fFilterViewer` was changed from `TableView` to `CheckBoxTableView`. But since it is initialized with `TableView` and `CheckBoxTableView` in 2.0 and 2.1 respectively, the client gets a table viewer with check boxes instead of a plain table viewer in 2.1.

Table 4-4. Data-type changes that might be wrapped

| Type of change | #instances |
|--|------------|
| String => StringBuffer, StringBuffer => String, char.1 => String | 138 |
| int => long | 96 |
| Type <=> Collection or array of type | 24 |
| Vector => List, Hashtable => Map or HashMap, Enumeration => Vector, List => Vector, HashMap => Hashtable | 23 |
| boolean => int or long flags | 21 |
| Total | 302 |

The above table lists the data-type changes that might be wrapped; there were about 20% (302/1524) changes of these types. A refactoring tool can swap the corresponding types if there is a specification, such as [4], that can be used for guiding the migration. Furthermore, about 9% (129/1524) of method changes involved the change of their return type from void to some type. The clients of these methods can simply ignore the returned object. Finally, the remaining 60% (901/1524) of data-type changes were too radical to be considered as refactorings. For example, the return type of method `getChangedClassFiles()` was changed from List to `ChangedClassFilesVisitor`. The field `binaryPath` of type String was renamed to `binaryFolder` of type `IContainer`.

4.2.1.1.6 Inheritance-hierarchy changes

The inheritance hierarchy of Eclipse is relatively stable. There were 304 instances of class-inheritance changes in total. 72 `XXXMessages` classes started extending `org.eclipse.osgi.NLS` in version 3.1, and 34 dialog classes changed their superclass to `org.eclipse.jface.dialogs.StatusDialog` since the duplicate `StatusDialog` scattered in several plugins were finally removed in version 3.1. Among the remaining 198 changes, 90 classes changed to extend a subclass of their previous superclass (76 such subclasses are newly introduced classes); 32 classes changed to extend the superclass (16 are because the classes previously extended were removed or inlined in the new version).

There are 466 instances of classes newly implementing an interface. In 186 of these cases, the interfaces in question were newly introduced. There were 389 cases where a

type was changed not to implement an interface any longer (in 203 among them, the interface in question was also removed).

Although there are some inheritance hierarchy changes resulting from such refactorings as *extract superclass*, *inline Superclass*, or *extract interface* (see Section 4.2.1.2.2), most of inheritance-hierarchy changes bring about behavior modifications: the client application may compile fine with the new version, however, it may behave differently.

4.2.1.1.7 Program-entity additions and deletions

Table E-3 summarizes the newly added and removed public or protected program entities between compared versions. Clearly, Eclipse grew fast in the past three years. Compared with the corresponding previous versions, the versions 2.1, 3.0, and 3.1 contain 7127, 14095, and 17343 newly introduced packages, classes and interfaces, fields and methods/constructors, respectively. In the mean time, a certain amount of public or protected program entities (much less than the newly introduced entities) were removed, 1298, 4157, and 2455 for version 2.1, 3.0, and 3.1 respectively. The removed public or protected program entities may cause the application to fail to compile. A small fraction of newly added or removed program entities are the results of various “Extract...” or “Inline...” refactorings, as discussed in Section 4.2.1.2. But most of these changes represent newly introduced API or removed obsolete API.

4.2.1.2 “Bigger” refactorings

In this subsection, we discuss “bigger” refactorings, which are composed of a coherent series of elementary changes to a set of related entities. Although, in principle, refactorings should be performed one step at a time, Fowler [32] and Kerievsky [51] demonstrate how a series of “small” refactorings can lead to the “big” changes, such as the introduction of design pattern. By looking at a set of changes as a coherent whole, we may gain a better understanding of the design evolution of a software system and the refactorings it has suffered, and consequently be in a better position to assess the state-of-the-art in tool support for the practice.

The refactoring support that Eclipse provides is representative of the state-of-the-art today. We reviewed the currently available refactoring tools and IDEs

(www.refactoring.com/tools.html) and Eclipse supports a superset of the refactorings supported by each of them. The only interesting exception is IntelliJ IDEA [124], which supports *extract super/subclass* and *replace inheritance with delegation*. Some other tools also support *extract superclass*. However, overall, each of these tools supports fewer types of refactorings than Eclipse.

4.2.1.2.1 Containment-hierarchy refactoring

Large software projects are often organized in terms of subsystems, packages, (nested) reference types; such organization makes the dependencies among the various components explicit and makes it easier to identify the use of a component by its implied container. The developers often restructure the containment hierarchy at different levels.

Table 4-5. Containment-hierarchy refactorings in Eclipse evolution

| Type of refactoring | # detected | Eclipse support |
|------------------------------------|------------|-----------------|
| Convert anonymous class to nested | 12 | √ |
| Convert nested type to top-level | 19 | √ |
| Convert top-level type to nested | 20 | × |
| Move member class to another class | 29 | √ |
| Extract package | 16 | × |
| Inline package | 3 | × |

The Eclipse plugins work as subsystems that contribute different features to the platform. A new plugin may be introduced as the appropriate placeholder for features that were originally placed in other plugins. In version 3.0, three new plugins, `jdt.junit.runtime`, `ltk.core.refactoring` and `ltk.ui.refactoring`, were split from two existing plugins, `jdt.junit` and `jdt.ui` (the `core.refactoring` and `ui.refactoring` folders) respectively; several packages were either moved or extracted into the new plugins.

Package is one way of grouping together related classes depending on their behavioral dependencies. When a package has too many classes to be easily understandable and is not cohesive because these classes are responsible for very different features, a new package may be extracted to hold some important groups of classes. For example, `org.eclipse.jdt.internal.ui.refactoring.reorg` was extracted from `org.eclipse.jdt.internal.ui.refactoring` in the same plugin, and

`org.eclipse.jdt.internal.formatter.comment` in `jdt.core` was extracted from `org.eclipse.jdt.internal.ui.text.comment` in the `jdt.ui` plugin. Other times, a package is removed and its contents may be inlined to other package(s). For example, three classes of the removed package `org.eclipse.jdt.internal.corext.template` were inlined to `org.eclipse.jdt.internal.corext.template.java` package.

Java classes and interfaces can define their own nested types. Sometimes, the top-level types may be converted to nested type of a particular class in order to group together the relevant classes and make the dependencies among them clear. On the other hand, nested types may be converted to top-level so that they are available to other classes. In Java, anonymous classes are widely used to avoid creating a bunch of simple subclasses or implementations of interfaces. However, when the anonymous classes grow so large that the code becomes difficult to read or maintain, they may be converted to nested type.

All these changes can be accomplished by various types of refactorings: convert anonymous class to nested, convert nested (top-level) type to top-level (nested), move member class, and extract or inline package. Three of them are supported in modern IDEs, such as Eclipse, while the other three are not explicitly supported.

4.2.1.2 Inheritance-hierarchy refactoring

Programming to interfaces and not to implementations is an important tenet of object-oriented development [34]. A corollary of the programming-to-interfaces principle is the *extract interface* refactoring. For example, in version 3.1, a new interface `IChangeAdder` was introduced for class `JUnitRenameParticipant` and its two subclasses `ProjectRenameParticipant` and `TypeRenameParticipant`.

When two (or more) classes share a substantial part of their behaviors, their common features may be extracted to a superclass. For example, in version 3.1, a superclass `HierarchyRefactoring` was extracted (involving 57 fields and methods) from `PullUpRefactoring` and `PushDownRefactoring`. When a class defines features that are only applicable in some cases, a subclass may be extracted for that subset of features. For example, a subclass `ImportMatchLocatorParser` was extracted from `MatchLocatorParser`, which holds two methods that are used only for compilation unit.

Table 4-6. Inheritance-hierarchy refactoring in Eclipse evolution

| Type of refactoring | # detected | Eclipse support |
|------------------------------|------------|-----------------|
| Pull up field/method | 279 | √ |
| Push down field/method | 53 | √ |
| Extract interface | 33 | √ |
| Extract superclass | 15 | × |
| Extract subclass | 4 | × |
| Inline superclass/subclasses | 11 | × |

Collapsing hierarchies is another important refactoring that deals with generalization. When a superclass does not deliver much functionality or a subclass is not that different from its superclass, the two may be merged. For example, in version 2.1, the superclass `BufWriter` was inlined into subclass `VerboseWriter`; in version 3.0, three subclasses `MemberTypeDeclaration`, `LocalTypeDeclaration`, and `AnonymousLocalTypeDeclaration` were inlined into their superclass `TypeDeclaration`.

Finally, within the inheritance hierarchy, common fields and methods of subclasses were pulled up to the superclass, while the fields and methods that were only applicable to some subclasses were pushed down to them.

4.2.1.2.3 Class-relationship refactoring

Object-oriented systems are designed around classes that model abstractions of real-world entities. Classes collaborate with each other to deliver the application functionalities.

Table 4-7. Class-relationship refactoring in Eclipse evolution

| Type of refactoring | # detected | Eclipse support |
|----------------------------|------------|-----------------|
| Extract constant interface | 5 | √ |
| Inline constant interface | 2 | × |
| Extract class | 95 | × |
| Inline class | 31 | × |

In Java, interfaces are often used to define static final constants; the classes may implement them to access the constants or access them in the static way. For example, in version 2.0, class `JavaPartitionScanner` and `FastJavaPartitionScanner` used to define four same constants, which were extracted to a new interface `IJavaPartitions` implemented by

the two classes in subsequent release 2.1. This refactoring also removed the duplication. When the constants are only used by a single class and its subclasses, the interface may be inlined. For example, in version 3.1, the constant interface `BindingIds` was removed and the constants it defined were inlined to the class `Binding`.

Complex classes are sometimes incohesive because they are responsible for delivering many responsibilities. Such classes should be simplified by extracting some of their features into other classes, created for exactly that purpose. The simplified class can then delegate to the newly created class to deliver its responsibilities. For example, in version 3.0, a new class `DeltaProcessingState` was extracted from `DeltaProcessor`; `DeltaProcessor` newly declared a field of type `DeltaProcessingState`, to which it delegates the maintenance of the global state of delta processing.

Another frequent case involves the extraction of helper or utility class. For example, the helper class `RefactoringExecutionStarter` was extracted from `ReorgMoveAction` in version 3.1.

When a class does not have many responsibilities, its features may be inlined. For example, class `ReferenceScopeFactory` that used to define a single public method creating an instance of `IJavaSearchScope` was inlined to `JavaSearchScopeFactory` in version 3.1. Sometimes, the helper class may be inlined to the class depending on it. For example, `SuperReferenceFinder` was inlined into `PullUpRefactoring`.

Developers often introduce new entities before they realize that similar features already exist. In such cases, the inline-class refactoring can be used to remove duplication. For example, in version 2.1.3, there were three `Util` classes scattered in three packages; in version 3.0, they were inlined into a single class.

4.2.1.2.4 Internal class refactoring

Eclipse supports various types of refactorings that reorganize the code within a class, including, *use supertype where possible*, *introduce factory method*, *change method signature*, and *extract or inline method*. We identified a large number of such refactorings in Eclipse's evolution history. However, Eclipse does not support the refactorings of *information hiding*, *encapsulate downcast*, *introduce parameter object*, which also often being applied.

Table 4-8. Class-internal refactorings in Eclipse evolution

| Type of refactoring | # detected | Eclipse support |
|-----------------------------|------------|-----------------|
| Information hiding | 751 | × |
| Use supertype when possible | 177 | √ |
| Encapsulate downcast | 85 | × |
| Introduce factory method | 19 | √ |
| Change method signature | 4497 | √ |
| Introduce parameter object | 4 | × |

On the other hand, Eclipse supports several refactorings that change the code within a method, such as *extract local variable*, *extract constant*, *introduce parameter*, *convert local variable to field*. However, at the current stage, our analysis does not take into account the statement-level information regarding the usage of local variable so that it does not support the analysis on these statement-level refactorings.

4.2.1.3 Design-change sequences

Finally, let us look at the program entities that undergo two or more types of changes. 27% (2104/7851) of the modified entities underwent two or more types of changes. We have already discussed several such cases in Section 4.2.1.1, including renaming program entities to reflect their data-type change; renaming to conform to a naming convention for static final fields; moving methods and using an additional parameter of the type of the “old” home class; moving program entities and changing their visibility correspondingly.

However, by closer inspection, we noticed that in most cases, subsequent modifications to an entity were not inherently related, such as “bigger” refactorings discussed in Section 4.2.1.2. For example, in version 3.1, `ASTParser.convert()` was moved to class `CompilationUnitResolver`; its visibility changed from private to public; it was newly declare with static; its return type was downcast from `ASTNode` to `CompilationUnit`; and its signature was modified to take three more parameters as input.

4.2.2 Analysis of the case-study findings

Conducting this comprehensive study has given us some interesting insights into the design evolution of object-oriented software system. We discuss them in this subsection.

4.2.2.1 Refactoring is a frequent practice

Refactoring is indeed a common activity in the development process of object-oriented software. In recent years, refactoring has been popularized in object-oriented software development, especially in the context of agile, lightweight development processes such as “Extreme Programming” [8]. However, it is not clear how prevalent refactoring is actually in practice. In our Eclipse-evolution case study, there were 58973 (see Table E-3) changes reported by *UMLDiff*. Most of the radical design and implementation changes were made in the major releases 2.1, 3.0, and 3.1. A considerable amount of new features were introduced, and many existing features were redeveloped with a totally different implementation, such as the AST-rewrite feature. We excluded from our analysis about 75% of all the changes that, according to our understanding – based on code inspection, the *UMLDiff* result, the help document, and the Javadocs comments – represent the introduction of new features or the removal of obsolete API.

When considering the remaining changes, over 70% of them were the results of refactoring or a sequence of refactorings, including renamings, moves, downcasting or use of supertype, information hiding, reorganizing containment or inheritance hierarchy, changing the relationships among classes, and changing the code within a class, as disused in Section 4.2.1. We cannot know whether they represent changes resulting from intentional refactorings or they are just accidental. Overall, about 17% of all the changes (including adding and removing) can be expressed in terms of “standard” refactorings, which we believe is an indicator that a considerable amount of effort has been spent on intentionally restructuring the existing system in the evolution of Eclipse.

This is evidence that a refactoring engine would be a valuable functionality for the development environment in order to provide (semi-)automatic refactoring support to developers instead of them having to perform refactorings manually.

4.2.2.2 Support is still missing for many types of frequently applied refactorings

Modern IDEs, such as Eclipse, support the most commonly used, low-level refactorings, including renaming, move, and using supertype. But they do not support *encapsulate downcast* and *information hiding* refactorings, which our case study shows are also frequently applied. Especially for the *information hiding* changes, we found out that a

class may have several members to hide; manually hiding all of them could be error-prone.

Eclipse supports moving static fields and methods to a specified type, but it treats moving instance fields simply as a textual move and the references to the moved instance fields will not be updated. Furthermore, Eclipse only supports moving instance methods to types of its parameters or types of fields declared in the same class as the method. The Eclipse “pull up” and “push down” refactorings support moving instance fields and methods to their direct superclass or subclass. However, in our case study, instance fields and methods may be moved to any type, which may or may not be directly related to their current declaring class.

Eclipse supports some of the “bigger” refactorings discussed in Section 4.2.1.2, but it lacks support for the refactoring of the containment and inheritance hierarchies and general class relationships. Suppose, for example, that we want to extract a helper class C that contains an instance method M declared in D. With current tool support, the developer may perform the following activities: create a new class C; declare a new field F of type C in class D; move M to C and then may remove field F. It seems that copy and paste would be an easier solution. However, as summarized in [54], about 22% of the copies the developer leaves off-screen references unchanged or only copies part of the code being distributed within several files.

Based on our findings of the refactorings actually applied to Eclipse throughout its evolution history, an effective refactoring tool should support the following (in addition to what are commonly supported in current IDEs):

- information hiding refactoring, such as “hide a group of method in a class”,
- more flexible move of instance field and method in terms of object-oriented entity instead of simply text;
- a refactoring user interface to collect the information about more complex refactoring tasks, such as those refactoring inheritance-hierarchy.

4.2.2.3 The reuse-based software development can benefit from refactoring-migration tools

Eclipse is built as a plugin-based framework. It is an IDE as well as a software development kit (SDK). The developers can build their own plugins by extending the existing ones and then integrate them into Eclipse. Our JDevAn tool [125] is one of such plugins we have developed in our research group. Even for such a small-size research prototype, we have suffered from breaking API changes as the underlying Eclipse platform evolved.

In the last section, we discussed that over 70% of design changes can be expressed in terms of refactorings from the perspective of the Eclipse framework developers. To them, a refactoring, such as *move method*, affects only the structure of the software and not its behavior. However, it is simply impossible for Eclipse developer to update all the third-party plugins built on it when they refactor the code. Thus, to third-party plugin (i.e., framework-based client application) developers, such a refactoring may be a breaking change, which indicates that they have to migrate their code to the new version of Eclipse. Such migration is often perceived as disturbing.

However, our case study shows that, for over 60% of design changes that may be due to refactorings, the references to the affected entities in client applications can be automatically updated by a refactoring-migration tool if the relevant information of refactored components were properly gathered. This indicates that a refactoring-based development environment can benefit a lot from refactoring-migration tools, such as CatchUp [42]. However, the refactorings that CatchUp can record and replay are only renamings and moves. These account for about 70% of the tedious updating tasks that may be handled automatically for applications that use the refactored components. However, there exist several other frequently used low-level refactorings, such as “information hiding”, “encapsulate downcast”, which CatchUp cannot support. Furthermore, refactoring-migration tools are unaware of the impact of “bigger” refactorings.

4.2.2.4 Tools should implement refactorings using the command and composite patterns

The question then becomes: “What might an appropriate internal representation for refactorings be, such that it would enable a tool to meet the above requirements?”

As discussed in Section 4.2.1.3, about 27% of all the program entities that have been modified underwent two or more types of changes, which can be any combination of the elementary design changes discussed in Section 4.2.1.1. Furthermore, there were about 370 “bigger” refactorings that have been applied to refactor the containment, inheritance, class relationships and class internals. These “bigger” refactorings are composed of a series of coherent related structural changes to a set of relevant entities.

These facts imply that a good possible implementation of an automated refactoring functionality would be to view a design change as a command object: thus, simple refactoring commands could be composed into larger ones [34] and they could also be done, undone and replayed. For example, 264 methods (see Table 4-2) moved methods change their identifiers and/or parameter lists as well. Suppose that a method is moved and then one of its parameters is removed. These changes can be stringed together as a `MoveMethodCommand` followed a `RemoveParameterCommand`, which are contained in a `CompositeCommand`. A memento object [34] may be used to record which parameter is removed. As another example, consider the “Extract Superclass” refactoring: it can be an instance of `CompositeCommand`, composed of a `NewClassCommand`, a `ModifyClassInheritanceCommand`, and several `PullUpCommands`. A `PullUpCommand` can further be an instance of `CompositeCommand`, which may be composed of a `MoveFieldCommand` and a `ModifyVisibilityCommand`.

The other benefit with command objects is that they can be executed at different times [34]. The refactoring tool can record the command objects and replay them (if possible) on the applications that reuse the refactored components. Such “refactoring deferral” would effectively constitute refactoring migration.

4.3 Summary

In this chapter, we presented our method for detecting refactorings in the evolution history of an object-oriented software system. Instances of refactorings are detected as

compositions of design changes reported by *UMLDiff*. This method is precise: it reports the instances of detected refactorings in terms of their particular types and their key participant elements and relations. It is robust to “multiple-changes-to-same-entity” issue. This method is general: it is able to recognize a broad range of “standard” refactorings and it can be extended with special-purpose queries for detecting change patterns of interest to the user. A refactoring-detection capability, such as ours, could potentially be a helpful utility: it could recognize design changes to component frameworks when the refactoring is not supported by the refactoring tool or is not applied explicitly through the tool, which could then be composed and replayed in order to propagate them in the context of client applications.

Chapter 5: API-Evolution Support with Diff-CatchUp

Software reuse simplifies the design of new systems but, at the same time, it implies that their design and implementation heavily depends on the components they reuse. Stable interfaces to a reusable component framework (or library) isolate the client application from changes in those components, under the assumption that the components' developers limit themselves only to extending – as opposed to changing – the components' application programming interface (API). In practice, however, this assumption is frequently violated; the new components' versions change their APIs and, as a result, the applications that rely on them may fail.

The fundamental challenge in evolving applications built on reusable component frameworks is the fact that these applications and their underlying component frameworks are subject to two independent, asynchronous, and potentially conflicting evolution processes. The scope of the first process is the component framework and is driven by the need to improve the framework functionality and quality, while maintaining its generality. The second evolution process is motivated by the more specific requirements and desired qualities of the application's stakeholders. Although, there exists extensive software-engineering research on methods and tools for supporting evolutionary development, such as refactoring in object-oriented software development [32,51,72], they usually rely on the assumption that the entire software is accessible by the tool. However, it is simply impossible for component-framework developers to access and update all the client applications, and it is ill advised for application developers to modify the components that they reuse – even when they have access to their source code – because that would essentially defeat the reuse motivation.

This challenge gives rise to the question “How can one specify changes to the reusable component's APIs that may impact reuse and support their consistent propagation?” API changes that render components obsolete may be documented through programming-language syntax or in a framework-specific style [119,121]. However, such documentation may be incomplete and does not come with any programmatic support for accommodating the changes in the application code. To our knowledge, there are only two methods that do support the application developers' migration task: the first assumes

that the component developers will provide complete transformation rules for all API-breaking changes [15] and the second assumes that the component framework and application developers use the same development environment [42,50] so that the recorded changes can be interpreted and replayed.

In Chapter 4, we discussed our method for detecting refactorings by querying the *UMLDiff* design-change facts and we reported on our study on the pattern of changes in the evolutionary history of Eclipse [116]. Based on the assessment of empirical data, we found that more systematic support could be developed for propagating framework API changes to the applications using it. In this chapter, we present our Diff-CatchUp approach to tackling the API-evolution problem in the context of reuse-based software development. This approach assumes that the component framework itself represents good usage of its evolving APIs. It relies on *UMLDiff* to recognize the evolution of the APIs of a component framework. Once the specific API changes have been identified, our approach supports the migration of client applications to appropriately use the evolved APIs, based on “voluntary” working examples of the framework code base. This approach has been implemented in the Diff-CatchUp tool. We report on two case studies that we have conducted to evaluate the effectiveness of our approach with its Diff-CatchUp prototype.

5.1 An illustrative example

Let us consider a developer who is reusing the version 0.9.4 of JFreeChart [126], to implement a visualization application. Three relevant JFreeChart classes, `PlotFit`, `LinearPlotFitAlgorithm`, and `MovingAveragePlotFitAlgorithm`, are contained in the `com.jrefinery.chart.data` package. In version 0.9.4, these three classes work together to produce a `XYDataset` object: first, a `PlotFit` object is constructed with a `XYDataset` object and an instance of either `LinearPlotFitAlgorithm` or `MovingAveragePlotFitAlgorithm`, and then, a modified `XYDataset` object is produced, which is used to create a chart. When the developer attempts to build the application with version 0.9.5, the compiler complains that (a) the import `com.jrefinery.chart.data` cannot be resolved, (b) `PlotFit`, `LinearPlotFitAlgorithm`, and `MovingAveragePlotFitAlgorithm` cannot be resolved to a type, and (c) the method `getFit()` is undefined for the type `PlotFit`.

Looking at the release notes shipped with the new version 0.9.5, the developer can only find the sentence “introduced new `MovingAverage` class”, which might be relevant since one of three broken classes is named `MovingAveragePlotFitAlgorithm`. Unfortunately, the documentation does not provide any information about what happened to the `com.jrefinery.chart.data` package and the `PlotFit`, `LinearPlotFitAlgorithm`, and `MovingAveragePlotFitAlgorithm` classes, and how the newly introduced `MovingAverage` class might be related to the `PlotFit`-related classes. At this point, the developer is probably uncertain of the next step needed to complete the migration task.

Let us discuss how our `Diff-CatchUp` tool can help in this situation. Highlighting the offending code (e.g., the call to the `PlotFit.getFit()` method) causes the tool to identify the broken API involved in the problem (i.e., the `PlotFit.getFit()` method) and to search the repository of the evolving `JFreeChart` library for the changes to this API (*UMLDiff* reports that `PlotFit.getFit()` was removed in version 0.9.5). Next, `Diff-CatchUp` attempts to locate plausible replacements for the removed `PlotFit.getFit()` by examining the methods that used to call `PlotFit.getFit()` and are not broken in version 0.9.5 and formulates a set of replacement proposals for the developer. For example, it suggests that the `MovingAverage.createMovingAverage(XYDataset,...)` method may be used to replace the removed `PlotFit.getFit()` method since they both declare the same return type. It also recommends several usage examples, one of which is the `JFreeChartDemoBase.createCombinedAndOverlaidChart1()` method, which demonstrates two ways to obtain a `XYDataset` object that might be used as the first argument of an invocation to the replacing method. Upon the developer’s request, `Diff-CatchUp` reports the textual differences between the versions 0.9.4 and 0.9.5 of the `createCombinedAndOverlaidChart1()` method, which clearly demonstrate how to migrate the application code that relies on the old API `PlotFit.getFit()` so that it uses the replacing API `MovingAverage.createMovingAverage(XYDataset,...)`.

5.2 API-evolution catch-up

Our API evolution catch-up approach does not require any additional information provided by the component developers. The API changes are automatically recovered with the *UMLDiff* algorithm, given the old and new versions of a component framework.

The client-application developers review the migration problems that they encounter when building their application with the new version of the component framework, as reported by the compiler, and select to work on one (Section 5.2.1). In response, the Diff-CatchUp tool searches the logical-model and API-change facts of the evolving component framework (a) to determine the changes to the broken API of the offending component involved in the migration problem (Section 5.2.2), (b) to identify plausible replacements of the broken API in the new version of the component framework (Section 5.2.3), and (c) to collect examples of how these replacements have been used in the code to deliver what the broken API used to do (Section 5.2.4). Finally, it proceeds to form and present specific migration proposals to the developers (Section 5.2.5).

5.2.1 Selecting an API migration problem

When the client-application developers decide to import an evolved component framework, they have to build their application with the new version of the framework, which may result in various types of problems being reported. They have to resolve all the problems before they can successfully build and retest the application. As a first step, the developer must select some fragment of source code or some compilation error/warning as the locality of the migration problem to be addressed. Table H-1 summarizes the API migration problems that Diff-CatchUp can currently handle. Migration problems may be caused by the removal, renaming¹², or move of the API element involved in the problem, changes to its attributes, such as visibility, modifiers, deprecation-status, and changes to its relation to other elements, such as associated data type, declared exception, inheritance hierarchy.

As can be seen in Table H-1, the correlation between migration problems and the API changes that cause them may be many-to-many. Furthermore, data-type compatibility and polymorphism introduce several technical issues when determining the actual broken API element involved in a migration problem of “undefined method/constructor/field” and “parameter mismatch”. Resolving such issues may involve a significant amount of

¹² For method/constructor, renaming may involve identifier change and/or parameter list changes.

compiler-related work. Therefore, to obtain the actual broken API element whose change causes a given migration problem when building the client application with the new version of a component framework, Diff-CatchUp resorts to a successfully built copy of the client application with the old version of the component framework. Given a selected API migration problem, the contextual information about the involved compilation unit and the start and end positions associated with the problem is extracted, which is then used to access the copy of the successfully built client application to retrieve the actual broken API element involved in the migration problem.

5.2.2 Determining the changes to a broken API

Given a broken API element involved in a migration problem, the question becomes to determine what changes the API has undergone that cause the migration problem. Instead of resorting to the documentation, change specification, or recorded refactoring scripts provided by component-framework developers, our approach relies on the API changes that are automatically recovered by the *UMLDiff* algorithm.

Let us briefly discuss the *UMLDiff* algorithm here. Interested readers are referred to Chapter 3 for the detailed discussion. *UMLDiff* is a heuristic algorithm for automatically detecting the changes that the logical design of an object-oriented software system has gone through, as the subject system evolved from one version to the next. *UMLDiff* takes as input two models of the logical design of the system, corresponding to two of its versions. The underlying meta-model is defined according to the UML semantics [69]. *UMLDiff* traverses the two models in parallel, moving from one type of model elements to its children types; as it does so, it identifies corresponding elements, i.e., model elements that correspond to the same conceptual design entity in two compared models, based on their *lexical* and *structural similarity*. It produces as output a set of change facts (summarized in Table 3-2), reporting the differences between the two versions of the logical model in terms of (a) additions, removals, moves, renamings of subsystems, packages, classes, interfaces, fields and methods/constructors, (b) changes to their attributes, and (c) changes to the relations among these model elements. When adapting a client application to the API changes of the underlying component framework, Diff-CatchUp searches the API-change facts reported by *UMLDiff* algorithm to determine

what changes have been made to the existing component API, which have consequently resulted in the migration problem.

5.2.3 Proposing replacements for a changed API

At this point in the process, the client-application developers know how the broken API of a component framework has been changed. The next question is to decide what plausible replacements to the changed API may exist in the new version of the component framework. Table 5-1 summarizes the actions that Diff-CatchUp takes for adapting different types of API changes.

Table 5-1. The Diff-CatchUp actions for adapting different types of API changes

| Types of API changes | Diff-CatchUp actions |
|----------------------------------|--|
| Renaming or move | Return renaming or move counterpart(s) <i>UMLDiff</i> identifies |
| “Removal” ¹³ | Search logical-model and API-change facts for replacing APIs |
| Changes to attribute or relation | Visualize in JDevAn Viewer (see Chapter 7) for further exploration |

5.2.3.1 Renamed or moved API

For migration problems caused by the renaming or move of API elements (bold \surd in Table H-1), Diff-CatchUp simply returns their counterpart element(s) in the new version, as identified by *UMLDiff*, which serve as the plausible replacement(s) to the changed API. If there are multiple counterpart elements, such as several move-target elements for a move-source element, these elements are sorted by their *UMLDiff* overall similarity metrics. If the application developer is not satisfied with the mapped element counterparts returned, he can explicitly request the given API element to be processed as “removed”, according to the process detailed in the next subsection.

¹³ The “removed” element includes the actually removed element, the deprecated, visibility-restricted, and class-made-abstract element, and the mapped element that the developer explicitly request to be processed as removed.

5.2.3.2 “Removed” API

For migration problems caused by the removal of API elements (bold \checkmark in Table H-1), Diff-CatchUp searches the logical-model and API-change facts to generate plausible replacement(s) to the removed API. The deprecation of an API indicates that something is obsolete and the component developers do not want their users to continue programming to the old API. Thus, a deprecated API element (italic \checkmark in Table H-1) is processed in the same way as a removed element. Diff-CatchUp also treats as removed an element whose visibility is restricted causing the “not visible” problem, and a class that is made abstract causing the “invalid class instantiation” problem.

Note that the mapping between a “removed” API and its replacements is not necessarily one-to-one. Several “removed” APIs may have been replaced by a single API, or a single “removed” API may have a few different substitutions, or the roles of several APIs may have been replaced by another set. As demonstrated in Section 5.3.1, one advantage of our approach is that it does not place any constraint on the mappings between the broken APIs and their plausible replacements; all potential replacements are selected, ordered by their relevancy, and presented to the developer for consideration.

The underlying intuition to recommending replacing API(s) for a “removed” API is that the places that a “removed” API used to be used should use its replacing API(s). Thus, our Diff-CatchUp approach takes the following four steps to propose the replacement(s) to a “removed” API element E .

1. It collects all the mapped user elements U that used to use E but no longer do so.¹⁴
2. It collects as candidates all elements C that U newly uses or continues using.
3. It examines the heuristics (discussed in detail below) between the “removed” element E and the candidate C , and selects one as a plausible replacement R if the set of valid heuristics is not empty; if no candidate is qualified with some valid heuristics, it selects as plausible replacements those candidates that are newly used by U .

¹⁴ The term “use” refers to some types of relationships between two elements, which are defined in Table 5-2.

4. It orders the selected replacements R according to their $UMLDiff$ status, the number of valid heuristics, and the support of R in terms of the number of user elements that use R divided by the number of all user elements.

Table 5-2. Proposing replacements for a “removed” API

| Element | No-longer users | Candidates | Plausible replacements |
|----------------------|--|--|---|
| Class & Interface | Instantiate Class usage Inheritance Data type | Instantiate Class usage Inheritance Data type | Same name Inheritance or sibling Usage dependency Move of children |
| Method & Constructor | Call | Call, read and write | Overriding, overloading or same signature Declared in same or super- and subtype, or declared in sibling types Extract/Inline operation or usage dependency Compatible data type |
| Field | Read (write) | Call and read (write) | Same name Declared in same or super- and subtype, or declared in sibling types Usage dependency Compatible data type |

Table 5-2 summarizes the relations that Diff-CatchUp examines to collect the mapped user elements U that no longer use the “removed” API element E (the second column) and to collect the candidates C that U newly uses or continues using (the third column). For a “removed” reference type T , the mapped users U include the methods/constructors that used to instantiate T , the methods/fields whose return/data type used to be T , and the reference types that used to use the members declared in T and those types that used to extend or implement T . The candidates are the classes/interfaces that are related to the users U with the corresponding type of relationship in the new version. For a “removed” method or constructor M , the mapped users U are the methods/constructors that used to call M . The candidate elements include the methods/constructors and fields that U calls, reads and writes in the new version. For a “removed” field F , the mapped users U are the methods/constructors that used to read (or write) F . The candidate elements include the methods/constructors and fields that U calls and reads (or writes) in the new version.

Note that the candidates and the subsequently selected plausible replacing APIs are not necessarily of the same element type as the broken API. For example, a superclass may be replaced by some of its super-interfaces; instantiating a certain type directly (a constructor call) may be encapsulated into a method that checks some pre-conditions and returns the object if everything is correct. Furthermore, for a removed package, our approach does not propose replacing package(s); instead, it determines the changes that have been made to the class or interface involved in the problematic import declaration or qualified name and proposes the replacement(s) for the involved classes and interfaces, depending on their changes.

Given a set of candidate elements, four sets of heuristics (the fourth column in Table 5-2) are examined to select the most plausible replacing API elements from the identified candidates. The heuristics examine four different aspects between the broken API element and the candidate replacing element, including (a) their names, (b) their inheritance relations, (c) their usage dependencies, and (d) their associations with other elements (e.g., associated data type). Each set of heuristics may contain one or more heuristics in a decreasing order of priority. If a high-priority heuristic is true, Diff-CatchUp stops examining the remaining ones in the set. For example, if a “removed” class and its candidate replacing class have direct or transitive inheritance relation, Diff-CatchUp will not examine whether they are sibling classes that share some common ancestor types. If there is no candidate qualified with some valid heuristics as replacement, those candidate elements that are newly used by the user elements are selected as plausible replacing elements, with the assumption that the newly used elements would most likely be the substitutions to a “removed” API, i.e., “newly used” heuristic. Let us now review these heuristics for selecting plausible replacing API elements.

NAME: This is the simplest of the heuristics. Name is a “safe” indicator that the “removed” API element may be related to its potential replacements, especially when a consistent and meaningful naming scheme is adopted. For two classes, interfaces, or fields, Diff-CatchUp simply examines whether they declare the same identifier. For two methods or constructors, it examines whether one overrides the other directly or transitively (same-signature and declared in super- and subtype), whether one overloads

the other (same-identifier and declared in the same type), or whether they declare same-signature and are declared in different types with no inheritance relation.

INHERITANCE: Component developers sometimes reorganize inheritance hierarchies, for example, by extracting superclasses, pushing down (or pulling up) methods, or forming template methods [32]. Searching along inheritance relations may reveal replacing elements that are declared in super- or subtypes of a “removed” API. In addition to elements with direct or transitive inheritance relations, sibling types that share common ancestor types with the removed element may also be examined. Such sibling types indicate that the concerned types declare similar interfaces and deliver similar behavior. The more common ancestors they share, the more similar their interfaces and behavior is likely to be. The methods and fields that are declared in such types may be interchangeable.

USAGE: Component developers sometimes restructure the usage dependencies between objects. Examining the usage dependency between model elements may reveal what can be used to substitute a “removed” API. For example, a middle-man API is removed and its users start accessing directly the features that they used to delegate to the middle man; several small steps are merged into a bigger one that executes these steps internally instead of within the control of their original users; a method is deprecated and its body is extracted into a new method to which it delegates. Note that *UMLDiff*, in its differencing process, detects the redistribution of the behavior among operations, by analyzing the removals and additions of usage dependencies of the mapped operations and the related removed or added operations along their transitive usage and inheritance relations, and reports such behavior redistribution in terms of *extract operation* and *inline operation* changes. The extract/inline operation change facts, if they exist, are preferred over the ordinary usage dependency between two methods/constructors.

ASSOCIATION: An association is a declaration of a semantic relation between model elements, such as the associated data type of a method or field and the declaring type of a constructor. Diff-CatchUp examines whether the “removed” and potential replacing methods, constructors, and fields declare compatible data types. APIs that have compatible data types with a “removed” API may be used to replace the role of the “removed” API in its users. Note that a constructor does not explicitly declare a return

type; instead, its declaring type is used. The compatible data type is defined as the same type or the super- and subtype (direct or transitive). Currently, our approach does not handle such compatible types as int and double, character array and String object, etc.

For a “removed” reference type and its potential replacing types, one more special heuristic, i.e., the move of children element between them is examined, since the new host of the moved features may be a good substitute for their original declaring type.

Finally, the selected replacements are ranked according to their *UMLDiff* status, the number of heuristics that apply to them, and their support. The *UMLDiff* status of the replacing element of a “removed” API can be newly added or mapped. Our approach prefers the newly added replacing elements to the mapped ones. For the replacing elements with the same *UMLDiff* status, they are further ordered by the number of their valid heuristics and their support. Currently, the same weight (i.e., 1) is assigned to all types of heuristics, which means that any replacing API can have a maximum heuristics score of 4 in the current implementation. The replacing elements are finally sorted by their name-similarity with the “removed” API in terms of longest common subsequence of their names.

5.2.3.3 Changes to attributes or relations

Diff-CatchUp does not attempt to suggest ways to adapt to migration problems (regular \surd in Table H-1) caused by changes to the attributes of model elements or changes to relations among elements, since it is hard to guess how the application developer wants to adapt the changed API. For example, a method that used to return a primitive type is changed to return an object that encapsulates the original primitive value along with several new values. The application developer may retrieve the original primitive value from the returned object or may decide to preserve the whole object since the original value and the other newly added values should be used together. It is difficult for an automatic process to infer which option is more appropriate in a particular context. Therefore, Diff-CatchUp simply presents the broken API element and its corresponding attribute or relationship change to the application developer. The developer can then interactively explore the neighborhood of the concerned API element through JDevAn

Viewer (see Chapter 7) to build up the knowledge about how the changed API can be used.

5.2.4 Recommending usage examples of a concerned API

Having formulated a hypothesis regarding the API elements that can potentially replace a broken API is only part of the story when adapting client application to the API changes of a component framework. The application developer still needs assistance on how to use those replacing APIs. Diff-CatchUp does not provide usage examples at class granularity, i.e., how a class is used by other classes as a whole, since such examples are too coarse-grained and cannot effectively help the application developer learn how to use a particular method, constructor, or field declared in that class. Furthermore, not all proposed replacing methods, constructors, and fields require usage examples, for example, the move of a static field, the renaming of a method that involves only the change of its declared identifier or the removal of some parameters. We identify the following three cases in which usage examples of the concerned API are useful:

1. a non-static method/field – the developer needs to know how to obtain the object of the declaring type of the relevant method/field in order to refer to it (*obtain-object usage example*);
2. a method/constructor declares one or more matched, type-changed and newly added parameters – the developer needs to know how to obtain the object or value of the concerned parameter in order to invoke the method/constructor (*parameter-list usage example*); and
3. a mapped replacing method/constructor/field of a “removed” API – the developer needs to know how to migrate from the “removed” API to its replacing API (*replacement usage example*).

An API of concern may require several different types of usage examples at the same time. For example, in the case that a moved non-static method that declares a new parameter is proposed as a replacing API for some removed method, all the three types of usage examples may be necessary. The replacement usage example is only applicable to the mapped replacing APIs of a “removed” method, constructor, or field. However, the obtain-object and parameter-list usage examples are applicable to a qualified API element,

whether it is a replacing API of a renamed, moved, or “removed” API, or it is a usage-example element. It is important that the developer is able to request further usage examples for the recommended usage-example elements, in some cases. For example, a usage example of a replacing API is a moved non-static method. The developer may then want to know how to get the object of the declaring type of this moved method. As another example, suppose that the method $m()$ is renamed to take one more parameter of some type T , all direct usage examples that are recommended take one argument of type T , which they use to invoke renamed method $m(T)$. These usage examples do not actually show how to obtain the concerned object of type T . The developer can then iteratively request more usage examples, until one is found that demonstrates the appropriate ways to construct the object of type T .

5.2.4.1 Obtain-object usage example

First, let us discuss obtain-object usage examples that demonstrate how to get the concerned object in order to invoke one of its methods, or reference one of its fields. The same procedure is used to get the concerned object or value to invoke a method or constructor with it as one of the method’s arguments, which serves as the building block of parameter-list usage example discussed in 5.2.4.2.

Table 5-3. Input parameters for recommending obtain-object usage example

| Concerned API | T_{new} in new version | T_{old} in old version |
|------------------------------|-----------------------------|---|
| Method/Field of case 1 | Current declaring type | Previous declaring type if moved, null otherwise |
| Method/Constructor of case 2 | Current type of a parameter | Previous type of this parameter if type-changed, null otherwise |

Table 5-3 lists the input parameters based on which obtain-object usage examples are identified. Given an API element (method, constructor, or field) E , all the methods/constructors M that use (call, read or write) E are collected. Then, all elements E' (not equal to E) that M uses – the methods that M invokes, the fields that it references, the objects that it instantiates, and the parameters that it declares – are examined. If E' is of the same (sub-) type as the relevant type T_{new} of element E , then it is recorded as a possible way to get the concerned object of type T_{new} in order to use E . If M uses one or

more such E' , it is selected as one valid obtain-object usage example element with one or more possible ways to obtain the relevant object. Optionally, if M is mapped and the relevant type T_{old} of element E is not null, all the elements that M uses whose associated type is of the same (sub-) type of T_{old} are also collected. This knowledge is used to rank the recommended obtain-object usage examples.

The obtain-object usage examples are sorted according to the *UMLDiff* status of the usage-example element M , the status of the relation between the usage-example element and the concerned API element E , and the number of effective ways E' to obtain the concerned object of type T_{new} . Furthermore, our approach prefers the mapped usage-example elements that have the newly added ways to obtain T_{new} object and no longer existing ways to obtain T_{old} object, since they demonstrate how to migrate from the old API to the new one.

5.2.4.2 Parameter-list usage example

A method or constructor may declare one or more matched, type-changed, and newly added parameters. Each parameter may have its own set (possibly empty) of obtain-object usage examples, which consists of distinct usage-example elements that show the ways to obtain the proper argument for this particular parameter. The sets of obtain-object usage examples for different parameters may intersect, when one usage-example element demonstrates how to obtain arguments for more than one parameter. For a method/constructor, the sets of obtain-object usage examples of its parameters are merged into a single set of parameter-list usage examples, by combining several obtain-object usage examples that share the same usage-example element into one parameter-list usage example. Thus, a parameter-list usage example is composed of obtain-object usage examples for the corresponding parameters that the developer specifies when requesting usage example for a method or constructor, which share the common usage-example element and demonstrate how to invoke a method or constructor with one or more proper arguments.

For a method/constructor, its parameter-list usage examples are sorted according to the *UMLDiff* status of usage-example element, the status of the relation between the usage-

example element and the concerned method/constructor, and the number of parameters whose usage a particular usage example demonstrates.

5.2.4.3 Replacement usage example

Replacement usage examples are relevant only to the mapped replacing APIs of a “removed” method, constructor, or field, and are meant to show how to migrate application code from using the “removed” API to its replacing API. Note that, the newly added replacing APIs of a “removed” API element can be illustrated through the obtain-object and parameter-list usage examples; they do not need replacement usage example. Furthermore, some of the replacing APIs of a “removed” element may be qualified for obtain-object and parameter-list usage examples. In addition to these examples, the replacement usage examples that demonstrate how these replacing APIs are generally used can also be provided.

Given a mapped replacing API R of a “removed” element (method, constructor, or field) E , all the methods/constructors M that use (call, read, or write) R are collected as its replacement usage examples. Optionally, if the status of M is mapped, Diff-CatchUp also examines if M uses the “removed” element E in the old version. The replacement usage examples are sorted according to the *UMLDiff* status of usage-example element M and the status of the relation between the usage-example element and the replacing API R . Furthermore, our approach prefers the mapped usage-example elements M that newly use the replacing API and no longer use the “remove” API, since they demonstrate how to migrate from the “removed” API to the one that replaces it.

5.2.5 Presenting replacement and usage example proposals

Finally, given the plausible replacing APIs or usage-example elements, a list of migration proposals is formulated. A replacing API proposal consists of (a) the broken API element, (b) the replacing API element, (c) the heuristics for why this replacing API element was selected (a set of textual rationale descriptions), (d) the relevant model elements and relations collected when examining these heuristics; (e) the user elements and their relations with the broken API element and the replacing element; and (f) the changes to all the above elements and relations as reported by *UMLDiff*.

An obtain-object or parameter-list usage example proposal consists of (a) the concerned API element and its relevant types for which the developer selects to see usage examples; (b) the usage-example element and its relations with the concerned API; (c) the elements that represent possible ways to get the concerned object and their relations with the usage-example element; and (d) the changes to these elements and relations as reported by *UMLDiff*.

A replacement usage example consists of (a) the “removed” API element and its replacing API element; (b) the usage-example element and its relations with the “removed” and replacing element; and (c) their changes as reported by *UMLDiff*.

The list of generated replacement and usage example proposals is sorted and returned to the application developer for inspection. Our approach allows developers to customize the ordering priorities when inspecting the returned proposals. Furthermore, Diff-CatchUp is supported with our interactive visualization framework, the JDevAn Viewer (see Chapter 7), which illustrates the relevant API elements, relations and their changes in an UML-style class diagram. JDevAn Viewer enables developers to inspect the detailed model and change information and to interactively explore the neighborhood of a proposal. Focusing on a proposal in the JDevAn Viewer and exploring its relevant elements and relations enables a compact and local view of otherwise scattered model elements and relations and their changes. This localization has been helpful in helping developers to quickly explore and assess if the proposal is worth examining more closely. When the developers consider the proposal for a change API to be promising, they can then request the textual comparison results of the source code of usage-example elements that demonstrate how to adapt the application source code to properly work with the changed API.

5.3 Evaluation

In this section, we discuss our evaluation of the effectiveness of our approach for catching up the API evolution of a component framework and supporting the migration of client applications that reuse it. We conducted two case studies with two subject systems with the Diff-CatchUp prototype: HTMLUnit [123] and JFreeChart [126]. The subject systems both have been developed for several years with multiple major releases.

In addition to the core framework/library APIs, each release of the subject system also includes the corresponding regression tests (e.g., JUnit test suites) and some classes that demonstrate the typical usage scenarios of the framework/library. The subject systems have undergone a substantial number of API changes (see Appendix C and Appendix D).

With our design-evolution analysis tool JDEvAn [125], we populated the repositories for two subject systems respectively, which store the UML logical models of each major release of the subject system (including the core framework/library APIs and the accompanied demonstration and testing classes) and the API-change facts reported by pair-wise differencing subsequent releases with *UMLDiff*.

Given one of the major releases N of the subject system, we built the demonstration and testing code¹⁵ of the version N with the core framework/library APIs released in the subsequent version $N+1$, which resulted in various types of API migration problems being reported. We then collected all the distinct broken APIs (see Table 5-7 and Table 5-8) involved in the reported migration problems. For each broken API element, we ran the Diff-CatchUp tool against the repository of the subject system to generate its corresponding replacement proposals. Finally, we compared the Diff-CatchUp replacement proposals with the changes that the developers of the subject system actually made in order to adapt the demonstration and testing code of version N in response to the core framework/library API changes when the subject system evolved into version $N+1$. If the actual change was captured within the Diff-CatchUp top ten replacement proposals and Diff-CatchUp was able to recommend relevant usage examples, we considered that the Diff-CatchUp approach would have effectively helped the application developers to successfully resolve the given migration problem and consequently evolve their applications in the face of the corresponding interface changes of the subject system.

In Section 5.3.1, we qualitatively describe our use of Diff-CatchUp to support adapting application code to several API changes when JFreeChart evolved from the version 0.9.4

¹⁵ We also attempted to collect the API migration problems from the publicly available client applications that reuse HTMLUnit or JFreeChart. However, we failed to achieve that goal since we cannot find such an application that has been well evolved in sync with the evolution of the subject systems.

to 0.9.5. In Section 5.3.2, we discuss the runtime performance of Diff-CatchUp and quantitatively evaluate the effectiveness of our approach in terms of types and numbers of distinct broken APIs we encountered in our case studies and the statistics of successful and failing proposals Diff-CatchUp generates.

5.3.1 A usage scenario of the Diff-CatchUp tool

First, let us discuss in detail how Diff-CatchUp helps resolving the migration problems discussed in the motivation example of Section 5.1, through which we illustrate the typical usage scenarios of the tool. In the version 0.9.4 of JFreeChart, there is a demonstration class `BaselImageServlet`. When building the `BaselImageServlet` class of the version 0.9.4 with the core library APIs of the version 0.9.5, 43 compilation errors and 2 warnings are reported, as shown in the Eclipse's Problems view in Figure 5-1. One of three "The import `com.jrefinery.chart.data` cannot be resolved" problems is selected. A request for "*Catchup API Evolution*" from the context menu of the Problems view invokes a Diff-CatchUp search on the JDEvAn repository of JFreeChart, which reports that the problematic package `com.jrefinery.chart.data`, which existed up until version 0.9.4, was removed in version 0.9.5.

Diff-CatchUp next automatically identifies the specific type involved in the given problematic import declaration, which is the class `PlotFit` for the selected "import not found" problem and attempts to catch-up the evolution of the `PlotFit` class. Similarly, for the other two "The import `com.jrefinery.chart.data` cannot be resolved" problems, it identifies the involved classes to be `LinearPlotFitAlgorithm` and `MovingAveragePlotFitAlgorithm` respectively. Table 5-4 summarizes the changes made to these three broken classes, as reported by *UMLDiff*, and the corresponding replacement proposals generated by Diff-CatchUp.

In version 0.9.4, the `PlotFit`, `LinearPlotFitAlgorithm`, and `MovingAveragePlotFitAlgorithm` classes work together to produce a `XYDataset` object. In the subsequent version 0.9.5, these three classes are all removed. However, the Diff-CatchUp replacement proposals indicate that the relevant feature does not actually disappear with the removal of these three classes. Instead, the roles of the classes `PlotFit`, `LinearPlotFitAlgorithm`, and `MovingAveragePlotFitAlgorithm` appear to be replaced by the newly added class

MovingAverage and three matched classes LineFunction2D, DatasetUtilities, and Regression. These four replacing classes are all contained in a matched package com.jrefinery.data. They are all newly used by the classes that stopped using the three removed PlotFit-related classes. For the classes PlotFit and MovingAveragePlotFitAlgorithm, the recommended replacing classes ranked at the 1st, 6th, and 9th – 17th places are omitted in Table 5-4. Those classes are contained in com.jrefinery.chart.plot and com.jrefinery.chart.renderer packages. Package is one way of grouping together related classes according to their functionalities. Thus, the developer using the Diff-CatchUp tool may conjecture that such plot- or renderer-related classes should be irrelevant to the PlotFit-related feature or he may examine them after those classes listed in Table 5-4.

Table 5-4. Catch-up the evolution of the three PlotFit-related classes

| Broken classes | Replacement proposals | | | |
|---|-------------------------|---------|------------|---------|
| | Replacing classes | Status | Heuristics | Support |
| PlotFit (Removed) | 2. MovingAverage | added | newly used | 0.250 |
| | 3. KeyedValues2D | added | newly used | 0.125 |
| | 4. TimeSeriesCollection | matched | newly used | 0.625 |
| | 5. LineFunction2D | matched | newly used | 0.250 |
| | ... | ... | ... | ... |
| | 7. DatasetUtilities | matched | newly used | 0.125 |
| | 8. Regression | matched | newly used | 0.125 |
| | ... | ... | ... | ... |
| MovingAveragePlotFitAlgorithm (Removed) | 2. MovingAverage | added | newly used | 0.250 |
| | 3. KeyedValues2D | added | newly used | 0.125 |
| | 4. TimeSeriesCollection | matched | newly used | 0.625 |
| | 5. LineFunction2D | matched | newly used | 0.250 |
| | ... | ... | ... | ... |
| | 7. DatasetUtilities | matched | newly used | 0.125 |
| | 8. Regression | matched | newly used | 0.125 |
| | ... | ... | ... | ... |
| LinearPlotFitAlgorithm (Removed) | 1. MovingAverage | added | newly used | 0.500 |
| | 2. LineFunction2D | matched | newly used | 1.000 |
| | 3. DatasetUtilities | matched | newly used | 0.500 |
| | 4. Regression | matched | newly used | 0.500 |

However, knowing that the PlotFit-related feature may be replaced by the classes MovingAverage, LineFunction2D, DatasetUtilities and Regression is not nearly enough to successfully migrate the client BaselmageServlet class to use the replacements to deliver the same or similar functionalities previously implemented by the classes PlotFit,

LinearPlotFitAlgorithm, and MovingAveragePlotFitAlgorithm. There are 27 methods available across these four replacing classes. It is not simple to determine which ones to use and find the correct sequence of interaction between them. Furthermore, for the removed classes PlotFit and MovingAveragePlotFitAlgorithm, two more classes in the com.jrefinery.data package, KeyedValues2D and TimeSeriesCollection, are also recommended by Diff-CatchUp. Are they relevant to replacing PlotFit-related classes? If so, what roles are they supposed to play?

Table 5-5. Replacement proposals for PlotFit.getFit() with full heuristics checking

| Replacing APIs | Status | Heuristics | Support |
|--|---------|------------|---------|
| 1. <i>MovingAverage.createMovingAverage(XYDataset, ...)</i> | added | data type | 0.50 |
| 2. TimeSeriesCollection() | matched | data type | 0.67 |
| 3. DemoDatasetFactory.createHighLowDataset() | renamed | data type | 0.50 |
| 4. <i>DatasetUtilities.sampleFunction2D(Function2D, ...)</i> | matched | data type | 0.17 |
| 5. JDBCXYDataset(...) | renamed | data type | 0.17 |

Table 5-6. Replacement proposals for PlotFit.getFit() with only “newly used” heuristic

| Replacing APIs | Status | Heuristics | Support |
|---|---------|------------|---------|
| 3. <i>MovingAverage.createMovingAverage(TimeSeries, ...)</i> | added | newly used | 0.67 |
| 4. <i>MovingAverage.createMovingAverage(XYDataset, ...)</i> | added | newly used | 0.50 |
| 5. TimeSeriesCollection() | matched | newly used | 0.67 |
| 6. TimeSeriesCollection.addSeries(TimeSeries) | matched | newly used | 0.67 |
| 8. DemoDatasetFactory.createJPYTimeSeries() | matched | newly used | 0.50 |
| 9. DemoDatasetFactory.createEURTimeSeries() | matched | newly used | 0.33 |
| 12. DemoDatasetFactory.createUSDTimeSeries() | matched | newly used | 0.17 |
| 13. LineFunction2D(double, double) | matched | newly used | 0.17 |
| 14. Regression.getOLSRegression(XYDataset, int) | matched | newly used | 0.17 |
| 19. <i>DatasetUtilities.sampleFunction2D(Function2D, ...)</i> | matched | newly used | 0.17 |

To answer these questions, the developer needs more fine-grained method-level information in relation to these plausible replacing classes. The key API method relevant to the old PlotFit-related feature is PlotFit.getFit(). Highlighting a call to PlotFit.getFit() in the Eclipse’s Java editor enables the developer to request Diff-CatchUp support for determining how it changed from version 0.9.4 to 0.9.5. It turns out that the method PlotFit.getFit() was removed in 0.9.5. Table 5-5 summarizes the replacement proposals that Diff-CatchUp generates when selecting replacing elements with full heuristics checking, which recommends a total of 5 methods/constructors that declare the same or compatible

data type as the removed `PlotFit.getFit()`. When Diff-CatchUp selects the replacing elements based only on “newly used” heuristic, there are a total of 23 recommended methods and constructors. 10 of them, which are declared in the replacing classes listed in Table 5-4, are summarized in Table 5-6. The other 13 axis-, plot-, and renderer-related methods and constructors are omitted in Table 5-6 since they are deemed less relevant to replacing the removed method `PlotFit.getFit()`.

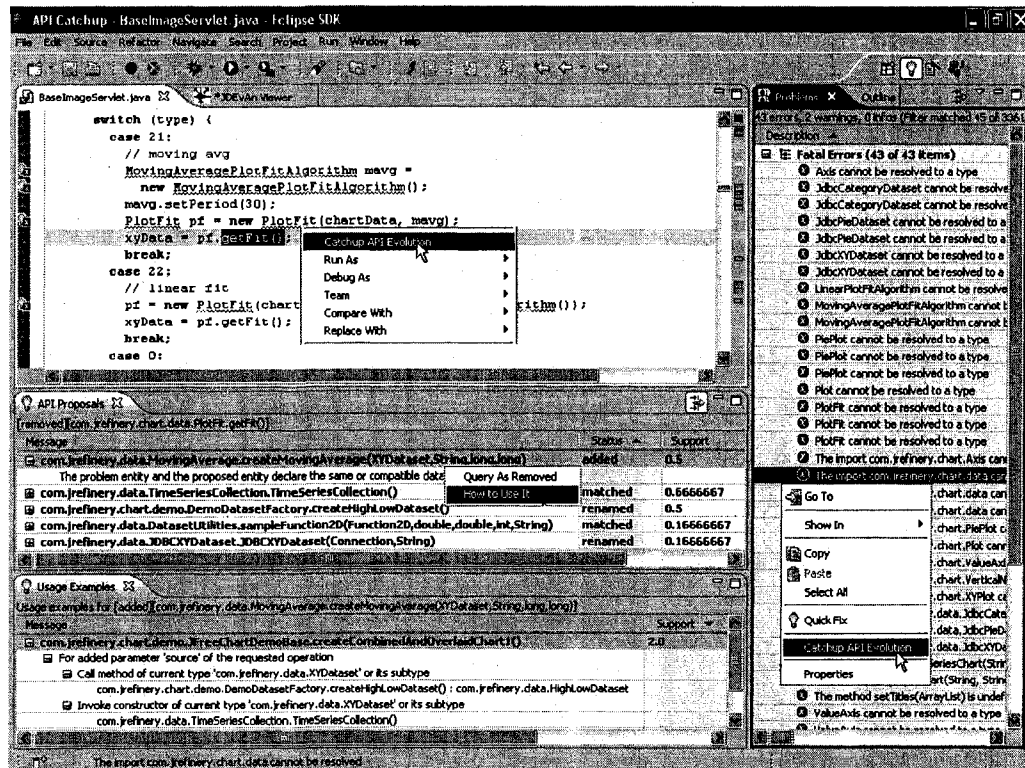


Figure 5-1. Diff-CatchUp perspective in Eclipse

The generated replacement proposals are returned to the Diff-CatchUp client for presentation to the developer in the *API Proposals* view. For a selected proposal, the developer can ask Diff-CatchUp to recommend its obtain-object, parameter-list, and/or replacement usage examples, which will be presented in the *Usage Examples* view. Figure 5-1 shows a snapshot of the Diff-CatchUp perspective in Eclipse, when the developer attempts to adapt the application code in response to the removal of the method

PlotFit.getFit(). The middle- and bottom-left parts of the perspective show the API Proposals and Usage Examples views. The API Proposals view shows the broken API under investigation and the change it underwent, which states “[removed][com.jrefinery.data.PlotFit.getFit()]”. In addition, the API Proposals view lists the five replacement proposals of the removed method PlotFit.getFit(). The highest-rank replacing method MovingAverage.createMovingAverage(XYDataset,...) is selected; its corresponding table row is expanded to show the textual description of the rationale (“compatible data type” in this case) for why this method is selected. The relevant obtain-object usage examples for this method are listed in the Usage Examples view. The first one, JFreeChartDemoBase.createCombinedAndOverlaidChart1(), is selected and expanded; its rationale indicates that it was recommend because it demonstrates two effective ways to obtain a XYDataset object, i.e., by calling method DemoDatasetFactory.createHighLowDataset() or instantiating a TimeSeriesCollection object, which can be used as the first argument to invoke the replacing method MovingAverage.createMovingAverage(XYDataset,...). Note that the method createHighLowDataset() and the constructor TimeSeriesCollection() are two of the proposed replacing APIs, which indicates that they may not be the direct replacements to PlotFit.getFit() but they should be highly relevant to properly using the replacing method, such as MovingAverage.createMovingAverage(XYDataset,...).

When a replacement or usage example proposal is selected in the API Proposals and Usage Example views, the relevant model elements, relations and their changes, which are enclosed in the proposal, are visualized in a JDEvAn Viewer. Figure 5-2 shows a screenshot of the JDEvAn Viewer and its Outline and Properties view when the developer investigates the replacing method MovingAverage.createMovingAverage(XYDataset,...) and its usage example JFreeChartDemoBase.createCombinedAndOverlaidChart1().

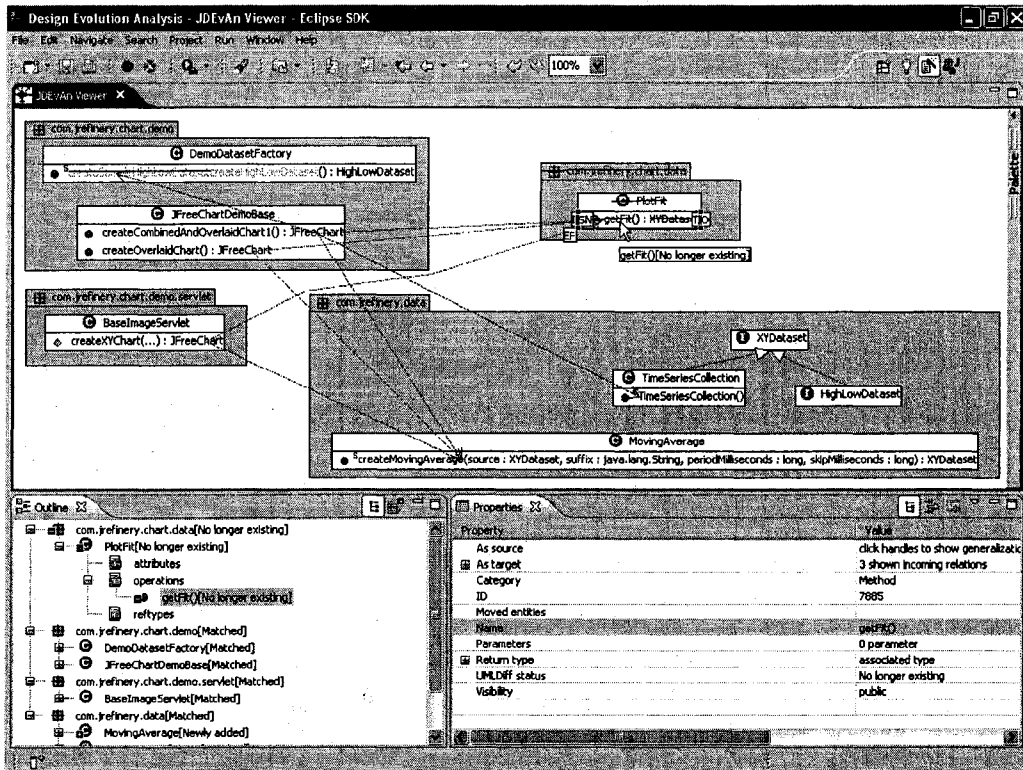


Figure 5-2. Explore PlotFit.getFit() and its replacing APIs with JDEvAn Viewer

The screenshot shows the "Compare Java Elements" dialog box. It compares two versions of the `createCombinedAndOverlaidChart()` method:

- Left side (Original):**

```

String subtitleStr = this.resources.getString("com
String domain = this.resources.getString("combined
String[] ranges = this.resources.getStringArray("c

HighLowDataset highLowDataset =
    DemoDatasetFactory.createHighLowDataset();
XYDataset highLowDatasetHA =
    MovingAverage.createMovingAverage(highLowDataset,
        5 * 24 * 60 * 60 * 1000L, 5 * 24 *

TimeSeries jpy = DemoDatasetFactory.createJPYTimeS

TimeSeriesCollection dataset0 = new TimeSeriesColl
dataset0.addSeries(jpy);

TimeSeries mav = MovingAverage.createMovingAverage
TimeSeriesCollection dataset1 = new TimeSeriesColl
dataset1.addSeries(jpy);
dataset1.addSeries(mav);

```
- Right side (Updated):**

```

String subtitleStr = this.resources.getString(
String domain = this.resources.getString("comb
String[] ranges = this.resources.getStringArra

// calculate Moving Average of High-Low Dacese
HighLowDataset highLowData =
    DemoDatasetFactory.createSampleHighLowData
MovingAveragePlotFitAlgorithm mavg =
    new MovingAveragePlotFitAlgorithm();
mavg.setPeriod(5);
PlotFit pf = new PlotFit(highLowData, mavg);
XYDataset highLowHData = pf.getFit();

// calculate Moving Average of Time Series
XYDataset timeSeriesData = DemoDatasetFactory.
mavg = new MovingAveragePlotFitAlgorithm();
mavg.setPeriod(30);
pf = new PlotFit(timeSeriesData, mavg);
XYDataset timeSeriesHData = pf.getFit();

```

Figure 5-3. Code differences demonstrating how to replace getFit() with createMovingAverage(...)

At this point, if the developer believes that `MovingAverage.createMovingAverage(XYDataset,...)` is a promising candidate for replacing `PlotFit.getFit()`, he may want to examine how its client methods, such as `createCombinedAndOverlaidChart1()`, have been modified from using `PlotFit.getFit()` to using `MovingAverage.createMovingAverage(XYDataset,...)`. Double-clicking the method `createCombinedAndOverlaidChart1()` brings up the Eclipse Compare Dialog, which shows the textual differences between the versions 0.9.4 and 0.9.5 of this method (see Figure 5-3). The code differences clearly demonstrate how to migrate the application code that relies on the old API `PlotFit.getFit()` into using the replacing API `MovingAverage.createMovingAverage(XYDataset,...)`.

The other replacement proposals are similarly examined. Three APIs emerge as candidates for replacing the removed `PlotFit.getFit()` method (highlighted in italic font in Table 5-5 and Table 5-6): `MovingAverage.createMovingAverage(XYDataset,...)`, `MovingAverage.createMovingAverage(TimeSeries,...)`, and `DatasetUtilities.sampleFunction2D(Function2D,...)`. Furthermore, several other proposed APIs, such as `TimeSeriesCollection()`, `JDBCXYDataset()`, `Regression.getOLSRegression()`, `createHighLowDataset()`, etc., are also highly relevant. They serve the important auxiliary roles in order to replace the removed method `PlotFit.getFit()`. They are used to construct objects that are necessary to call the replacing methods or to wrap the returned objects of these methods before they are used.

5.3.2 The effectiveness of Diff-CatchUp

We have evaluated our approach for catching-up and supporting API evolution with two subject systems, HTMLUnit and JFreeChart. Appendix C and Appendix D report the numbers of model element and relation facts extracted by JDEvAn fact extractor and the API changes discovered by *UMLDiff* for JFreeChart and HTMLUnit respectively.

As shown in Appendix C, our repository contains more than 1,400,000 model facts and thousands of API-change facts¹⁶ for the JFreeChart system. However, because Diff-CatchUp repository should mainly be accessed far more often than having new model or

¹⁶ The changes to usage dependency between model elements are not included.

change facts inserted, we have the tables well indexed in the database. Therefore, Diff-CatchUp typically takes a few seconds to search the repository and form and return the replacement and usage example proposals. Our case studies indicate that Diff-CatchUp approach allows the on-line interactive catching-up of API evolution of a component framework when adapting the client applications that depend on it.

Table 5-7. The success rate of Diff-CatchUp in the evolution of JFreeChart

| Type of problem ¹⁷ | #broken API | #proposal | % |
|---|-------------|-----------|------|
| ImportNotFound | 17 | 17 | 100 |
| UndefinedType+ImportNotFound+ UndefinedName | 254 | 247 | 97.2 |
| InvalidClassInstantiation | 1 | 1 | 100 |
| UndefinedMethod/Constructor | 180 | 151 | 83.9 |
| ParameterMismatch | 54 | 53 | 98.1 |
| UndefinedField+UndefinedName | 33 | 29 | 87.9 |
| UsingDeprecatedType | 3 | 3 | 100 |
| UsingDeprecatedMethod/Constructor | 35 | 34 | 97.1 |
| Total | 577 | 535 | 92.7 |

Table 5-8. The success rate of Diff-CatchUp in the evolution of HTMLUnit

| Type of problem | #broken API | #proposal | % |
|-----------------------------------|-------------|-----------|------|
| UndefinedType | 1 | 1 | 100 |
| UndefinedMethod/Constructor | 11 | 9 | 81.8 |
| ParameterMismatch | 3 | 3 | 100 |
| UsingDeprecatedType | 1 | 0 | 0 |
| UsingDeprecatedMethod/Constructor | 10 | 7 | 70 |
| Total | 26 | 20 | 76.9 |

Let us now discuss the general effectiveness of our approach in support of catching-up and assisting API evolution in the context of reuse-based software development. Table 5-7 and Table 5-8 summarize the statistics of applying our Diff-CatchUp approach (similar to the migration process illustrated in Section 5.3.1) to JFreeChart and

¹⁷ As it is currently implemented, our Diff-Catchup tool is able to handle all types of migration problem listed in Table H-1. All of them have been tested through the mock-up test cases. However, we only collected the listed types of migration problems in the evolution history of the two subject systems.

HTMLUnit respectively. The “#broken API element” column reports the number of distinct broken API elements whose changes cause the corresponding type(s) of migration problems when building the demonstration and testing code of a previous version with the core library APIs released in the subsequent version. The “#proposal” column represents the times that our approach successfully generates the replacement proposals and the usage examples, given such a broken API element. Note that the migration problems caused by the change to the same underlying broken API element were only counted once in Table 5-7 and Table 5-8.

We collected in total 577 distinct broken API elements in the JFreeChart case study. Overall, the case studies suggest that our approach is worthwhile and effective: for about 93% of the broken API elements in the JFreeChart case study, Diff-CatchUp successfully generates the replacing API elements and the corresponding usage examples that demonstrate the migration from the old APIs to their replacing ones. The overall success rate in the HTMLUnit case study is lower (about 77%). However, we consider the statistics of the HTMLUnit case study less representative than that of the JFreeChart case study, since HTMLUnit has a much smaller set of broken API elements (only 26). Furthermore, as illustrated in Section 5.3.1, our approach does not place any constraints on the mappings between the broken API elements and their plausible replacements. It is able to handle the cases of one-to-many, many-to-one, or many-to-many mappings. In addition, the relevant auxiliary APIs to properly use the replacing APIs would most likely be proposed at the same time.

5.4 Threats to validity

There are several factors that can impact the quality of our Diff-CatchUp approach.

5.4.1 API changes without syntactic effects

Our approach starts with the API migration problems that a compiler generates when building a client application with the new version of a component framework. The migration problems are analyzed to determine the broken APIs whose evolution results in the problem. The compilation errors and warnings are essentially syntactic problems that the client-application developers have to resolve before they can build their application

successfully and retest it with the evolved component framework. The causes of some syntactic problems may be the results of behavior-preserving refactorings, while others imply the semantic changes to the broken APIs. For example, additional parameter(s) most commonly indicate new behavior; declaring a class abstract indicates that it can no longer be directly instantiated; declaring a field final indicates that it is no longer changeable. In such cases, our approach is able to help client-application developers understand the nature of the change and migrate their application accordingly.

However, not all API changes result in syntactic problems being reported. Some of them may result in the client applications behaving differently. For example, a method declares two parameters of type integer, which represent the start and end position of a sequence being processed within the method. In the new version, the method still declares two parameters of type integer. However, the second parameter changes to represent the length of a sequence starting at the given start position. If the client-application developer imports the new version of the method without making any changes to the application, the code will still compile. However, the application would most likely behave abnormally, since the interpretation of the second parameter has changed. In such cases, the client-application developer has to first determine what the potential broken API is, since it cannot be automatically determined based on compilation errors and warnings, before they can highlight the code fragment (e.g., a call to the potential broken method) and request the API evolution catch-up support. In this particular example, our Diff-CatchUp approach then would most likely report the concerned method being renamed with the “end” parameter being removed and the “length” parameter being added, which can eventually guide the modification of the application to accommodate the interface change of the method.

5.4.2 The quality of *UMLDiff* results

Our approach to catching-up API evolution of a component framework relies on the API-change facts reported by *UMLDiff* when it compares two subsequent versions of the system evolution. The renamings and moves that have been erroneously identified or missed by *UMLDiff* will negatively affect our Diff-CatchUp approach. For example, a method *M* is removed but it is erroneously identified as renamed. If the method *M* is

involved in an “undefined method” problem, M ’s renaming counterpart in the new version will be recommended as its replacing method based on the erroneous *UMLDiff* change fact, which could mislead the developer’s effort to adapt the application code to the removal of the method M . Our evaluation of *UMLDiff* has shown its precision and recall to be good in practice. Thus, its negative effects on the Diff-CatchUp’s API migration process should be minor. In addition, an interactive inspection step with the support of the JDevAn tool [125] could be injected, which has been done in our case studies, after the completion of *UMLDiff* and before starting the API migration process, to correct the erroneously identified renamings and moves and to identify potentially missed instances. Finally, our approach allows the developers to explicitly request a mapped API element to be processed as removed when they deem the mapped counterpart returned by default inappropriate.

5.4.3 Availability of “voluntary” migration examples

Our approach does not assume the existence of the special handcrafted migration examples that demonstrate how to evolve application code in response to the interface changes of a component framework. Instead, our approach relies on the fact that a component framework embodies “voluntary” migration examples in its evolution history and thus itself represents good usage of its evolving API.

We have identified four major reasons that cause the Diff-CatchUp failures, especially in the cases of the undefined problems caused by the removals of methods, constructors, and fields, i.e., the types of problems for which our Diff-CatchUp approach more frequently fails to recommend the corresponding replacing APIs.

The first reason is that our approach assumes that some user elements within the new version of the component framework have been properly migrated to the new APIs, i.e., they have stopped using the changed APIs in favor of their corresponding replacements, thus demonstrating how to migrate from the “old” APIs to their “new” replacements. Thus, if no such user element exists, Diff-CatchUp will fail to collect any candidates for replacements. For example, in some cases, the migrated user elements do not use the replacing APIs directly. Although the transitive usage dependencies between model elements are available in the underlying logical model, examining all the transitively used

elements is time-consuming and generally produces too much noise. Thus, Diff-CatchUp does not collect replacement candidates transitively along usage dependencies and may consequently fail to identify a valid demonstration of a replacement. Furthermore, the API of a component framework may sometimes change dramatically, including removal of some of its elements and changes to all their relevant elements. In such cases, the replacing features, including the replacing APIs and their corresponding user elements, are “completely” new. Consequently, our approach cannot locate any user elements that are related to the removed APIs and their replacing APIs at the same time, which results in its failure to generate any proposals.

The heuristics of our Diff-CatchUp approach for selecting most plausible replacing elements from the potentially large set of candidate elements may also prevent it from identifying valid replacements. On one hand, they are effective on filtering out irrelevant elements and generating a short and manageable list of replacement proposals returned to the developer for further inspection. However, the chances are that there exist no valid heuristics between the removed APIs and their replacements. Consequently, no candidate would be selected as plausible replacing element. For the deprecated methods and constructors, they generally have the usage dependency or even extract/inline operation relationships with their replacements, which makes the rate of successfully generating their replacement proposals much higher than that of the undefined methods and constructors.

Another potential cause for Diff-CatchUp failure to recommend replacements is the fact that user classes and methods that implement complex functionalities sometimes become incohesive. They often end up with a multitude of members, many of them used in multiple different contexts. When all or most of these members are modified, they will blur the most relevant changes to the concerned broken APIs, which makes it difficult to select the plausible replacements or rank them higher in the returned list of proposals.

Finally, some APIs are simply removed with no replacements at all. In JFreeChart case study, 2 removed classes, 7 removed methods and constructors, and 2 removed fields fall into this category. Our approach may still produce some proposals for them. However, upon the close inspection through JDevAn Viewer, a developer can generally determine,

without spending too much effort, that the recommended elements are irrelevant and the broken APIs disappear without replacements.

As discussed above, the existence of the user elements and the amount of changes they undergo affect the Diff-CatchUp's ability to collect potential candidate elements, select plausible replacement proposals, and determine the relevancy of the proposals. However, our experiments with the JFreeChart case study indicate that our assumption holds for most cases (about 93% overall) and our approach is worthwhile and effective on generating the replacing API elements and the corresponding usage examples in the face of the API evolution of a component framework.

5.5 Summary

In this chapter, we discussed our Diff-CatchUp approach to supporting the migration of client applications to appropriately use the evolved APIs of their underlying component frameworks. Diff-CatchUp does not require the component-framework developers to change their development practices and does not constrain the development environments adopted by the component and client-application developers. Instead, Diff-CatchUp recognizes the API changes that the reused framework has undergone – as automatically produced by the *UMLDiff* algorithm – and uses a set of heuristics to infer plausible replacements for the offending API that causes the API migration problem and examines the code base built on the evolved framework to select examples of how the potential replacements are used.

The Diff-CatchUp approach relies on the fact that a component framework itself represents good usage of its evolving API and thus is sensitive to the existence of “voluntary” migration example in its evolution history and the amount of changes they undergo. However, our evaluation indicates that its assumption holds for most cases and it is quite effective (over 93% success rate in JFreeChart case study) generating the replacing API elements and the corresponding usage examples in the face of the API evolution of a component framework.

Chapter 6: Longitudinal Design-Evolution Analysis

In addition to recognizing specific instances of design-change patterns, such as refactorings, some development tasks require that the developer understands the evolution trajectory, through which the system design has reached its current state. For example, when faced with the “shotgun surgery” smell [32], which may be corrected with the *form template method* refactoring [32], the developer has to understand how the classes affected by the “shotgun surgery” have been changing in the past, to decide what the skeleton, the default and the variant methods of the template-method class should be. Other tasks imply a need to understand the practices that the system-development team has adopted in the past. For example, when faced with a complex, non-cohesive class delivering multiple features, the developer has to decide how many of its features to extract in a new class. This judgment may depend, to some extent, on the specific ranges of class-complexity metrics that the developing organization has adopted as acceptable. A developer, who may be new to the team and may not have experience with the issue, could make a “right” decision by appealing to the way similar situations have been resolved in the past evolution history of the project. In addition to developers, managers are also interested in having an up-to-date mental model of the evolution trajectory of their projects, in order to abstract evolution patterns characteristic of desired qualities or symptomatic of problems so that they can better manage the project and predict the evolution of other projects in the future.

There already exists a substantial body of literature on the general “software-evolution understanding” topic. A large subset of work [7,30,31,33,41,58,87] in this area involves analyzing the “history” recorded by version-management systems. Unfortunately, more frequently than not, such “history” data is sparse and inconsistent [13]. Another line of research [9,29,35,57,110] has focused on the visualization of software-process statistics, source code metrics, static dependence graphs, CVS-like deltas and their derivatives, etc. However, such visualizations do not scale well to handle large systems and require a substantial interpretation effort on behalf of their users.

In this chapter, we discuss a suite of automated software-evolution analyses, based on the design changes reported by *UMLDiff*, to study the longitudinal evolution of object-

oriented systems. First, the quantitative report of *UMLDiff* changes is discretized to produce a qualitative record of the volatility of the design evolution of each individual class, i.e., discrete class-evolution profile¹⁸, throughout the system's history. Then, four types of longitudinal analyses – phasic analysis [84], gamma analysis [75], optimal matching analysis [1], and association rule mining [2] – are applied to the discrete class-evolution profiles to recover a high-level abstraction of distinct evolution phases and their corresponding styles and to identify class clusters with similar evolution trajectories.

These longitudinal design-evolution analyses aim at addressing the following general research questions:

- Are there distinct patterns in the evolution trajectories of classes in object-oriented software systems?
- What types of inter-dependencies are there among the evolution trajectories of different classes and what might their implications be for the system's subsequent development?

6.1 Classifying evolution behavior

For an evolving software system with N successive versions, *UMLDiff* can be applied N times to generate the differences between the $(I+1)^{th}$ and I^{th} versions, where $0 \leq I < N$ (supposing there is a virtual version 0 with no entities), resulting in an audit trail of the design changes that the system classes have suffered throughout their evolutionary lifecycle. This trail is analyzed to produce a *class-evolution profile* of length N (at most) for each individual system class (including interface). The class-evolution profile reports quantitatively the complete history of changes made to an individual class in each subsequent system version, i.e., in which version it was created, how many signature-changes it underwent and how many of its member elements were newly added, removed,

¹⁸ Similarly, subsystem- or system-evolution profiles can be produced. There is no fundamental difference between them, except for the different granularity of analysis. The classifications of change activities and the subsequent longitudinal analyses discussed in this chapter are applicable to all levels of evolution profiles.

moved, signature-changed¹⁹ in a particular version over its lifespan, and, possible, in which version it was deleted. Table 6-1 shows the evolution profile of class CategoryPlot from the JFreeChart case study. Note that this class was originally named BarPlot. *UMLDiff* correctly identified the renaming to CategoryPlot at version 0.8.0.

Table 6-1. The evolution profile of class CategoryPlot

| | Version | Add | Remove | SigChange | Move | Volatility |
|--------------|---------|-----|--------|-----------|------|--------------------|
| BarPlot | 0.5.6 | 19 | 0 | 0 | 0 | Rapidly developing |
| BarPlot | 0.6.0 | 11 | 3 | 19 | 0 | Intense evolution |
| BarPlot | 0.7.2 | 3 | 0 | 3 | 0 | Steady-state |
| BarPlot | 0.7.3 | 2 | 0 | 1 | 0 | Steady-state |
| BarPlot | 0.7.4 | 0 | 0 | 1 | 0 | Steady-state |
| CategoryPlot | 0.8.0 | 2 | 0 | 6 | 6 | Restructuring |
| CategoryPlot | 0.9.0 | 30 | 6 | 7 | 14 | Intense evolution |
| CategoryPlot | 0.9.1 | 2 | 0 | 1 | 0 | Steady-state |
| CategoryPlot | 0.9.3 | 1 | 0 | 5 | 0 | Restructuring |
| CategoryPlot | 0.9.4 | 3 | 0 | 1 | 0 | Steady-state |
| CategoryPlot | 0.9.5 | 52 | 18 | 25 | 4 | Intense evolution |
| CategoryPlot | 0.9.7 | 5 | 0 | 8 | 0 | Restructuring |
| CategoryPlot | 0.9.9 | 35 | 20 | 11 | 10 | Intense evolution |
| CategoryPlot | 0.9.10 | 42 | 1 | 14 | 11 | Intense evolution |
| CategoryPlot | 0.9.11 | 2 | 0 | 0 | 0 | Steady-state |
| CategoryPlot | 0.9.12 | 4 | 0 | 8 | 0 | Restructuring |
| CategoryPlot | 0.9.13 | 8 | 0 | 9 | 0 | Restructuring |
| CategoryPlot | 0.9.14 | 8 | 5 | 4 | 0 | Slowly developing |
| CategoryPlot | 0.9.17 | 1 | 0 | 3 | 0 | Steady-state |
| CategoryPlot | 0.9.19 | 7 | 14 | 40 | 0 | Intense evolution |
| CategoryPlot | 1.0.0 | 4 | 2 | 4 | 0 | Slowly developing |

As can be seen from Table 6-1, between any two versions of the system evolution, the types and amounts of changes may vary greatly. This level of detail and its variability make it very difficult to discern interesting patterns in the evolution of system classes. This is why the quantitative class-evolution profiles are discretized.

¹⁹ Signature-changes include renamings and other modifications of signature, such as visibility and modifier changes of class, interface, field, and method and constructor, changes of data and return type and changes of inheritance relation.

The discretization process is based on two descriptive statistics: quantiles and means. The quantile is the specific value of a variable that divides the distribution into two parts, those values greater than the quantile value and those values that are less. That is, p percent of the values are less than the p % quantile. The a % and b % quantiles ($a < b$) for “Addition” and “Removal” are calculated. A continuous value is discretized as “Low” if it is below a % quantile, “High” if it is above b % quantile, and “Medium” otherwise.

Since the majority of class-version (at least according to our experience) have no “Added” members (see Section 6.4.1), those class-versions with zero “Addition” are ignored when computing the “Addition” quantiles. Furthermore, classes with only one “Addition” account for about 25%-40% of the rest. To alleviate this substantial imbalance in the input data, these class-versions are ignored as well. For the remaining class-versions that have two or more “Addition” changes, the 25% (75%) “Addition” quantiles are computed, and rounded down (up) to get the corresponding integer low a % (high b %) “Addition” quantiles, which generally fall into the range of 15%-25% quantile or 75%-85% quantile respectively. “Removal” is treated similarly to “Addition”.

Because moves and signature-changes appear less frequently than additions and removals, the range of values for moves and signature-changes is narrower than the range of values for the latter two types of modifications. Therefore, the means of the numbers of “Movement” and “Signature-change” are computed and they are discretized into two ranges: “High” if they are above the mean and “Low” otherwise. Again, the class-versions with zero or one “Movement” or “Signature-change” are ignored. Furthermore, if a class is moved or renamed in a particular version, its corresponding qualitative value of “Movement” or “Signature-change” will be marked as “High”, irrespective of changes made to its member elements.

In general, there are four characteristic types of evolutionary behavior that software system and its components may exhibit at the design level. First, the design may simply not evolve – *steady state* - during periods of bug fixes, testing and documentation. Alternatively, the design may go through *restructuring*, when design entities are neither added nor removed but simply moved or slightly modified. During periods of *functionality extensions* in response to behavioral-change requests the design is bound to change with new entities introduced to the system to support the new features and

existing entities removed to eliminate the unwanted behaviors. Finally, the design may go through phases of *intense evolution* including both restructuring activities in anticipation of new features as well as changes in support of adaptations to the system behavior.

Table 6-2. Classifications of change activities

| Type | Combination of labels | Description |
|------|---|--------------------|
| A | (a _H or r _H) & (m _H or s _H) | Intense evolution |
| B | (a _H or r _H) & (m _L and s _L) | Rapidly developing |
| C | !(a _H or r _H) & (m _H or s _H) | Restructuring |
| D | (a _M or r _M) & (m _L and s _L) | Slowly developing |
| E | (a _L and r _L) & (m _L and s _L) | Steady-state |

These types of high-level evolution behavior can be recognized in terms of the qualitative values characterizing the amount of changes of each type in the class-evolution profiles. Table 6-2 lists five distinct classifications of evolution behavior, distinguishing between five levels of intensity of system classes' evolution. The characters *a*, *r*, *m*, and *s* represent addition, removal, movement, and signature-change respectively. The subscripts *H*, *M*, and *L* represent High, Medium, and Low respectively.

According to Table 6-2, a period, defined by a "before" version and an "after" version of an evolving software system, is considered to exhibit *intense evolution* (type-A) when a High number of entities are newly created and/or removed in the "after" version, and there is also a High number of moved design entities and/or signature-changes. The high number of moves and signature-changes implies that many perfective maintenance activities are happening during this period. At the same time, the high number of newly created and/or removed entities is very likely the result of such maintenance activities, such as adding new fields for the restructured class interface, removing deprecated methods, etc. It might also be the result of functionality extension.

Type-C periods correspond to *restructuring* periods. They contain many maintenance activities that result in a High number of moves and signature-changes, while their numbers of newly created and/or removed entities remain Medium or Low.

Type-B and type-D periods correspond to *functionality-extension* periods with different levels of intensity. Neither includes many perfective maintenance changes. Type-B periods contain a High number of newly created and/or removed entities. The

software system develops rapidly during such periods. Type-D periods, on the other hand, contain a Medium number of addition and/or removal changes; the system is slowly developing during such periods.

Finally, type-E periods represent the *steady-state* periods of system evolution, where the number of maintenance changes remains Low and the number of other types of changes is also Low. The system changes little during this time.

These five classifications characterize the volatility of system classes over their lifecycle. The discrete evolution profile (composed of a sequence of discrete volatility types) of the CategoryPlot class is shown in the last column of Table 6-1. Compared with the continuous value profile of CategoryPlot, the discrete view of its profile exhibits clearly the phases of the CategoryPlot's evolution, which can then be easily analyzed to discern its sequential evolutionary patterns.

The evolution profile of the CategoryPlot class started with a rapidly developing phase in which it was introduced into the system with 19 members. It then was under intense development until version 0.6.0, which was followed by five versions with few (or no) changes (note that version 0.7.1 is not in the table, since there is no change made to class CategoryPlot in that version). This steady-state phase ended at version 0.8.0, in which the class interface was refactored, and was then followed by an intense evolution phase until version 0.9.0. After that, the class underwent minor changes until version 0.9.5. From 0.9.5, the class entered its most active period over its lifespan. This continuous intense development resulted in the two restructuring phases in version 0.9.12 and 0.9.13. CategoryPlot then went into a slowly developing phase, followed by a steady-state phase, followed by an intense evolution phase again in version 0.9.19, and finally went stable.

6.2 Analyzing class-evolution phases and styles

After the class-evolution profiles have been discretized, a nonparametric sequential analysis (phasic analysis) [84] is applied to generate a flexible phase map that is composed of the sequence of discrete change activities. Further gamma analysis [75] of the class phase maps identifies general ordering relations among the various types of phases. Finally, optimal matching analysis [1] is applied to discover clusters of systems classes with similar evolution styles.

6.2.1 Phasic analysis

Phasic analysis recognizes distinct phases in the discretized evolution profile of a design entity, whether it is the system as a whole or an individual class. Intuitively, a phase consists of a consecutive sequence of system versions, all of which exhibit similar classifications of change activities. Identifying a phase in a class-evolution profile may provide some insight regarding the development goals during the corresponding period.

Phasic analysis has been widely applied in longitudinal analysis of temporal processes of social phenomena. It works with nominal and categorical data and has been successfully applied in the study of information-system development [84]. It assumes that social behaviors can be described in units larger than individual acts, which can cohere into phases or patterns in the developmental path of a social event. These patterns are the result of dynamics that drive the changes over time [43].

We think of the evolution of software system in analogy to a social event that unfolds and changes over time, according to some form of underlying process. Systems evolve to fix defects, meet customer-driven functionality enhancements, adapt to changes in the deployment environment, and so forth. Thus, we believe that the phasic analysis is a promising way to understand the general evolution of software systems and their components and to discover how they are born, developed, and terminated, and the processes that drive their unfolding.

We use *WinPhaser* to apply phasic analysis on the discrete evolution profiles. *WinPhaser* is a set of sequence description and analysis tools [43] for the study of sequential data consisting of a time-ordered set of discrete elements. *WinPhaser* generates a flexible phase map from a data sequence consisting of discrete data elements. A phase map is composed of a series of coherent units. *WinPhaser*'s flexible phase-mapping module parses the data sequence into phases of different length based on shifts in the data sequence. It labels the phase with the type of predominant elements in that phase and identifies noisy periods with no predominant elements as *pending* phases.

Therefore, in our use of *WinPhaser*, there are six different types of phases that can be identified in the evolution profiles. They are: *Intense evolution*, *Rapidly developing*, *Restructuring*, *Slowly developing*, *Steady-state* and *Pending*.

6.2.2 Gamma analysis

Gamma analysis [75] recognizes recurring patterns in the relative order of phases in an evolution profile, such as consistent precedence of a phase type over another. Different process models advocate distinctive ordering of activities in the project lifecycle; gamma analysis can reveal such consistent relative orderings and thus hint at the adopted process model.

In particular, Gamma analysis provides a measure of the general order of elements in a sequence and a measure of the distinctiveness or overlap of element types. It measures the proportion of *A* events that precede or follow *B* events in a sequence. A pair-wise gamma is given by $P-Q/P+Q$, where *P* is the count of *A* events preceding *B* events and *Q* is the count of *B* events preceding *A* events. Gamma analysis of a sequence yields a table (see Table 6-5) consisting of pair-wise gamma scores for each possible pair of element types.

Based on the pair-wise gamma analysis, the precedence and separation scores (see Table 6-6) are calculated for an element type. The precedence score is given by the mean of its pair-wise gamma scores. The precedence score indicates the location of the element in the overall ordering of element types and can range from -1 to 1. A score of 1 occurs in the beginning of a sequence. A score of -1 occurs at the end of the sequence. The separation score for an element type is given by the mean of the absolute value of its pair-wise gamma scores. It is a measure of the relative distinctiveness of the element type and can range from 0 to 1. Separation approaches 1 as a larger proportion of the units of a given element type occurs contiguously. An element will obtain a lower separation score if it occurs at several widely separated points in a sequence.

Gamma mapping is the final step in gamma analysis. Precedence and separation scores are used to construct gamma maps. Element types are ordered sequentially on the basis of precedence scores (from largest to smallest). Phases with separation scores below .50 are not clearly separated from other phases.

Gamma analysis constructs an abstract sequential pattern from a phase map. The resulting gamma map is simple enough to be used to qualitatively evaluate the overall class evolution against its underlying development process (see Section 6.4.2) or

qualitatively cluster a number of system classes according to their evolution patterns at this higher-level of abstraction (see Section 6.4.4).

6.2.3 Optimal matching analysis

Optimal matching analysis [1] provides a generic tool for sequence comparison when each sequence is represented by well-defined elements drawn from an alphabet of a relatively small set of (repeating) element types. It can be used to recognize how similar the evolution profiles of two (or more) classes are.

Optimal matching analysis produces an index of the relative "distance" between any two sequences. This index is the smallest possible cost of operations of insertion, substitution and deletion of sequence elements required to align two sequences, that is, to transform one sequence into the other. The more similar the sequences being compared, the fewer operations required to align them and the smaller the index of distance or dissimilarity.

Each type of operations should be assigned a cost that represents the difficulty of making that change [6]. The cost can be thought of as the perceived unlikelihood of the change having arisen at random in whatever process produced the changed element. For example, as we discussed in Section 6.1, the steady-state phases of system evolution represent the time period in which few changes have been made, while rapidly developing phases represent the major function extension and restructuring phases represent the system maintenance activities. Therefore, in the cost model for the alignment of two phase maps of system evolution, the cost to substitute a restructuring phase with a rapidly developing one should be much higher than that to insert a steady-state phase, since rapidly developing and restructuring are very different type of phases by their very nature, but phase type of steady-state is almost equal to nothing. Furthermore, there are many possible sets of operations to align any two sequences. The dynamic programming algorithm [98] is applied to calculate the least possible cost for the alignment.

A single distance index provides only a numerical value for the dissimilarity between two sequences. In and of itself, this index may not be especially interesting. But typically a larger set of sequences are compared and the resulting distance matrix can then be

subjected to cluster analysis in order to generate a sequence typology, such as different types of class-evolution styles discussed in Section 6.4.3.

6.3 Detecting co-evolving classes

The class-evolution phases and styles reveal the evolution characteristics of each individual class throughout the lifecycle of the system. As software systems evolve over a long time, non-trivial and often unintended relationships among system components arise, which are frequently undocumented and usually are not easily perceivable in the source code. A particularly interesting such relationship is class co-evolution: because of implicit design dependencies sets of classes change in “parallel” ways and recognizing such co-evolution is crucial in effectively extending and maintaining the system. First, the system maintainers may decide to restructure the system in order to eliminate this interdependence, thus evolving it into a more modular and less coupled design. Alternatively, they may document the interdependence as a predictor of maintenance activities, so that, when some of the co-evolving classes have to be modified, the rest of the cluster is also examined and retested.

Given a set of discrete class-evolution profiles, a transaction database is populated, which is subsequently data-mined using the Apriori algorithm for association-rule mining [2] to elicit class co-evolution rules among two or more classes. In contrast to Boolean association rules [9,33,87,110], which concern associations between the change and no-change of system classes, our class co-evolution rules are fine-grained, which concern associations between different types of changes, such as the following: “when adding members to class A, class B also gets additional members too”; or “when adding members to class A, it is often needed to restructure class B’s interface”.

For a software system with N versions, a database with N entries is generated from the discrete class-evolution profiles. Each entry T corresponds to a system version and is assigned a unique identifier, the version *ID* (*VID*); it contains a set of classes that changed in that version. The modifications of classes in each entry are represented in terms of their volatility classification (as defined in Table 6-2) in the corresponding system version: *Intense evolution*, *Rapidly developing*, *Restructuring*, *Slowly developing*, and *Steady-state*.

We programmatically use the implementation of Apriori association-rule mining algorithm in the Weka [130] toolkit. Our initial intention was to integrate the Weka Apriori implementation within the JDEvAn tool. But this implementation is in-memory and does not scale well for medium or large software systems, such as the ones in our case studies that involve several dozens of system versions, with thousands of classes, with each class-version belonging in one of five types of volatility. This data set can easily use up the Java virtual-machine memory.

To address this scalability issue, we used a data-reduction technique [40] by abstracting our original categorical volatilities to a higher conceptual level. More specifically, based on the nature of the different types of discrete change activities, when constructing the transaction database for class co-evolution analysis, we consider the rapidly and slowly developing (Type-B and Type-D) periods as instances of a general function-extension category; at the same time, we consider intense-evolution and restructuring (Type-A and Type-C) periods as instances of a refactoring category. Since a steady-going period indicates that there are few changes, it is ignored (considered as no-change) when building the class co-evolution database for mining co-evolution patterns. Therefore, the volatilities of classes in each database entry fall into three categories: function-extension, refactoring, and no-change.

The reduced representation of the data resulting from this classification-abstraction step is smaller in volume while still reflecting the nature of the original data. It reduces the memory cost of the Weka's implementation of Apriori association rule mining without substantially compromising the effectiveness of the analysis. Nevertheless, the risk of running out of memory still exists for large software systems or at low mining thresholds. Furthermore, the co-evolution rules that hold at higher conceptual level may not hold at lower levels of abstraction. Finally, a post-processing may be necessary to infer more details about the discovered rules. To address these issues, OLAP (On-line Analytical Processing) [94] may be used, to enable mining multi-level class co-evolution rules on large software system.

We briefly discuss the Apriori association rule-mining algorithm here. Readers are referred to the original paper [2] for details. Given a transaction database, the Apriori algorithm generates all association rules with at least some user-specified *minimum*

support and *confidence*. The algorithm involves two sub-problems. First, it generates all sets of items (itemsets) that have transaction support above minimum support. The *support* for an itemset is the number of entries that contain the itemset. Itemsets with minimum support are called *large itemsets* and all others are *small itemsets*. Next, the large itemsets are used to generate the desired rules. The general idea is that, if ABCD and AB are large itemsets, then the rule $AB \Rightarrow CD$ holds if its *confidence*, i.e., the ratio $\text{support}(ABCD)/\text{support}(AB)$ is greater than the user-specified minimum confidence. Note that the rule will surely have minimum support because ABCD is large.

The Weka toolkit also supports a significance test on the generated confidence-based rules. The confidence-based rules, so-called strong rules, may not be interesting to the user, since the antecedent and consequent may be negatively associated, which means that the occurrence of one of them may decrease the likelihood of the occurrence of the other. The *lift* [40] metric can be used to measure the statistical dependence (correlation) between the occurrences of itemsets. If the lift value of significant test on a strong rule is less than one, then the occurrence of the antecedent of the strong rule is negatively correlated with (or discourages) the occurrence of the consequent. If the lift value is greater than one, then it means the occurrence of the antecedent implies the occurrence of the consequent. If the lift value is equal to one, then the antecedent and consequent are independent and there is no correlation between them.

Because the transaction database for class co-evolution analysis is built on the design changes reported by *UMLDiff*, which are classified into the function-extension or refactoring categories, mining this database with Apriori produces the following types of class co-evolution rules:

- class function-extension(s) \Rightarrow class function-extension(s)
- class function-extension(s) \Rightarrow class refactoring(s)
- class refactoring(s) \Rightarrow class function-extension(s)
- class refactoring(s) \Rightarrow class refactoring(s)

Table 6-3. Transaction database²⁰ for association-rule mining

| | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 |
|-----|----|----|----|----|----|----|----|----|----|
| V01 | D | D | D | D | | D | | D | |
| V02 | | | | | | | | | |
| V03 | | | | | | | | | |
| V04 | | | | | | | | | |
| V05 | | | | | | | | | |
| V06 | | | | | | | | | |
| V07 | T | T | T | T | D | T | D | T | |
| V08 | | D | | | T | | | | D |
| V09 | | | | | | | | | |
| V10 | | | | | T | | | | |
| V11 | | | D | | | | | | |
| V12 | D | | | | | D | | | |
| V13 | T | T | T | T | T | T | T | T | T |
| V14 | | | | | | | | | |
| V15 | | T | T | T | | | T | T | T |
| V16 | | | | | | | | | |
| V17 | D | T | T | T | T | D | D | D | T |

Let us consider a system with 17 versions, whose final version contains 9 classes, C1 through C9. Table 6-3 shows the transaction database for this example system in the form of a pivot table. For this transactional data, if the minimum support is set to 20% and the minimum confidence to 50%, the following co-evolution rules are discovered (with lift value of significant test greater than one).

- C1=D (3) => C6=D (3) [confidence=1]
- C6=D (3) => C1=D (3) [confidence=1]

The number in bracket following the antecedent and consequent of the rule is the support value of large itemset. These two rules state that, in three different versions, both classes C1 and C6 were similarly modified, with new feature additions and/or old feature removals. This is essentially evidence that the two classes have parallel function-

²⁰ “D” represents function-extension, “T” represents refactoring, and empty cell represents no-change.

extension phases throughout their lifecycle. Additional concrete examples are discussed in Section 6.4.4.

6.4 Evaluation

In this section, we used JFreeChart [126] as the subject system to evaluate our longitudinal design-evolution analysis methods. More specifically, this case study was designed to examine the following specific research questions, refining the general research questions, stated in the beginning of this chapter:

- How are classes introduced to, maintained, and eliminated from the system?
- Are there any consistent order relations among the various class-evolution phases? Do they appear periodically?
- Are there distinct styles of multiple class-evolution trajectories?
- Are there any interesting inter-dependencies among the evolution trajectories that individual classes follow?

6.4.1 Class-evolution phases

First, we review the characteristics of the evolution phases of individual system classes. In particular, we are interested in “how are classes introduced to, maintained and eliminated from the system?” Over the life span of JFreeChart, 1122 classes and interfaces (including inner classes and interfaces but not anonymous classes) appeared in the system. The evolution profile of each of these classes was analyzed and Table 6-4 reports the number of different types of phases in their evolution profiles.

Table 6-4. The summary of evolution phases in JFreeChart

| | Intense evolution | Rapidly developing | Restructuring | Slowly developing | Steady-state |
|---------------|-------------------|--------------------|---------------|-------------------|--------------|
| Start with | 9 | 325 | 15 | 706 | 67 |
| In the middle | 87 | 47 | 654 | 197 | 1530 |
| End with | 15 | 199 | 16 | 269 | 623 |
| Remove with | 11 | 176 | 9 | 225 | 16 |

A “Start with” phase is the first phase of the class lifecycle, following its introduction to the system. An “End with” phase is the last phase of the class. If a class happens to be

removed from the system – either before or at the last version of the system – its “End with” phase is also a “Remove with” phase; that is, “Remove with” phases are a subset of “End with” phases. All other class lifecycle phases, between the “Start with” and “End with” phases of classes are “In the middle” phases.

6.4.1.1 How are classes introduced into system?

Most classes (773 out of 1122, about 69%) were introduced into the system with an initial slowly developing or steady-state phase. These classes include demo classes (used to demonstrate features such as how to use CategoryPlot), JUnit classes, change-event classes (encapsulating information about a change to a particular chart) and their corresponding listener interfaces, utility classes (with static methods and/or public constants to be used by other classes), high-level interfaces, simple classes and subclasses of almost fully implemented abstract classes. With few exceptions, these classes were simple, they did not incorporate too much application logic, and they did not change much after their introduction to the system.

About 29% (325) classes were introduced into system with a rapidly developing phase. They were almost all the core objects of JFreeChart system, which include all the important components of a chart object, such as plot, axis, renderer, title, legend, and concrete dataset. In general, these classes were much more complex than most of the classes introduced with a slowly developing or steady-state phase. They were frequently modified over their lifecycle for function extension and/or restructuring.

6.4.1.2 How do classes evolve after their introduction to the system?

After their introduction into the system, most classes and interfaces went through steady-state phases (1530 out of 2509, about 61%). Only a small fraction (244 out of 2509, about 10%) of phases that classes went through are rapid or slow development phases, indicating function extension. Intense evolution and restructuring phases account for the rest 30% phases of class evolution. The classes that are introduced later are more likely to have intense evolution, rapid development, and restructuring phases.

Based on this data, it seems that the JFreeChart system classes and interfaces had most of their functionalities ready in the beginning of their evolution, and afterwards they were sometimes extended with new features, but they were most often restructured to better fit

in the whole system. There can be two explanations for this phenomenon. JFreeChart may have been developed following a design-driven development process, with a detailed requirement analysis and high-level system design upfront. Having made all this analysis in the beginning, the JFreeChart developers would know what architecture to adopt, what functionalities to support and how to implement them. A more agile development process would exhibit a more incremental function-extension development style. On the other hand, it could be that distinct modules delivering well-defined features are contributed by individual developers, who do their incremental development outside the project repository. Once the modules are mature, they are added into system and then they do not or rarely evolve any more. This latter type of process would be more in tune with the overall open-source development model, where individuals develop “patches” that are submitted to the project maintainers and are included only after they have been thoroughly tested.

6.4.1.3 When do classes disappear and where do they go after they are removed?

Most of the system classes went gradually into a steady state, since about 55% (623 out of 1122) of the evolution profiles of classes and interfaces ended up with steady-state phases in the most recent system releases. Interestingly, the classes that ended with active, rapidly developing, restructuring, and slowly developing phases were mostly removed from the system (see the fourth row of Table 6-4). This phenomenon indicates that as long as the classes and interfaces still exist in the system, they undergo fewer and fewer modifications and eventually become stable.

Finally, let us look more closely at these fairly rare classes and interfaces that exhibited intense evolution or restructuring phases in the beginning and/or at the end of their lifecycle. Some of them were “legacy” classes [102]: they were introduced into the system as placeholders for fields and methods moved in from other classes. Some were “die-hard” classes [102], i.e., they were removed from the system when their functionalities were moved out to other classes. These types of classes represent evidence of redistribution of functionality or reorganization of the class hierarchy.

Several distinct types of legacy classes were identified in JFreeChart. Some of them, such as ChartPanelConstants, were the product of extracting constants into an interface.

Others, such as `PlotRenderIno` or `ObjectTable`, were the result of *extract class* or *extract superclass* refactoring. Finally, yet other legacy classes, such as `AxisLocation`, were the product of even more complex refactorings, such as *replace type code with class*.

Several classes, such as `Vertical/HorizontalCategoryAxis`, `Vertical/HorizontalCategoryPlot`, were removed from system in version 0.9.9. They ended up with intense evolution or restructuring phases. These classes exhibited similar evolution profiles to each other as discussed in Section 6.4.4. Such evolution similarity implies that the vertical and horizontal sibling subclasses of `CategoryAxis` and `CategoryPlot` most likely do similar things in a similar or even exactly the same manner. In fact, before version 0.9.9, there existed parallel `Plot`, `Axis` and `Renderer` hierarchies in the `JFreeChart` system. This fact caused a lot of code duplication into the system, which is among the most common “bad smells” in software systems. In version 0.9.9, the `JFreeChart` developers made some important changes to the system design, one of which is to redevelop the `Plot`, `Axis`, and `Renderer` hierarchies, which resulted in the above vertical and horizontal axis and plot classes becoming die-hard classes.

We also found several pairs of die-hard and legacy classes, such as `CrosshairInfo` in version 0.9.16 and `CrosshairState` in version 0.9.17. These pairs are the results of renaming and moving classes and interfaces at the same time. As we have already discussed, to be more efficient, *UMLDiff* ignores synchronous renamings and moves of entities: this would require the pair-wise comparison of all the not-yet-mapped model elements, which would be very inefficient. However, such renaming-and-moves are very likely to be recovered by querying for pairs of die-hard and legacy classes.

6.4.2 Relative order of evolution phases

Gamma analysis examines the relative order of the various phase types in the class evolution profiles, in order to reveal any consistent relative-ordering relations between them. For example, according to the agile-development lifecycle, function-extension phases should be interleaved with refactoring phases. If a project follows an agile-development style, then gamma analysis should reveal that rapidly and/or slowly developing phases are followed by intense evolution and/or restructuring phases and they appear periodically.

Table 6-5. Pair-wise gamma scores of class Axis

| | Intense evolution | Rapidly developing | Restructuring | Slowly developing | Steady-state |
|--------------------|-------------------|--------------------|---------------|-------------------|--------------|
| Intense evolution | 0.000 | 1.000 | -0.750 | 0.000 | -0.579 |
| Rapidly developing | -1.000 | 0.000 | -1.000 | -1.000 | -1.000 |
| Restructuring | 0.750 | 1.000 | 0.000 | 0.750 | 0.105 |
| Slowly developing | 0.000 | 1.000 | -0.750 | 0.000 | -0.474 |
| Steady-state | 0.579 | 1.000 | -0.105 | 0.474 | 0.000 |

Table 6-6. Separation and precedence scores of class Axis

| | Intense evolution | Rapidly developing | Restructuring | Slowly developing | Steady-state |
|------------|-------------------|--------------------|---------------|-------------------|--------------|
| Separation | 0.582 | 1.000 | 0.651 | 0.556 | 0.539 |
| Precedence | 0.082 | 1.000 | -0.651 | 0.056 | -0.487 |

Take the class Axis as an example. The pair-wise gamma scores are computed for each possible pair of phase types as shown in Table 6-5. Table 6-6 shows the corresponding precedence and separation scores for five phase types. Finally, the gamma map of the Axis' evolution profile is:

<Rapidly developing, Intense evolution, Slowly developing, Steady-state, Restructuring>.

The gamma map displays the phase types in their precedence order. The separation scores of all five phase types are greater than 0.50, which means that all these phase types are sufficiently distinct from each other. The Gamma map abstracts the overall sequential pattern from a phase map. In the case of the Axis class, refactoring (i.e. intense evolution or restructuring) phases follow the function-extension (i.e., rapidly or slowly developing) phases. In [103], we studied the evolution phases of Mathaino project [92] – a research project developed by a single developer using a refactoring-driven process, and we observed the similar interweaving phenomena between function-extension and refactoring phases at the system level.

In addition to the relative order of evolution phases of individual system classes, we are also concerned with how the two types of evolution phases followed each other for all the system classes as a whole.

6.4.2.1 How frequently should the classes be refactored in JFreeChart system?

Refactoring phases are interjected in the system-development lifecycle to keep the system design from deteriorating after adaptations in support of function extension. However, in practice, different projects inject in their process refactoring phases with different frequency. The question then becomes, for a given project how frequently should the developers attempt to refactor it?

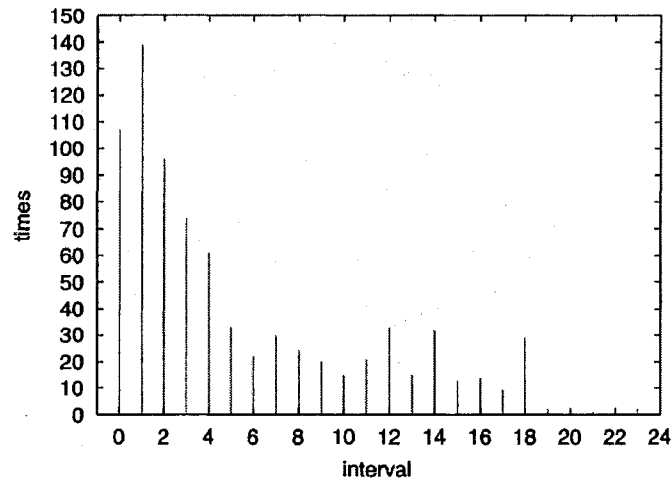


Figure 6-1. Refactoring frequency

Figure 6-1 summarizes the restructuring frequency of JFreeChart's class evolution. The horizontal axis represents the interval between two refactoring phases or the number of rapidly developing, slowly developing and steady-going phases before the first refactoring phase, while the vertical axis show how many times refactoring are made at a particular interval.

As shown in Figure 6-1, there are 107 times that a refactoring phase immediately follows a previous refactoring phase, at interval 0. Note that 9+15 of them are instances of classes that start with intense evolution or restructuring phases (see Table 6-4). Therefore, there are actually 83 instances of consecutive refactoring phases. In addition, there are 139 refactoring phases separated from another refactoring phase with only one phase of another type (most frequently a slowly-developing or steady-state one). These 222 (about 29% out of total 771) close refactoring phases represent almost continuous

refactoring work of a small set of related classes over a short time period. For example, in JFreeChart, tooltips and label-related features are continuously refactored from version 0.9.16 to 0.9.20.

There are 318 refactoring phases that are separated by five or more phases of other types. Close inspection revealed that about 190 out of 318 (about 24% of the total 771) refactoring phases are the result of three major package splits. This is not surprising, given that a phase is considered as “restructuring” as long as the corresponding class is renamed or moved in that phase, irrespective of what other types of changes it has also suffered. Therefore, only 128 cases (about 16% of total 771) involve refactoring activities really far apart of each other.

In about 30% of the 771 cases, refactoring phases are separated from each other with two to four phases of other types.

Given the consistently frequent refactoring phases in the JFreeChart classes, which may be the result of an established practice or even an explicit project policy, a new JFreeChart developer would likely be advised to refactor a class, after no more than four consecutive function-extension phases. This project-specific advice – although by no means definitive – is very important and should be recovered and presented to the developers in the context of a particular system.

6.4.3 Class-evolution styles

We applied optimal matching analysis to compute the pair-wise similarity of any two class evolution profiles in a numerical distance index. The resulting distance matrix was then subjected to cluster analysis in order to generate the groups of classes with similar evolution styles. This analysis revealed several similarly evolving clusters of classes.

The first, and most obvious, clusters correspond to the demo and the JUnit test classes. As discussed in above section, in general, these classes were simple and they did not change much after their introduction into the system.

There are three much more interesting clusters of classes, each one exhibiting a distinct evolution style: shorted-lived classes, idle classes, and active classes.

Short-lived classes: The characteristic of the clusters of short-lived classes is that they exist only in a few versions of the system and then disappear. There exist groups of

classes with very short lives, such as for example, `ObjectTable` and its subclasses that were introduced in version 0.9.9 and were removed in the following version 0.9.10, and `DataSource` and its subclasses that existed only in the first version, 0.5.6, and then disappeared in version 0.6.0. These classes were actually moved to a related library, `JCommon` [127].

The more interesting short-lived classes are such classes as `VerticalDateAxis`, and `VerticalColorBarAxis` and `HorizontalLogarithmicColorBarAxis` and `VerticalLogarithmicColorBarAxis`. They were introduced into system in version 0.9.5. They actually contained almost the same fields and methods as the classes `HorizontalDateAxis` and `HorizontalColorBarAxis` respectively. They also exhibited very similar efferent and afferent usage to the `HorizontalDateAxis` and `HorizontalColorBarAxis` classes respectively. Few changes were made to them since their introduction into the system, and these duplicate classes were finally removed in version 0.9.9 where the major development was done to redevelop the axis, plot, and renderer hierarchy to remove the parallel inheritance (see Section 6.4.4).

Idle classes: The major characteristic of the idle-classes' clusters is that they rarely undergo changes after their introduction into system. There were several distinct types of idle classes discovered in `JFreeChart`. First, there were some stand-alone utility classes. Second, root abstract classes and interfaces were mostly idle. An exception was the root abstract classes of axis, plot, and renderer hierarchies, such as class `Axis`, `CategoryPlot`, `AbstractRenderer`, are, in fact, active. Third, concrete subclasses of almost fully-developed abstract superclasses tend to be idle. For example, the strategy pattern was applied to the needle classes used by the `CompassPlot` class. The abstract superclass `MeterNeedle` was well-defined and all its subclasses only need to override the `drawNeedle()` method to provide different algorithms to draw different shapes of needles. These subclasses rarely changed after they were added to the system. Finally, some features contributed together by the same author were rarely modified after they were introduced into system. For example, the classes `ContourValuePlot`, `StandardContourToolTipGenerator`, `ContourEntity`, `ContourToolTipGenerator`, and `ContourDataset` were added in version 0.9.5, contributed by a single author who probably tested them thoroughly before adding them to the system.

Active classes: Active classes keep being modified over their whole lifespan. We already discussed the fact that many of the core components of `JFreeChart` system were

active classes. Since JFreeChart is still under active development, it should be normal that these core classes are still volatile and undergo substantial modifications.

6.4.4 Class co-evolution

Frequently sets of classes exhibit “parallel” evolution profiles, due to hidden inter-dependencies among them, not necessarily visible when examining their relations at the design stage. A simple indicator of the co-evolution symptom can be obtained by qualitatively clustering together classes that exhibit similar gamma maps.

Table 6-7. VerticalCategoryAxis vs. HorizontalCategoryAxis

(a) Evolution profiles

| Version | VerticalCategoryAxis | | | | HorizontalCategoryAxis | | | |
|---------|----------------------|--------|--------|------|------------------------|--------|--------|------|
| | Add | Remove | SigCha | Move | Add | Remove | SigCha | Move |
| 0.5.6 | 8 | 0 | 0 | 0 | 11 | 0 | 0 | 0 |
| 0.6.0 | 2 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| 0.7.4 | 3 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0.9.0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| 0.9.3 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| 0.9.4 | 0 | 0 | 2 | 0 | 1 | 0 | 3 | 0 |
| 0.9.5 | 5 | 1 | 5 | 1 | 6 | 2 | 5 | 1 |
| 0.9.7 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0.9.9 | 0 | 10 | 0 | 7 | 0 | 10 | 0 | 16 |

(b) Gamma maps

| | Gamma map |
|------------------------|---|
| VerticalCategoryAxis | Slowly developing Steady-state Restructuring Intense evolution |
| HorizontalCategoryAxis | Rapidly developing Slowly developing Steady-state Restructuring Intense evolution |

Consider for example, the classes VerticalCategoryAxis and HorizontalCategoryAxis: their evolution profiles are shown in Table 6-7 (a) and their corresponding gamma maps are shown in Table 6-7 (b). Inspecting their gamma maps, as a high-level abstraction of their

evolutionary patterns, it is evident that their evolution paths are very similar. The only difference between their gamma maps is that class `HorizontalCategoryAxis` started with a rapidly developing phase. Next, they both had slowly developing phases, followed by steady-state phases, followed by restructuring phases, and they finally ended up with an intense evolution phase.

An exact similar behavior is also found in the plot and renderer hierarchies, as shown in Table 6-8 by the evolution profiles of the classes `VerticalCategoryPlot` and `HorizontalCategoryPlot`.

Table 6-8. The evolution profiles of `VerticalCategoryPlot` vs. `HorizontalCategoryPlot`

| Version | VerticalCategoryPlot | | | | HorizontalCategoryPlot | | | |
|---------|----------------------|--------|--------|------|------------------------|--------|--------|------|
| | Add | Remove | SigCha | Move | Add | Remove | SigCha | Move |
| 0.5.6 | 18 | 0 | 0 | 0 | 18 | 0 | 0 | 0 |
| 0.6.0 | 4 | 0 | 7 | 0 | 2 | 1 | 4 | 0 |
| 0.7.1 | 2 | 0 | 6 | 0 | 2 | 0 | 4 | 0 |
| 0.7.2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.7.3 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.7.4 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.8.0 | 0 | 3 | 4 | 2 | 0 | 2 | 4 | 2 |
| 0.9.0 | 4 | 3 | 10 | 2 | 3 | 2 | 12 | 2 |
| 0.9.3 | 0 | 0 | 5 | 0 | 0 | 0 | 5 | 0 |
| 0.9.5 | 1 | 5 | 2 | 1 | 1 | 3 | 2 | 1 |
| 0.9.7 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.9.9 | 0 | 9 | 0 | 5 | 0 | 9 | 0 | 5 |

However, such qualitative clustering is likely to miss important distinctions among larger collections of complex sequences. In such cases, data-mining technique, such as the one discussed in Section 6.3, can be applied to recover and make explicit such “hidden knowledge”. By applying association-rule mining, we discovered an interesting set of co-evolving classes that consists of the (a) `CategoryPlot`, `HorizontalCategoryPlot` and `VerticalCategoryPlot` and their subclasses, (b) `BarRenderer`, `VerticalBarRenderer` and `HorizontalBarRenderer` and their subclasses, and (c) `CategoryAxis`, `DateAxis`, `NumberAxis`, `SymbolicAxis` and their corresponding horizontal and vertical axis subclasses. We also discovered some less remarkable sets of co-evolving classes, such as for example `Plot` and `Axis`. They exhibit three types of co-evolution:

- Function-extension => function-extension
- Refactoring => refactoring
- Function-extension => refactoring

We discuss concrete examples for different types of co-evolution in the following subsections. The data-mining algorithm also produced some co-evolution rules of type refactoring => function-extension. However, after closely inspecting the participants of these co-evolution relationships, we believe that they are most likely accidental.

6.4.4.1 Parallel function extension

The system we used in Section 6.3 to illustrate the co-evolution detection process is actually a snippet of the evolution profiles of the BarRenderer class and its subclasses. Among them, C1 is VerticalBarRenderer, and C6 is HorizontalBarRenderer, which are responsible for the drawing of bars in horizontal and vertical category plot respectively. They are both direct subclasses of the BarRenderer class. Table 6-9 shows their evolution profiles side by side. The column (from left to right) represents the number of changes, i.e., addition, removal, signature-change and movement (including class move) respectively. As we can see, these two classes exhibit almost the exact same types and amount of changes in each version.

Table 6-9. VerticalBarRenderer vs. HorizontalBarRenderer

| Version | VerticalBarRenderer | | | | HorizontalBarRenderer | | | |
|---------|---------------------|--------|--------|------|-----------------------|--------|--------|------|
| | Add | Remove | SigCha | Move | Add | Remove | SigCha | Move |
| 0.6.0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 0 |
| 0.7.1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.7.3 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0.8.0 | 1 | 0 | 3 | 2 | 1 | 0 | 3 | 2 |
| 0.9.0 | 2 | 0 | 2 | 0 | 2 | 0 | 2 | 0 |
| 0.9.2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.9.4 | 2 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| 0.9.5 | 0 | 2 | 5 | 3 | 2 | 2 | 6 | 1 |
| 0.9.7 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 |
| 0.9.8 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.9.9 | 0 | 6 | 0 | 2 | 0 | 9 | 0 | 1 |

Furthermore, the association-rule mining algorithm produces the following rules with high value of support and confidence:

VerticalBarRenderer=D (3) => HorizontalBarRenderer=D (3) [confidence=1]

HorizontalBarRenderer=D (3) => VerticalBarRenderer=D (3) [confidence=1]

These rules indicate that when adding and/or removing features from class VerticalBarRenderer, the developer always (the confidence of these rules is equal to one) performed similar modifications to the HorizontalBarRenderer class, and vice versa. For example, both of them were introduced into the system in version 0.6.0, and they are both removed in version 0.9.9. In version 0.9.4, constructors that take as input parameters of type CategoryURLGenerator and CategoryToolTipGenerator were added to both classes. In version 0.9.9, the methods barWidthsPerCategory(), hasItemGaps(), and drawRangeMarker() were removed from both of them. Clearly, these two sibling classes exhibit the type of *function-extension => function-extension* co-evolution.

6.4.4.2 Parallel refactoring

Let us look at another type of co-evolution. VerticalBarRenderer, VerticalBarRenderer3D and StackedVerticalBarRenderer3D. VerticalBarRenderer3D extends VerticalBarRenderer and is responsible for drawing vertical bars with 3D effect. StackedVerticalBarRenderer3D extends in turn VerticalBarRenderer3D. Table 6-10 lists their evolution profiles. They have undergone very similar changes, and Apriori mining discovers the following association rules based on their evolution profiles:

VerticalBarRenderer3D=T (3) => StackedVerticalBarRenderer3D=T (4) [confidence=1]

StackedVerticalBarRenderer3D=T (4) => VerticalBarRenderer3D=T (3) [confidence=0.75]

VerticalBarRenderer=T (3) =>

VerticalBarRenderer3D=T (3), StackedVerticalBarRenderer3D=T (4) [confidence=1]

These association rules imply that in three out of four times that the interface of class VerticalBarRenderer3D was modified, the interface of its subclass StackedVerticalBarRenderer3D was also refactored. In addition, in three out of four times that the class VerticalBarRenderer was refactored, the interface of its subclasses, VerticalBarRenderer3D and StackedVerticalBarRenderer3D (indirectly extends) were also modified.

Table 6-10. VerticalBarRenderer, VerticalBarRender3D, StackedVerticalBarRender3D

| Ver | VerticalBarRenderer | | | | VerticalBarRender3D | | | | StackedVerticalBarRender3D | | | |
|-------|---------------------|------|------|------|---------------------|------|------|------|----------------------------|------|------|------|
| | Add | Remo | SigC | Move | Add | Remo | SigC | Move | Add | Remo | SigC | Move |
| 0.6.0 | 6 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 5 | 0 | 0 | 0 |
| 0.7.1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.7.3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0.8.0 | 1 | 0 | 3 | 2 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 |
| 0.9.0 | 2 | 0 | 2 | 0 | 7 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.9.1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.9.2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.9.4 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0.9.5 | 0 | 2 | 5 | 3 | 14 | 5 | 10 | 1 | 1 | 3 | 6 | 2 |
| 0.9.7 | 0 | 0 | 2 | 0 | 2 | 0 | 3 | 0 | 0 | 0 | 2 | 0 |
| 0.9.8 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0.9.9 | 0 | 6 | 0 | 2 | 0 | 22 | 0 | 1 | 3 | 0 | 6 | 0 |

In particular, in version 0.8.0, `VerticalBarRenderer` started extending the `BarRenderer` class and implementing the `CategoryItemRenderer` interface. However, in its previous version, it extends and implements nothing (that is, extends `java.lang.Object`). In the same version, its method `drawBar()` was renamed to `drawCategoryItem()`. The `drawBar()` method of `VerticalBarRender3D` and `StackedVerticalBarRender3D` were also renamed to `drawCategoryItem()`. Furthermore, depending on inheritance-hierarchy transitive closure, *UMLDiff* is able to report that these two subclasses also started implementing the `CategoryItemRenderer` interface in version 0.8.0. In addition, all these renderer-related classes were moved in a newly created package `com.jrefinery.chart.renderer` in version 0.9.5. Finally, in version 0.9.7, the signature of the `drawItem()` method of all these three classes was modified to take as input a parameter of type `KeyedValues2DDataset`, instead of its earlier parameter of type `CategoryDataset`. Moreover, `VerticalBarRenderer` started to implement interface `java.io.Serializable` in version 0.9.7, which affects the interface of its subclass `VerticalBarRender3D`. Two new methods, `readObject()` and `writeObject()`, were added to `VerticalBarRender3D`, and in addition, its field `wallPaint` was declared to be transient in this version.

This set of changes indicates the strong *refactoring* => *refactoring* co-evolution relation between a superclass and its subclasses.

6.4.4.3 Parallel function extension and refactoring

Apriori also identified interesting co-evolutions of classes in different parts of the inheritance hierarchy. The classes in this example are unlike the previous two examples, in that they have substantially different identifiers that make their “accidental” discovery even more unlikely. For example:

Plot=D (4) => Axis=T (10) [confidence=0.5]

These are the root abstract classes of the plot and axis hierarchies respectively. All plot objects implement, directly or indirectly, the methods of the Plot class. Plot objects control the drawing of Axis objects. They hold the instances of Axis objects and the attributes, such as location, space, offset, of axis, and delegate the actual drawing to Axis objects. Therefore, the changes made to the Plot class frequently affect the interface of the Axis class. For example, in release 0.9.10, two methods that take as input a parameter of type AxisLocation were removed from the Plot class and corresponding methods that take as input a RectangleEdge parameter were added. Consequently, several methods in Axis class, draw(), getLabelEnclosure(), drawLabel(), reserverSpace(), and refreshTicks() had their signatures changed to use the parameter of RectangleEdge instead of AxisLocation. This is an example of the function extension in one part of system class model resulting in the refactoring in some other part of hierarchy.

Such function-extension => refactoring rules indicates an intentional interdependency between classes, which requires certain classes to be modified in this particular way. In this sense, the detection of fine-grained class co-evolution can serve as a design-recovery tool that elicits the implicit (possibly not well-documented or even lost) interdependency between classes.

6.4.4.4 Discussion on parallel-inheritance co-evolution

The elegance of the data-mining method is that it can quickly bring to surface the interesting patterns, which can focus the developers’ attention directly to a potentially problematic area.

The parallel refactoring class co-evolution may be necessary, since the developer is trying to change the class interface, which probably should ripple up to the superclass and down to the subclasses in order not to break the collaboration with other classes. The

function-extension co-evolution between sibling classes may indicate a potentially intentional co-evolution. The developers should be advised that if they add and/or remove some features from one of such sibling classes, they probably should also look at the other one in order to keep the interface consistent.

However, if refactoring the interface or extending the features of some classes often means that the developers have to perform similar changes in some of its sibling classes, such as those render classes we discussed in the above sections (please note that the similar cases exist also for the plot and axis hierarchy), this is most likely a good indicator that there is something wrong with the design of the involved classes, or even the overall relevant hierarchy. Such co-evolution means that sibling classes do similar things in a similar or even exactly the same manner. This would introduce a lot of code duplication into the system, which is among the most common “bad smells” in software systems. As the system evolves, it may cause more and more maintenance efforts, since when a change is necessary, all of them have to change in a similar manner. Even worse, such hidden relations are easily lost due to the team-member turnover and they are hard to impart to the new members.

The existence of such large amount of co-evolution within the Plot, Axis, and Renderer hierarchy strongly suggests that there may exist multiple parallel inheritance hierarchies in the JFreeChart system, which is validated by the major design changes made for the release 0.9.9. In version 0.9.9, the JFreeChart developers redesigned the Plot, Axis, and Renderer hierarchies, which are captured by *UMLDiff* and our refactoring-detection process:

- The classes `HorizontalCategoryPlot` and `VerticalCategoryPlot` were removed and became “die-hard” classes [102], i.e., some of their members, such as `render()`, `handleClick()`, were pulled up into the superclass `CategoryPlot`. Some methods of the `CategoryPlot` class had their parameter list extended with one more parameter of `PlotOrientation` type, which was just added in this release.
- `OverlaidVerticalCategoryPlot` (there is no `OverlaidHorizontalCategoryPlot`) was renamed into `OverlaidCategoryPlot` to keep the naming convention consistent.
- The classes `HorizontalCategoryAxis` and `VerticalCategoryAxis` were removed and they too became die-hard classes. Some of their members were merged into the

superclass `CategoryAxis`. The same changes were made to `HorizontalDateAxis` and `VerticalDateAxis` and their superclass `DateAxis`.

- Class `HorizontalLogarithmicAxis` was renamed as `LogarithmicAxis`, and `VerticalLogarithmicAxis` was removed. A similar change was made to horizontal and vertical `SymbolicAxis` and `ColorBarAxis`.
- Class `HorizontalBarRenderer` and `VerticalBarRenderer` were removed and their subclasses, `Horizontal/VerticalInternalBarRenderer`, `Horizontal/VerticalStatisticsBarRenderer`, `StackedVertical/HorizontalBarRenderer` were merged into `IntervalBarRenderer`, `StatisticalBarRenderer`, `StackedBarRenderer` respectively, which extend `BarRenderer` directly in this release. For example, the `VerticalIntervalBarRenderer` class was renamed to `IntervalBarRenderer`, and `HorizontalIntervalBarRenderer` was removed.

All these recovered design changes indicate that the JFreeChart developers became at some point aware of the existence of separate horizontal and vertical hierarchies, and in release 0.9.9, they made a great effort to reorganize the Plot, Axis, Renderer hierarchies and to eliminate the parallel inheritance and to reduce duplicate code. These design changes constitute evidence of suggestions that our longitudinal design-evolution analysis could have provided, based on its discovery of design-evolution smells, which could have been quite helpful.

6.5 Summary

In this chapter, we studied the longitudinal evolution of object-oriented software systems with three sequential-pattern analyses and the association-rule mining method. These longitudinal analyses rely on the design changes reported by *UMLDiff*. Our study showed that there exist distinct evolution phases with coherent evolution behavior in the evolution of the system and its classes. They exhibit repetitive patterns in their relative-order relations, consistent with the adopted development process. The classes demonstrate distinct evolution styles characteristic of their roles in the system. As software systems evolve over a long time, non-trivial and often unintended inter-dependencies among system classes arise, because of implicit design dependency or evolution smells. These longitudinal analyses facilitate the overall understanding of

system evolution, help to recognize system instabilities, and provide support regarding the scope of future maintenance activities. Together with recognizing design-change patterns, such as refactorings, they constitute a solid base for mentoring object-oriented evolutionary development, based on learned experiences from past evolution activities.

Chapter 7: Exploring Design Evolution Concerns

Software system usually grows in size and complexity as it evolves over time [58]. The developers face increasing difficulties in comprehending the system design and its rapid evolution, since the amount of information is overwhelming. The top-down style of [29,35,57,110] visualization does not work very well to precisely capture the changes and their underlying motivations since they generally start with an overview of the whole subject system and assume their users to be able to drill down to the interesting parts of the system evolution.

This is why, in our work, we have adopted a bottom-up approach to design-evolution analysis. First, the elementary design changes are detected by *UMLDiff* algorithm, based on which, a suite of longitudinal evolution analysis methods [1,2,75,84] and a set of refactoring-detection queries are then applied to recognize interesting evolution concerns (i.e., evolution phases and styles, co-evolving software artifacts, and refactorings) in the evolution history of individual system classes, clusters of classes and the system as a whole. In this chapter, we introduce our visualization component – JDEvAn Viewer²¹. Given the key participant model elements and relations of an evolution concern, a so-called *core evolution concern*, JDEvAn Viewer visualizes them with change tree and UML class diagram, and it supports developers to interactively create, explore and maintain the recovered evolution concerns they are interested in.

The software developers start with the minimum amount of information about the core evolution concerns. Then, they can iteratively augment the core evolution concerns with the relevant model elements, their relations, and their changes by querying logical 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 7-1 displays such a snapshot, at some point in our investigation process, of two sets of co-evolving classes and the refactorings that address these co-evolution smells.

²¹ The JDEvAn Viewer has been implemented as an Eclipse plugin and it relies on the Eclipse GEF (Graphical Editor Framework) [117]

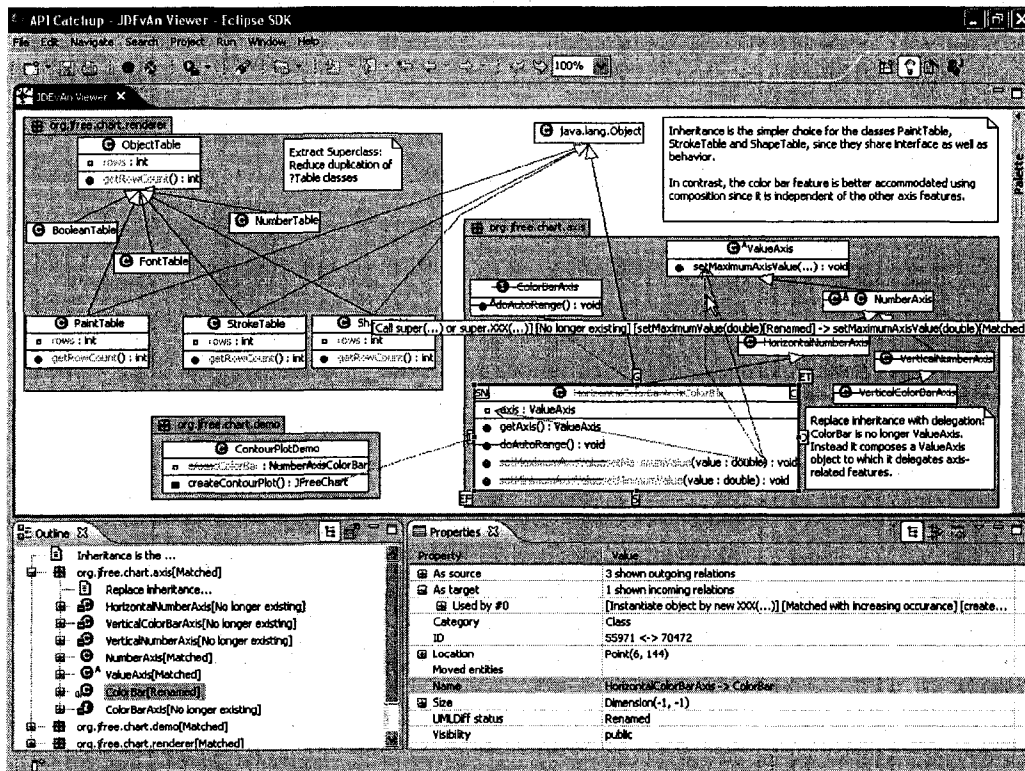


Figure 7-1. A screenshot of JDEvAn Viewer

7.1 The JDEvAn Viewer

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

7.1.1 Presenting design-evolution concern

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 *Outline* view depicts 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 [104]. 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 versa. The thumbnail outline 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 7-1, for example, the renamed class `ColorBar` is selected. Its corresponding properties sheet lists its element type, visibility, name, *UMLDiff* status, unique 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 logical models *UMLDiff* compares and its comparison results in the form of UML class diagram. In the evolution concerns shown in Figure 7-1, three packages are under investigation, each of which contains one or more classes. The classes declare attributes and operations, 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 and relations is visualized by coloring the name (identifier for operation) 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: Renamed²² 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 operation) is shown with a strikeout line as well. Furthermore, the matched parameters of operations are initially hidden with “...” placeholder, which can be expanded and collapsed by clicking the “+” or “-” handle of the placeholder. For data/return/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 7-1, 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

²² The renamings of operation include the changes to their identifiers and/or parameter lists.

setMinimumAxisValue() are renamed to setMaximumValue() and setMinimumValue() 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 7-1, the cursor is pointing to a no longer existing super-call relationship between the renamed method ColorBar.setMaximumValue(double) and the matched method ValueAxis.setMaximumAxisValue(double).

7.1.2 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 7-1. Table 7-1 summarizes the applicable handles attached to various types of model elements.

The handles allow the 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 7-1, 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, 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.

Table 7-1. The handles attached to model elements in JDEvAn Viewer

| Handle | Model element | Handle usage |
|--------|-----------------------------|---|
| G | Class | The direct superclass and the (direct or transitive) super-interfaces |
| | Interface | The (direct or transitive) superinterfaces |
| S | Class | The direct subclasses |
| | Interface | The (direct or transitive) subinterfaces and implementation classes |
| O | Class | The classes and interfaces it uses |
| | Operation | The attributes it reads and/or writes The operations it calls The objects it instantiates The exceptions it declares, throws, and/or catches |
| | Class | The classes that use it The operations that instantiate it The operations that declare, throw, and/or catch it as exception |
| | Interface | The classes that use it |
| I | Operation | The operations that call it |
| | Attribute | The operations that read and/or write it |
| | Subsystem | The model elements it contains or declares |
| C | Package | |
| | Class | |
| | Interface | |
| | Attribute | The declared type |
| T | Method | |
| | Parameter | |
| | Moved element | The move source (target) elements of the given move target (source) element |
| SN | All | The similar name elements of the same type, based on regular expression of the words in the element name |
| EF | All but the new element | The predecessor elements of the given element in previous versions |
| ET | All but the removed element | The successor elements of the given element in following versions |

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.

7.1.3 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, starting at a particular version, such as how a set of elements are introduced in the system, what are their states before refactoring and how they evolve into these states, what benefits the refactoring brings about, and so on.

7.1.4 Attaching user comments

The JDEvAn Viewer allows developers to attach one or more comment(s) to model elements and relations and their changes to record the hard-earned evolution knowledge. For example, in Figure 7-1, 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.

7.1.5 Requesting source code

As users investigate the evolution of software system at the design level, a mapping between the design-level representation and the source code corresponding to each model element is maintained, 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, the Eclipse Java Editor is shown with the corresponding code fragment highlighted. If the model element is mapped, the Eclipse Compare Editor or Dialog pops up to show the textual comparison results of the source code of the double-clicked element.

7.1.6 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 why the system evolved the way it has. 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 next chapter, 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.

7.2 Demonstrations

In this section, we demonstrate, through two pairs of design-evolution concerns from our HTMLUnit and JFreeChart case studies, how JDEvAn Viewer facilitates the understanding of the system's design and its evolution. In particular, it helps us capture:

- The different motivations behind the two seemingly similar *extract class* refactorings;
- The different design remedies that address the two similar class co-evolution smells.

7.2.1 Different problems but same solution

The types of refactorings that can be automatically detected constitute the basic building blocks for accomplishing many other refactoring tasks, listed in Fowler's refactoring

catalog [32]. Table 7-2 lists some of these refactorings (right column) and their corresponding core refactorings (left column), which can be automatically detected by our refactoring-detection 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 and 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 7-2. The motivations of 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 refactoring-detection queries 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). Software developers can then examine the refactoring participants and the relevant model elements, relations, and their changes with the support of JDevAn Viewer and draw their own conclusions regarding the motivation and rationale behind the given refactoring (*why* they have changed).

Let us examine two particular instances of *extract class* refactoring in the evolution of the HTMLUnit and JFreeChart system 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*.

7.2.2 Same problem but different solutions

Applying Apriori association-rule mining to class evolution profiles 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. It seems that these classes suffered from the smell 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 [32], 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, refactoring-detection 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 7-1 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 interfaces 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 7-1: 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 (i.e. white-box 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 7-1. 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 color-bar feature 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-related features.

We finally annotated these two evolution concerns, including evolution traces of these co-evolving classes and the corresponding instances of refactorings, with the above conclusion with JDEvAn Viewer’s comment node as shown in Figure 7-1, 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 encountering similar smells. For example, when they are faced with class 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.

7.3 Summary

To enable an intuitive means of communicating all the design changes and evolution patterns produced by *UMLDiff* and the subsequent analyses it enables, we developed JDEvAn Viewer. The JDEvAn Viewer supports the interactive visualization, annotation and persistence of the recovered evolution concerns. With JDEvAn Viewer, the users can selectively explore the system's design and its evolution, by localizing the relevant elements, relations, and their changes of design-evolution concerns and exploring their neighborhood and evolution traces. It has been helpful in understanding why the system evolved the way it has. We demonstrated how JDEvAn Viewer helps the developer examine two seemingly similar evolution concerns and their participants and captures their completely different underlying software-quality motivations. We plan to conduct an empirical user study in the future to fully evaluate JDEvAn Viewer's features and compare the bottom-up evolution concern understanding it enables with the existing top-down visualization approaches to software evolution understanding.

Chapter 8: Towards Mentoring Object-Oriented Evolutionary Development

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 [12,26,48,74]. 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 software is increasingly 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. Therefore, understanding the system's design rationale becomes – to some extent – synonymous with recognizing interesting changes in its design-evolution history. Experts, such as senior designers, often serve the role of the design mentor, 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.

On the other hand, software itself embodies examples of object-oriented design principles, design and refactoring patterns, and programming hints previously adopted by the system. Several software recommenders [17,44,73,78,108,109] attempt to 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 recommenders 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, revealing the usage of an API, etc.

In this chapter, we discuss our initial work on monitoring and mentoring object-oriented software design and its evolution. This design mentor relies on the *UMLDiff*-based refactoring detection and design-evolution analyses. In Section 6.4.4.4, we discussed a real case in which we discovered several instances of class co-evolution and

we also found evidence that the project developer acted according to the advice that our design mentor would have generated, had it been in place during the system's development. We believe that the very process of recognizing and reflecting upon problematic design-evolution patterns and concrete examples of their associated design remedies in the evolutionary history of the software system can help developers reach informed decisions on their current development and maintenance activities, such as recommending the redistribution of features based on Law of Demeter [60], or advising when and where to apply which refactorings. We evaluated the feasibility of such a design mentor with a real-world pilot study. In particular, we evaluated its ability to

- uncover design changes aimed towards improving the object-oriented design of software, and
- detect opportunities for such changes.

8.1 Design mentoring

Producing a good design is often a daunting task for novice programmers, and so is evolving an existing system in a manner consistent with the rationale behind its design history, since there are 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. They are able to point out problematic patterns in the design model of an artifact and questionable events and trends in its evolution.

The underlying UML meta-model of our work captures the logical design of object-oriented software system. The *UMLDiff* algorithm and the subsequent design-evolution analyses reveal which parts of the system have changed and how exactly. The interactive exploration of design-evolution concerns help us infer why they have changed. A set of queries have been defined to recognize potentially problematic patterns in the UML logical models of the system and the results of the above analyses. Associated with each of these queries are the design changes that have been applied to remedy the similar problems in the past of the system development or general advice 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 helps developers draw informed decisions on solving their current similar problems.

For some design-evolution concerns, such as the same or similar design changes made in a large amount over a short period of time, there may or may not exist the systematic theories behind them, but they represent the project-specific evolution knowledge, which can not be learnt from the textbook. For example, we studied, in Section 6.4.2, the refactoring frequency in the evolution of JFreeChart system and we discovered that, as a project-specific advice, a new JFreeChart developer would likely be advised to refactor a class, after no more than four consecutive function-extension phases. However, more often than not, such project-specific evolution knowledge is not recorded in the development log; they usually just exist in the developers' minds, as part of their overall software-engineering experience with a particular project. However, our design mentor is able to recover them and present them to developers as a set of contextual advices, which may be valuable to guide future development and maintenance activities.

Furthermore, our design mentor has taken a broader stance to the problem of advising software developers. In addition to providing "contextual" project-specific advice based on the analysis of the system design-evolution history, it is also endowed with knowledge of object-oriented design principles, design patterns and refactorings. Thus, it can relate this knowledge to its understanding of the system under development to offer advice on how to improve the system design based on project independent terms. Design evolution has to be guided by high-level object-oriented design principles, such as adhere to consistent meaningful names, do not unnecessarily expose fields and methods, and comply with the Law of Demeter [60]. At the same time, it is also informed by state-of-the-art practices, such as extract interface, superclass or class, collapse hierarchies, form template method, use typesafe-enum objects instead of numeric type codes.

The instances of these design-evolution patterns, when discovered in the evolution of the software system, indicate that the developers are trying to comply with the well formulated object-oriented design principles and practices, and they can serve as the concrete design and design-evolution examples for developers to learn how to design and

evolve the object-oriented software in general. For example, in Section 7.2.2, we discussed how an advice could be offered, based on instances of refactorings in the project history, to make a choice between *replace inheritance with delegation* and *extract superclass* refactorings in order to correct class co-evolution smell.

8.2 The JFreeChart pilot study

We evaluated the feasibility of the proposed software design mentor with the JFreeChart system. Based on our own software design and development experience, we defined a set of queries to recognize potentially problematic patterns before and in version i . The queries refer to the information regarding the logical models of the subject system of version m ($1 \leq m \leq i$), the design changes reported by *UMLDiff* when comparing version m to the version $m-1$, and the results of the various subsequent analyses. Then, we examined the changes reported by *UMLDiff* when comparing version j to version $j-1$ ($j > i$) to see if the changes implied by the problematic patterns discovered by our queries were actually made. When this was the case, we recorded the corresponding queries as valid heuristic mentors that advise developers on how to maintain and evolve their system based on the object-oriented design principles and practices and/or the understanding of the evolution of the subject system.

8.2.1 Adherence to “first principles” of object-oriented development

In this section, we discuss design changes motivated by high-level principles of object-oriented design.

8.2.1.1 Adopting a consistent, meaningful naming scheme

The adoption of a consistent and meaningful naming scheme is very important in object-oriented design. A case in point is the object-oriented method that advocates the discovery of the system classes from the nouns in the textual requirements specification of the system, and the various renaming refactorings aimed at improving code readability and understandability by alluding to the functions of the design entities.

In JFreeChart, many renamings were discovered. For example, `DEFAULT_COLORBAR_THICKNESS_PCT` was renamed into the more meaningful

DEFAULT_COLORBAR_THICKNESS_PERCENT. Renamings frequently coincide with more “substantial” changes to the entity. In version 0.6.0, fields `categoryGap`, `introGap` and `trailGap` of class `BarPlot` were renamed to `categoryGapPercent`, `introGapPercent` and `trailGapPercent` respectively, to reflect the change of these fields’ type from type `int` to type `double`. Similarly, in version 0.9.3, method `getNormalColor()` of class `MeterPlot` was renamed to `getNormalPaint()` to reflect the fact that its return type was changed from `java.awt.Color` to `java.awt.Paint`.

Note that renamings are among the elementary change facts discovered by *UMLDiff*. Advice on when to rename and how has to be project specific. One could imagine that “renaming critique” could be offered based on project-specific naming scheme, such as “static final constants should be capitalized”, or when other similar design changes happen, such as the data type of an entity changes and the original name of the entity matches a given regular expression.

8.2.1.2 Programming to interfaces

Programming to interfaces and not to implementations is an important tenet of object-oriented development [34]. When the client is implemented to be unaware of the internal implementation of the supplier class, assuming only the specification of its public behavior interface, enables flexibility in the evolution of the supplier: as long as the public interface remains the same, modifications to each implementation will not break its clients.

The design advice here is to not declare fields and methods with particular concrete classes but rather to commit only to interfaces and abstract classes. The more abstractions introduced, the more flexibly can the system implementation evolve. The developers of `JFreeChart` made efforts to comply with this principle. For example, before version 0.7.3, the class `XYPlot` declared four fields (`horizontalColors`, `horizontalLines`, `verticalColors`, `verticalLines`) of the concrete java collection class `java.util.ArrayList`, which were change to the interface `java.util.List` in version 0.7.3. In the same version, the `LinePlot`’s method `getValueAxis()` was changed to return the abstract class `ValueAxis` instead of its subclass `VerticalNumberAxis`. Furthermore, in version 0.9.19, the return type of several methods of

CategoryPlot and XYPlot was changed from the interface `java.util.List` to the more general interface `java.util.Collection`.

The underlying logical model of our work captures the generalization and abstraction relationships, including their transitive closure, among classes and interfaces. We have defined queries to obtain fields, methods and parameters whose associated types are declared as:

- Concrete classes that implement interfaces
- Specialized interfaces that extend general ones
- Subclasses that extend abstract super classes

These fields, methods and parameters are the candidates that should be examined to see if the general interfaces or abstract classes could be used. A customizable stop-list is in place in order to exclude “insignificant” interfaces and classes, such as `ActionListener`, `Serializable`, and `java.lang.Object` in Java, from being considered.

It is interesting to note that a super interface (or class) sometimes needs first to be extracted so that the clients can then start using it instead of its implementations (concrete subclasses). An example will be discussed in Section 8.2.2.1.

8.2.1.3 Favoring composition over inheritance

Object-oriented software engineering enables white-box reuse through class inheritance and black-box reuse through object composition. Frequently, software teams make the “stronger” commitment to white-box reuse when they only need black-box reuse. The result is high coupling among the classes in the inheritance hierarchy, brittleness in the evolution of the base class, and overriding of unwanted features by the subclasses. Object composition enables classes to reuse objects in terms of their well-defined interfaces, with limited implementation coupling and increased flexibility. This is the intent behind the “favor object composition over class inheritance” tenet [34].

In Section 7.2.2, we discussed a poor choice of class inheritance over object composition at the time four `?ColorBarAxis` classes were introduced, which was subsequently amended by a *replace inheritance with composition* refactoring. The evidence for the need to replace inheritance with composition is the simultaneous development of “parallel inheritance hierarchies”; this change is easily recognizable

through the detection of class co-evolution. Of course, there are many other types of “symptoms” where composition could be advocated instead of inheritance, such as subclass inherits many “unwanted” features that are not used at all by its client.

8.2.1.4 Law of Demeter

The “Law of Demeter (LoD)” [60] – “only talk to your friends” - is a simple style rule for object-oriented design. It advocates that the methods of a class should only manipulate the class’ own fields and should call methods defined in the class or the classes whose instances it contains. It is essentially an object-oriented formulation of the general “low coupling” software-engineering principle.

In our case study, the JFreeChart class coordinates such objects as legend, plot, axis and dataset in order to draw a chart on a Java 2D graphics device. In the early versions of the system, it used to delegate the actual drawing to the Plot object it contained. In version 0.5.6, it had four fields, seriesPaint, seriesStroke, seriesOutlinePaint, seriesOutlineStroke that were representing properties of the plot being drawn. Since they were only accessed by plot classes, they should, therefore, be accessed from within the Plot object according to LoD. Indeed, in version 0.6.0, these four fields were moved to the Plot class.

Computing the object form of LoD requires the dynamic analysis of software system. However, there are some symptoms that can be easily detected in the underlying logical models in terms of “high coupling” and “low cohesion”. For example, we have defined queries that return the fields and methods defined in one class but are mostly used in other classes. Such fields and methods often need to be moved in order to enhance encapsulation and reduce coupling.

8.2.1.5 Information hiding

Object-oriented languages provide explicit support for defining the scope of the various design elements of a system. Frequently, developers start off with making elements “too accessible”; as the picture of the scope of the valid clients of each element becomes clearer, the element’s visibility can be restricted.

For example, 519 (about 60% of all the visibility changes, see Table C-2) fields and methods changed their visibility to private in release version 0.9.4, which clearly indicates that JFreeChart underwent an information-hiding restructuring, an observation

validated by the CVS log statement “fix errors reported by CheckStyle”. Checkstyle is a tool to help programmers write Java code that adheres to the coding standard, such as Sun Java Specification.

We have defined queries that return design entities, such as fields, methods, nested classes and interfaces, which are not declared as private but have not been accessed outside their containing elements. Furthermore, visibility changes are one type of elementary changes reported by *UMLDiff*.

8.2.2 Refactorings

Refactoring is one of the most important practices in the agile software-development process, which aims at improving the design of existing code [32].

8.2.2.1 Extracting interfaces

A corollary of the programming-to-interfaces principle is the *extract interface* refactoring. If two or more classes have some stable common behaviors, an interface could be extracted to include the methods delivering the shared behaviors. In this manner, the clients of the refactored classes that are interested in their common behaviors can start depending on the extracted interface, get decoupled from the classes’ implementation and become able to use all implemented classes interchangeably.

We have defined queries that return the classes that declare enough (by enough, we mean over user-specific threshold) same-signature fields and/or methods. For some of the returned classes, the number of same signature fields and/or methods remains the same or changes a little, which indicates that these classes share the stable common interfaces. Among them, we identified instances of *extract interface* refactoring in order to comply with programming-to-interface principle.

For example, classes `HorizontalBarRenderer` and `VerticalBarRenderer` declared five same name fields and methods before version 0.8.0, while in that version, a common interface `CategoryItemRenderer` was extracted; the above classes were modified to implement the new interface and the field `renderer` of their client classes `HorizontalCategoryPlot` and `VerticalCategoryPlot` were pulled up into superclass `CategoryPlot` that declared it as the type

of interface `CategoryItemRenderer` instead of specialized `HorizontalBarRenderer` or `VerticalBarRenderer`.

We also discovered some other cases of interface extraction. In version 0.7.4, seven constants of the `Axis` class were extracted into the newly added interface `AxisConstants` that was then implemented by `Axis` and its subclasses. Similarly, `JFreeChartConstants` was extracted from `JFreeChart`, `ChartPanelConstants` from `ChartPanel` and `CategoryPlotConstants` from `CategoryPlot`. The intent for all these changes must have been to enable the use of the constants by classes other than their original containers. However, further development, it turned out that these constants were only accessed by the classes that originally contained them and the developers decided to move them back from the interfaces to the corresponding classes and remove the corresponding interfaces. We call such classes, whose features remain in the system even after the classes themselves are removed, *die-hard classes* [102].

8.2.2.2 Extracting superclasses

The *extract superclass* refactoring is advisable when two (or more) classes share a substantial part of their members, which also seem to be modified together over time. Again, the instances of classes that share the enough same-signature fields and/or methods are queried for inspection. But this time we are more interested in those classes that show the similar evolution profiles, such as, the same name fields and/or methods are often added to those classes in the same version, which results in the number of same features increasing over time. This is a good indicator of shot-gun surgery [32], which can be fixed by such refactorings as *extract superclass*. An example of extracting superclass `ObjectTable` from co-evolving `PaintTable`, `StrokeTable` and `ShapeTable` has already been discussed in Section 7.2.2.

8.2.2.3 Forming template methods

The template method design pattern [34] is applicable in situations where an algorithm is defined in a superclass, with its overall process and some of its steps being shared by the subclasses as-is, some of the steps being used as defaults when the subclasses do not override them while yet others being overridden or extended by the various subclasses

according to their needs. *Form template method* is one of the complex refactorings identified in the Fowler catalog [32] to get template method pattern.

We have defined queries that search for the sibling classes that declare methods with enough (again, user-specific threshold) same usage dependencies, such as field reads/writes, method calls, and object instantiations, which indicates that these methods do their job in a similar way. Therefore, they are the candidates for further examination of forming template methods.

For example, in version 0.9.19, an superclass `AbstractCategoryItemLabelGenerator` was extracted from class `StandardCategoryItemLabelGenerator`, in which `generateLabelString()` was defined as a template method that called the default tooltip and label implementation defined in method `createItemArray()`. The subclasses, `StandardCategoryLabelGenerator` and `StandardCategoryToolTipGenerator`, implemented the interfaces `CategoryLabelGenerator` and `CategoryToolTipGenerator` respectively and called the template method `generateLabelString()`. The other subclasses, such as `IntervalCategoryLabelGenerator`, overrode `createItemArray()` to provide their specific behaviour.

The class `StandardPieItemLabelGenerator` had the similar condition to `StandardCategoryItemLabelGenerator`. If the `JFreeChart` developers wanted to restructure `StandardPieItemLabelGenerator` later on, the changes made to `StandardCategoryItemLabelGenerator` as reported by *UMLDiff* constituted the contextual advice on how to accomplish the task.

8.2.2.4 Extracting classes

Complex classes are sometimes incohesive because they are responsible for delivering many responsibilities. Such classes should be simplified by extracting some of their features into other classes, created for exactly that purpose. The simplified class can then delegate to the newly created class to deliver its responsibilities.

For example, in version 0.9.14, a new class `RendererState` was created. The field `info` of type `PlotRendererInfo` and the method `getInfo()` were extracted from the `AbstractRenderer` to the `RendererState` class. A similar refactoring was also applied to `Axis` to extract a new `AxisState` class. Such state classes were designed to hold state information for renderer and

axis objects during the drawing process, which enable multiple threads to draw the same axis to different targets, since each drawing thread maintains its own separate state object.

The symptom motivating *extract class* refactoring is high class complexity, such as active classes that have been modified in at least, for example 60%, of the versions between two specific versions. In such cases, querying for the method sets used by the class clients may reveal subsets of methods used together which are candidates to become methods of a new extracted class.

8.2.2.5 Collapsing hierarchies

Collapsing hierarchies is another important refactoring that deals with generalization. Refactoring hierarchies often involves moving fields and methods or pulling them up into a newly added or an existing superclass, which, frequently, results in the classes that do little job or subclasses that are not that different from its superclass. In such cases, the (sub) classes should be merged to superclass.

For example, in version 0.9.9 JFreeChart was overhauled substantially. *UMLDiff* reported the largest number of changes to the system design between any two subsequent versions in the evolution history of JFreeChart. Several inheritance hierarchies were collapsed (see Section 6.4.4.4).

Actually, there exist three parallel inheritance hierarchies (Horizontal- and Vertical-plot, axis, and renderer) in the JFreeChart system before version 0.9.9. The horizontal class and its corresponding vertical one are very similar (or sometimes identical). The only major difference is that, one set was used for horizontal drawing, the other for vertical. Such parallel hierarchies make the subsequent changes difficult, since when it comes time to modify something, you have to change more than one place. This also results in a large amount of code duplication. The JFreeChart developers became, at some point, aware of the existence of separate horizontal and vertical hierarchies, and in release 0.9.9, they made a great effort to redesign the Plot, Axis, and Renderer hierarchies.

Parallel hierarchies, symptomatic of strong design interdependencies, can be discovered through the co-evolution analysis.

8.2.2.6 Replacing type code with typesafe-enum object

Numeric type codes are a common feature of procedural programming languages like C. Frequently, they are assigned as values to named constants to make them more readable. However, the compiler still sees the underlying number and it may alias it to any other number with no restrictions to its value range. In this case, there is nothing to force the named constants to be used; any arbitrary nonsense number can be passed in. A better alternative in object-oriented software is the typesafe-enum class [11]. The idea is to replace numeric type code with a class with private constructors; use factory methods to make sure only valid instances are created and passed around. One such typesafe-enum class has been discussed in Section 7.2.1. The same type refactorings were applied several times to produce such typesafe-enum classes as `HistogramType`, `RangeType`, `HorizontalAlignment` and `VerticalAlignment`, etc.

Monitoring the use of type code may depend on dynamic analysis, but a simple query returns all the constant fields of type `int` that are declared as `static` and `final` provides the developers a good start point to investigate the type code fields.

8.2.3 Contextual project-specific hints

In this section, we discuss some of JFreeChart specific design changes that could be valuable as contextual evolution knowledge, if properly recorded.

8.2.3.1 Splitting package

We investigated in detail the three major instances of *extract package* refactorings in JFreeChart. In version 0.9.4, the package `com.jrefinery.chart` contained 111 classes and interfaces. In version 0.9.5, 75 of them were moved into three new packages `com.jrefinery.chart.plot`, `com.jrefinery.chart.axis`, and `com.jrefinery.chart.renderer`. In version 0.9.7, 17 classes and interfaces were split out from package `com.jrefinery.chart.data` to a new package `com.jrefinery.chart.data.time`, but over 90 classes and interfaces were still left in it, until it was split again in version 0.9.21 to 8 new or existing packages. In version 0.9.21, 47 of 62 classes and interfaces were split out from `org.jfree.chart.renderer` to `org.jfree.chart.renderer.category` and `org.jfree.chart.renderer.xy` respectively.

These package-splitting activities generally reduced the number of classes and interfaces contained in each package to about 20-30. This project-specific behavior essentially constitutes a piece of contextual advice – although by no means definitive – that indeed, as a general rule:

- It is ok for a package to have ~30 classes and interfaces;
- A package should be split into subpackages when it has ~100 classes and interfaces;
- Increasing the size of a package over 60 can be flagged as a potential problem, because it reaches the range of complexity that makes it a splitting candidate.

8.2.3.2 More contextual advices

Other instances of JFreeChart-specific advices that we have recorded are listed as follows:

- Avoid introducing parallel horizontal and vertical Plot, Axis, or Renderer classes when working on these hierarchies, since a great effort was made in the past to eliminate them;
- If the constant fields are only used by a single class, do not separate them out;
- When adding new type of plot class, let it handle its own corresponding dataset. Avoid putting the dataset in the superclass Plot and let the subclasses do the downcasting;
- New dataset classes should implement interface Serializable and Cloneable;
- Two sets of methods should be provided in the dataset classes. One set for efficient access, the other for convenience.

8.3 Summary

In this chapter, we discussed our initial work on software design-mentoring, which could present developers with an advisable course of action, based on learned experiences from past evolution activities, whether mistakes or successes, especially at the design-level. It relies on *UMLDiff*-based refactoring detection and design-evolution analyses. It detects the opportunities for potential design improvement from the logical model of the system and its evolution history. Furthermore, it associates with these opportunities design changes aimed towards addressing similar problems that have been applied in the past of the system development, as reported by *UMLDiff* and the refactoring-detection queries.

We believe, by reflecting on these problematic patterns and their associated design remedies, a developer has a better chance to reach decisions on solving their current problems in a manner consistent with past ones.

Chapter 9: Conclusions, Contributions and Future Plan

Object-oriented software is increasingly developed using evolutionary development process model. Design is a continuous activity throughout the project lifecycle. Design evolves so that the system may support evolving features and is regularly refactored, as refactoring has become one of the most important core practices in the object-oriented software development. Although evolutionary development creates new software development challenges, such as shortened development cycles and increased frequency of software updates, it also represents new opportunities that, if suitably exploited, may provide supports to both existing and new development and maintenance activities.

To date, there has been no substantial support for reliably and accurately recognize design changes that software system has suffered through its evolution. The existing approaches either rely on low-level program representations that are not designed for understanding and supporting the design evolution of object-oriented software [3,20,45,47,82,107,118,128] or they require the high-quality consistently maintained change documentations [7,30,31,33,41,58,87]. Furthermore, although there has been a substantial amount of research in the general area of understanding the evolution, i.e., “past”, of software, there has been much less work on utilizing this understanding to “advising for the future”.

Continuous design improvement also poses serious problems for the asynchronous API evolution between reusable component frameworks and client application built on them. Unstable interfaces to a reusable component framework negatively impact reuse, since they require constantly client applications to adapt. The existing practices [15,42,50,119,121] rely on additional, and potentially substantial, information provided by the component-framework developers that document the changes and advise on how to adapt them. However, it is seldom the case that the change documentation and scripts provided with a large framework are sufficient for a client-application developer to effectively migrate to the changed APIs. All too often, application developers become lost when trying to reuse a changed API, unsure of how to make progress on a migration problem.

9.1 Contributions

This thesis presents a model-differencing based methodology for supporting object-oriented evolutionary development through capturing and analyzing the design-evolution history of object-oriented software. In this section, we summarize the contributions of this work.

9.1.1 Theoretical contributions

The major theoretical contributions of this work are the following:

- a) *Pair-wise model-differencing*: This methodology does not assume the existence of change documentation consistently maintained in the development process, nor does it rely on the comparison of low-level program representations, such as code lines, source code metrics, AST, control-flow graph, or XML. Instead, it relies on an original model-differencing algorithm, *UMLDiff*. *UMLDiff* compares UML logical models of an object-oriented software system and reports the design changes regarding additions, removals, moves, and renamings of subsystems, packages, classes, interfaces, attributes and operations, and changes to the attributes and relations of these model elements. As the applications developed based on its results demonstrate, its results are more directly relevant to the design-evolution of object-oriented software and correspond more closely to the intention of developers' changes. *UMLDiff* is sensitive to irregular usage of the versioning system, but with high precision and recall of design changes when the versioning system is used regularly. It is also robust to the user's choice of parameters, used to configure the differencing process.
- b) *Query-based refactoring detection*: Refactorings often result in many scattered low-level changes to the logical model of the system. One may still understand how the software system has been refactored by examining a set of small, elementary changes, such as those reported by *UMLDiff*, however, by combining the relevant elementary changes into refactorings, it becomes easier to understand the specific intent of the design-evolution and support refactoring-aware collaborative development. The instances of refactorings are recognized from their effects on the logical model, in

terms of queries of *UMLDiff* design-change facts. These queries are precise: they report the detected refactorings in terms of their particular types and participants. They are robust to “multiple-changes-to-same-entity” issue. They can be extended for detecting any structural change pattern of interest to the user. Query-based refactoring detection enables us to investigate how refactoring is practiced in general and elicit some high-level design requirement for a refactoring-aware development environment.

- c) *API-evolution Catch-up*: The Diff-CatchUp approach to adapting client applications in response to API changes of their underlying component frameworks does not require any additional work by the component-framework developers. Instead, it builds on our work on *UMLDiff* algorithm and refactoring detection. Once the specific API changes have been identified, not only does it formulate hypotheses for how the broken API might be replaced but also it collects specific examples of the hypothesized replacements have been used in order to provide the application developers with contextual information on the basis of which to evaluate its proposals. Furthermore, it does not focus on isolated changes but, instead, it aims to collect all API elements relevant to a particular migration problem. The Diff-CatchUp approach relies on the fact that a component framework itself represents good usage of its evolving API and thus is sensitive to the existence of “voluntary” migration example in its evolution history and the amount of changes they undergo. But its assumption holds for most cases and it is quite effective generating the replacing API elements and the corresponding usage examples in the face of the API evolution of a component framework.
- d) *Longitudinal design-evolution analyses*: The quantitative report of *UMLDiff* changes is discretized and classified to produce a qualitative record of the volatile nature of the design evolution of each individual class throughout the system’s history, i.e., phases of intensive evolution, rapidly developing, restructuring, slowly development, or steady-state. These distinct evolution phases allow us, through the application of sequential-pattern and association-rule mining methods, to study (a) how classes are introduced into, maintained, and eliminated from the system; (b) the evolution styles characteristic of the roles of classes in the system; (c) the relative-ordering relations

between function-extension and refactoring phases and whether they are consistent with the adopted development process; and finally, (d) the inter-dependencies among the evolution trajectories of different classes and their implications for the system's subsequent development. These analyses facilitate the overall understanding of system evolution, help to recognize system instabilities, and provide support regarding the scope of future maintenance activities.

- e) *Design mentoring*: Query-based refactoring-detection and longitudinal design-evolution analyses constitute a solid base for mentoring object-oriented evolutionary development, based on learned experiences from past evolution activities. Design mentoring is our attempt to bridge the gap between “understanding the past” and “advising for the future”. It detects the opportunities for potential design improvement from the logical model of the system and the results of longitudinal evolution analyses, guided by object-oriented design principles, design pattern and refactoring practices, or the development styles previously adopted by the system. It associates with these opportunities design changes aimed towards addressing similar problems, as reported by *UMLDiff* and the refactoring-detection queries that have been applied in the past of the system development. By reflecting on these problematic patterns and their associated design remedies, a developer has a better chance to reach decisions on solving their current problems in a manner consistent with past ones.

9.1.2 Software engineering tools

To support our theoretical work, we developed three relevant software engineering tools:

- a) *JDEvAn (Java Design Evolution and Analysis)* supports design-evolution analysis of Java software systems. It supports the reverse-engineering of UML models from Java source code; it provides a practically efficient implementation of *UMLDiff* algorithm and supports the inspection of *UMLDiff* results to correct erroneously identified and missed changes; it allows the developers to query a broad range of pre-defined “standard” refactorings, and also to define their own queries for any structural change patterns of their interests; it outputs discrete class-evolution profiles for third-party sequential-pattern analysis and data mining tools. *JDEvAn*'s front-end is an Eclipse

plugin. Its backend repository is a PostgreSQL relational database, which stores all the model facts, the change facts, and the analysis results.

- b) *Diff-CatchUp* tackles the problem of migrating client applications in the face of evolving APIs of component frameworks. The Diff-CatchUp front-end, an Eclipse plugin, allows the client-application developer to highlight the code fragments or compilation errors/warnings he wishes to update. In response, it identifies the model element of the component API involved in a selected migration problem and displays the replacement and usage-example proposals for further exploration with the JDEvAn Viewer. Its server hosts a JDEvAn repository regarding the logical-model and API-change facts of the evolving component framework, which is populated with the JDEvAn tool before Diff-CatchUp can be used. Diff-CatchUp searches the JDEvAn repository for the changes to the component APIs, the plausible replacements and their potential usage examples.
- c) The *JDEvAn Viewer* is the visualization component for JDEvAn and Diff-CatchUp. It enables an intuitive means of communicating all the design changes and evolution patterns produced by JDEvAn and the replacement and usage-example proposals generated by Diff-CatchUp. It provides software developers with a UML-style diagram and supports the interactive visualization, exploration, annotation, and persistence of JDEvAn and Diff-CatchUp output. JDEvAn Viewer has been implemented as an Eclipse plugin and it relies on Eclipse GEF (Graphical Editor Framework) [117]. It leverages the GEF facilities to provide Undo/Redo and Zoom-in/Zoom-out features.

We intentionally chose to implement our methodology on Eclipse, a popular Java development IDE, so that it can be tightly integrated within the development environment, and thus enable investigating the design-change patterns of object-oriented software evolution, exploring the underlying motivations behind them, and supporting future development and maintenance activities. These tools are publicly available to download. They have already been adopted by some users, who rely on them to investigate such topics as regression testing, model transformation, the detection of code smells, the correlation between the nature and size of the changes and the resulting bugs, the relationship between structural model and program refactoring, and so on.

9.1.3 Empirical case studies

We conducted three empirical case studies in order to refine and evaluate our methodology and the tools that implement it. These studies examined three independently developed software systems, which are of different size and complexity and from different application domains: HTMLUnit is a small-size open-source software system for unit testing; JFreeChart is a medium-size open-source Java class library for generating various types of charts; Eclipse is a large-scale industrial framework that has been under development for about five years. All of them have been actively developed for a long period of time and have suffered a substantial amount of design changes.

JFreeChart has been used as the subject system of extensive case study to evaluate all modules of our work, from *UMLDiff* algorithm to refactoring-detection queries, longitudinal design-evolution analyses, JDevAn Viewer, design mentoring, and finally Diff-CatchUp. HTMLUnit has been used in evaluation of the effectiveness of *UMLDiff* algorithm, refactoring-detection queries, JDevAn Viewer, and Diff-CatchUp approach. Eclipse has mainly been used to investigate the refactoring practice in the evolution of object-oriented software system and how it should be support in general.

These empirical studies have demonstrated that our approach is applicable and effective in practice. They are important for building confidence and trust in the whole methodology. Furthermore, all the analysis results are publicly available to download. We believe they will be useful to other researchers in the area.

9.2 Future Work

Our future work will focus on two essential aspects of modern software development practices, i.e., evolution and collaboration. First, we would like to apply the model-differencing based methodology to analyze and support the evolution of software systems developed in non-object-oriented development paradigms. Second, we want to develop techniques for supporting the collaborative production of evolving software systems. Third, we plan to develop methods for improving knowledge collaboration in the development of long-lived evolving software systems.

9.2.1 The software evolution in non-object-oriented paradigms

The core ideas of our research methodology to analyze and support evolutionary software development are to extract design models above code level, compute the differences between models, analyze these differences to identify interesting evolution patterns, and finally use the differences and evolution patterns to support further development. In this thesis, this methodology has been applied to study the evolution of static structure model of object-oriented software systems. However, we believe that this methodology is not restricted to object-oriented paradigm, nor is it restricted to static structure model.

We plan to apply the model-differencing based methodology to study the evolution of software systems developed in non-object-oriented programming paradigms. Assuming that the useful high-level models can be extracted, we believe that the combination of lexical and structure similarities should still apply to detect “meaningful” changes to the system design structure. However, the effectiveness of these similarity heuristics need to be thoroughly evaluated as the software development paradigms become more declarative and less structured.

We would also like to apply this methodology to analyze the evolution of dynamic models specifying the behavior and interaction of system processes or objects, such as message sequence chart [18,46], state transition model, or control and data flow diagram. New set of heuristics need to be developed to detect the “meaningful” changes to these dynamic models as they evolve over time. We believe that the change report of system design structure would inform the comparison of dynamic models. On the other hand, the evolution information of different structure and behavior models can be used to check their consistencies during system evolution. Furthermore, as different design models depict different aspects of a system design, the combination of their evolution information should enable a deeper understanding of the nature and impact of software evolution and a better support for further development.

9.2.2 The collaborative production of evolving software systems

Our Diff-CatchUp approach tackles the problem of asynchronous API evolution between the client applications and the component frameworks they reuse. In the future, we would like to develop more techniques for supporting the collaborative production of evolving

software system between design teams, development teams, and testing teams, such as merging parallel development branches, validating architecture/design compliance, and analyzing the nature and impact of software changes for selective regression testing.

Software systems are often subject to asynchronous and conflicting evolution from multiple parties. Today, there are two families of methods supporting the merging of conflict changes of parallel development branches: state based and operation based [65]. We would like to develop a merging method that combines the strengths of state- and operation-based merging, i.e., the easy adoption of state-based methods and the robust conflict detection and resolution of operation-based techniques. Similar to state-based methods, this merging method requires only a base version and its conflicting revisions to be merged. But it relies on a model-differencing algorithm, such as UMLDiff developed in this thesis, which is able to capture evolution operations applied to obtain each of the revisions from the base version, instead of low-level textual, AST, or dependency-graph differences between the based version and its revisions. And then it applies operation-based merging strategies, such as conflict table, to detect and resolve merge conflicts between these evolution operations. We also plan to apply this merging method to automatically effectuate the component's API changes in the context of client applications reusing the evolved components.

Software developers are often faced with questions throughout the course of software development regarding how well the implementation matches the system design, whether the evolved implementation is still compliant with the original design, and whether the intention of design evolution has been properly implemented. Static architecture/design compliance checking approaches can be used to address these questions. However, the existing approaches, such as rule-based compliance checking [71,77], software reflexion model [16,55,68], and software goal model [99], generally provide only an overview of high-level (e.g., subsystem-level) inconsistencies between design and code; they are rarely precise about which classes or methods cause the inconsistencies. We plan to develop model-differencing based method for the purpose of validating code-to-design compliance at fine-grained (e.g., method) level. The reported inconsistencies may reveal detailed information about unintended dependencies, misuse of patterns, breach of

architectural styles, and violation of evolution intentions in the system implementation. They may also be used to update the out-of-date design documents.

Selective regression testing relies on change impact analysis to identify tests that must be executed after software changes, to determine whether new tests need be created, and to prioritize the execution of test cases. The existing techniques [81] to selective regression testing attempt to improve the precision of change impact analysis by exploring different program representations, but little attention has been paid to the types of changes used in the analysis. We would like to raise the level of abstraction of concerned changes, as reported by model-differencing algorithms, when analyzing the nature and impact of software changes. Such change impact analysis could enable more reliable test selection, test prioritization, and test augment for selective regression testing. In addition, it may be used to estimate efforts to migrate client applications to the new versions of component frameworks. It may also be used to determine potential fault incidences in the client applications reusing the evolved components and identify changes responsible for these failures.

9.2.3 The knowledge collaboration in software evolution

As software evolves, it embodies a huge amount of useful information for future development, such as the quality attributes that are important to the system, the practices that the development team has adopted in the past, and the decisions that have been made in different situations, the persons that have expertise on a given task, and the working examples that the evolved APIs have been used. However, as the information become immense volumes, it is difficult to retrieve the information that developers want. Our research on the knowledge collaboration in the development of long-lived evolving software systems will focus on the sharing and transfer of software evolution knowledge among software developers. We believe that informed software developers are able to make intelligent decisions in their further development and maintenance activities.

We would like to develop methods for enhancing the awareness of each other's expertises, activities, and evolution decisions among software developers. As a starting point we will continue our work on software design mentoring. The goal is to develop a software design mentor that can detect anomalies in the system design structure and its

evolution history and offer suggestions for solving them by retrieving the past similar cases and their corresponding remedies. We believe that the analysis of software design structure and its evolution history, such as change-pattern detection and longitudinal design evolution analyses developed in this thesis, can uncover design changes aimed towards improving the system design and dependability and detect opportunities for such changes. The challenge is to develop an appropriate representation for recording and maintaining such design evolution knowledge so that a software design mentor can effectively reason about them and offer suggestions accordingly.

In addition to design structure and its evolution history, other types of information can be extracted from the development history of software systems, regarding who did what, how, and more importantly why such development occurred. Such information used to scatter in different sources, such as version control system, issue tracking system, and the developer's communication (e.g., emails, instant messages, or postings to newsgroups). The recent team collaboration products such as Jazz by IBM [120] and Visual Studio Team System by Microsoft [122] provide a common infrastructure that eases the management of software development assets across software lifecycle in collaborative software development. We plan to explore the information accumulated to such a team collaboration repository and apply information retrieval techniques such as Latent Semantic Analysis [19] to associate the relevant information with the results of model-differencing based design evolution analysis. The long-term goal of this research is to develop an "organization memory" that can provide quick and easy online access to software evolution knowledge rather than relying on human memory and experience.

Bibliography

1. A. Abbott and A. Hyrcak. Measuring resemblance in sequence data: An optimal matching analysis of musicians' careers. *American Journal of Sociology*, vol. 96, pp. 144-185, 1990.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Proceedings of the 20th International Conference on Very Large Databases*, pp. 487-499, September 1994.
3. T. Apiwattanapong, A. Orso and M.J. Harrold. A differencing algorithm for object-oriented programs. *Proceedings of the 19th International Conference on Automated Software Engineering*, pp. 2-13, 2004.
4. I. Balaban, F. Tip and R. Fuhrer. Refactoring support for class library migration. *Proceedings of the 20th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 265-279, 2005.
5. M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. *Proceedings of the 8th Working Conference on Reverse Engineering*, pp. 98-107, 2000.
6. D. Barnard, G. Clarke and N. Duncan. Tree-to-tree correction for document trees. *Technical report 95-375*, Queen's University, January 1995.
7. 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.
8. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
9. J. Bevan and E.J. Whitehead. Identification of software instabilities. *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 134-143, 2003.
10. A. Bianchi, D. Caivano, F. Lanubile and G. Visaggio. Evaluating software degradation through entropy. *Proceedings of the 11th International Software Metrics Symposium*, pp. 210-219, 2001.
11. J. Bloch. *Effective Java Programming Language Guide*. Addison Wesley, 2001.

12. J.E. Burge and D.C. Brown. Design rationale for software maintenance. *Proceedings of the 16th International Conference on Automated Software Engineering*, pp. 433, 2001.
13. E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pp 13-17, January 1990.
14. J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar and S. Midkiff. Escape analysis for Java. *Proceedings of the 14th ACM SIGPLAN Object Oriented Programming, Systems, Languages, and Applications*, pp. 1-19, 1999.
15. K. Chow and D. Notkin. Semi-automatic Update of Applications in Response to Library Changes. *Proceedings of the 12th International Conference of Software Maintenance*, pp. 359-368, 1996.
16. A. Christl, R. Koschke and M.A. Storey. Equipping the reflexion method with automated clustering. *Proceedings of the 13th Working Conference on Reverse Engineering*, pp. 89-98, 2005.
17. D. Cubranic and G.C. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. *Proceedings of the 25th International Conference on Software Engineering*, pp. 408-418, May 2003.
18. W. Damm and D. Harel. LSCs: Breathing life into message sequence chants. *Formal Methods in System Design*, 2001, 19:45-80.
19. S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 1990, 41:391-407.
20. S. Demeyer, S. Ducasse and O. Nierstrasz. Finding refactorings via change metrics. *Proceedings of the 15th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 166-177, 2000.
21. S. Demeyer. Maintainability versus performance: What's the effect of introducing polymorphism?, *Technical Report, Lab on Reengineering*, Universiteit Antwerpen, Belgium, 2002.
22. D. Dig and R. Johnson. The role of refactoring in API evolution. *Proceedings of the 21st International Conference on Software Maintenance*, pp. 389-398, 2005.

23. D. Dig, C. Comertoglu, D. Marinov and R. Johnson. Automatic detection of refactorings in evolving components. *Proceedings of European Conference on Object-Oriented Programming*, pp. 404-428, 2006.
24. D. Dig, K. Manzoor, R. Johnson and T. Nguyen. Refactoring-aware configuration management for object-oriented programs. *Proceedings of the 29th International Conference on Software Engineering*, pp. 427-436, 2007.
25. S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. *Proceedings of the 15th International Conference on Software Maintenance*, pp. 109–118, 1999.
26. A. Dutoit and B. Paech. Rationale management in software engineering. *Handbook on Software Engineering and Knowledge Engineering*, World Scientific, December 2001.
27. A. Egyed. Scalable consistency checking between diagrams - The VIEWINTEGRA approach. *Proceedings of the 16th International Conference on Automated Software Engineering*, 2001.
28. 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.
29. S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus and P. Schuster. Visualizing software changes. *Software Engineering*, 2002, 28(4):396-412.
30. 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.
31. 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.
32. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
33. H. Gall, K. Hajek and M. Jazayeri. Detection of logical coupling based on product release history. *Proceedings of the 14th International Conference on Software Maintenance*, pp. 190-198, November 1998.

34. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
35. 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.
36. M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 2005, 31(2):166-181.
37. C. Gorg and P. Weigerber. Detecting and visualizing refactorings from software archives. *Proceedings of the 13th International Workshop on Program Comprehension*, pp.205-214, 2005.
38. T.L. Graves and A. Mockus. Inferring change effort from configuration management databases. *Proceedings of the 5th International Symposium on Software Metrics*, pp. 267-273, 1998.
39. T.L. Graves, A.F. Karr, J.S. Marron and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 2000, 26(7):653-661.
40. J. Han and M. Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann, 2000.
41. A.E. Hassan and R.C. Holt. Studying the chaos of code development. *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 123-133, 2003.
42. J. Henkel and A. Diwan. CatchUp! Capturing and replaying refactorings to support API evolution. *Proceedings of the 27th International Conference on Software Engineering*, pp. 274-283, 2005.
43. M.E. Holmes and M.S. Poole. Longitudinal analysis. In S. Duck & B. Montgomery (Eds.), *Studying interpersonal interaction*, pp 286-302, 1991.
44. R. Holmes and G. Murphy. Using structural context to recommend source code examples. *Proceedings of the 27th International Conference on Software Engineering*, pp. 117-125, 2005.

45. S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 234-246, June 1990.
46. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC). 1999.
47. D. Jackson and D.A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. *Proceedings of the 9th International Conference on Software Maintenance*, pp. 243-252, September 1994.
48. 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.
49. R. Kazman and L. Bass. Toward deriving software architectures from quality attributes. *Technical Report CMU/SEI-94-TR-10*, August 1994.
50. C. Kemper and C. Overbeck. What's New With JBuilder. *2005 JavaOne Conference*.
51. J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.
52. M. Kim, D. Notkin and D. Grossman. Automatic inference of structural changes for matching across program versions. *Proceedings of the 29th International Conference on Software Engineering*, pp. 333-343, 2007.
53. S. Kim, K. Pan and J.E. James Whitehead. When functions change their names: Automatic detection of origin relationships. *Proceedings of the 12th Working Conference on Reverse Engineering*, pp. 143-152, 2005.
54. A.J. Ko, H.H. Aung and B.A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. *Proceedings of the 27th International Conference on Software Engineering*, pp. 126-135, 2005.
55. R. Koschke and D. Simon. Hierarchical reflexion models. *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 36-45, November 2003.
56. P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 1995, 12(6):42-50.
57. 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.

58. M.M. Lehman and L.A. Belady. Program evolution-processes of software change. *Academic Press*, London, 1985, 538pps.
59. R. Leitch and E. Stroulia. Assessing the maintainability benefits of design restructuring using dependency analysis. *Proceedings of the 9th International Symposium on Software Metrics*, pp.309, 2003.
60. K.J. Lieberherr and I. Holland. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices*, 1989, 24(3):67-78.
61. E. Lippe and N. van Oosterom. Operation-based merging. *Software Engineering Notes*, 17(5):78-87, 1992.
62. Y. Liu and E. Stroulia. Reverse engineering the process of small novice software teams. *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 102-112, 2003.
63. M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall, 1994.
64. D.W. McDonald and M.S. Acherman. Expertise Recommender: A Flexible Recommendation System and Architecture. *Proceedings of ACM Conference on Computer Supported Cooperative Work*, pp. 231-240, 2000.
65. T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 2002, 28(5):449-462.
66. T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation. 2002*, vol. 2505 of *Lecture Notes in Computer Science*, pp. 286–301, Springer-Verlag.
67. I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 235-250, 1996.
68. G.C. Murphy, D. Notkin and K.J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364-380.
69. *OMG Unified Modeling Language Specification*, formal/03-03-01, Version 1.5, (2003), <http://www.omg.org>.

70. D. Ohst, M. Welle and U. Kelter. Difference tools for analysis and design documents. *Proceedings of the 19th International Conference on Software Maintenance*, pp. 13-22, September 2003.
71. R. van Ommering, R. Krikhaar and L. Feijs. Languages for formalizing, visualizing, and verifying software architectures. *Computer Language*, 27(1):3-18, April 2001.
72. W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.
73. E. Ostertag, J. Hendler, R. Prieto-Daz and C. Braun. Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions of Software Engineering and Methodology*, 1992, 1(3):205-228.
74. 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.
75. D.C. Pelz. Innovation complexity and the sequence of innovating stages. *Knowledge: Creation, Diffusion, Utilization*, 1985(6):261-291.
76. J.H. Perkins. Automatically generating refactorings to support API evolution. *ACM SIGSOFT Software Engineering Notes*, 2006, 31(1):111-114.
77. A. Postma. A method for module architecture verification and its application on a large component-based system. *Information & Software Technology*, 2003, 45(4):171-194.
78. B.J. Rhodes and T. Starner. Remembrance Agent. *Proceedings of the First International Conference and Exhibition on the practical applications of intelligent agents and multi-agent technology*, pp. 487-495, 1996.
79. D. Roberts, J. Brant and R.E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object System* 1997, 3(4):253-263.
80. 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.
81. G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 1996, 22(8):529-551.

82. B.G. Ryder and F. Tip. Change impact analysis for object-oriented programs. *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 46-53, 2001.
83. F.V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. *Proceedings International Workshop on Principles of software Evolution*, pp. 126-130, September 2003.
84. R. Sabherwal and D. Robey. An empirical taxonomy of implementation processes based on sequences of events in information system development. *Organization Science*, vol. 4, pp. 548-576, 1993.
85. C. Schofield, B. Tansey, Z. Xing and E. Stroulia. Digging the development dust for refactorings. *Proceedings of the 14th International Conference on Program Comprehension*, pp. 23-34, June 2006.
86. P. Selonen, K. Koskimies and M. Sakkinen. Transformations between UML diagrams. *Journal of Database Management*, 2003, 14(3).
87. 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*, pp. 22-31, 2001.
88. V. Sinha, D. Karger, R. Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 187-194, 2006.
89. F. Simon, F. Steinbruckner and C. Lewerentz. Metrics based refactoring. *Proceedings of the 5th European Conference Software Maintenance and Reengineering*, pp. 30-38, 2001.
90. K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science 58, Automata, Languages and Programming*, pp. 376-386, 1986.
91. G. Spanoudakis and H. Kim. Reconciliation of object interaction models. *Proceedings of the 7th International Conference on Object Oriented Information Systems*, pp. 47-58, August 2001.
92. E. Stroulia and R. Kapoor. Metrics of refactoring-based development: An experience report. *Proceedings of the 7th International Conference on Object-Oriented Information Systems*, pp. 113-122, August 2001.

93. L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations, *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pp. 183–192, 2003.
94. E. Thomsen. *OLAP Solutions: Building Multidimensional Information Systems, Second Edition*. John Wiley & Sons, 2002.
95. F. Tip, A. Kiezun and D. Baumer. Refactoring for generalization using type constraints. *Proceedings of the 18th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 13-26, 2003.
96. T. Tourwe and T. Mens. Identifying refactoring opportunities using logic meta programming. *Proceedings of International Conference on Software Maintenance and Re-engineering*, pp. 91–100, 2003.
97. Q. Tu and M.W. Godfrey. An integrated approach for studying architectural evolution. *Proceedings of the 10th International Workshop on Program Comprehension*, pp. 127-136, 2002.
98. R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, vol. 21, no. 1, pp. 168-173, January 1974.
99. Y. Wang, S. McIlraith, Y. Yu and J. Mylopoulos. An automated approach to monitoring and diagnosing requirements. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 293-302, 2007.
100. P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. *Proceedings of the 21st International Conference on Automated Software Engineering*, pp. 231-240, 2006.
101. Z. Xing and E. Stroulia. Recognizing Refactoring from Change Tree, *Proceedings of REFACE (REFactoring: Achievements, Challenges, Effects) workshop in conjunction with the 10th Working Conference on Reverse Engineering*, November 13, 2003.
102. 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.

103. Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems' evolution. *Proceedings of the 20th International Conference on Software Maintenance*, pp. 242-251, 2004.
104. 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, November 2005.
105. 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*, pp. 458-468, September 2006.
106. Z.Xing and E.Stroulia. Refactoring detection based on UMLDiff change-facts queries. *Proceedings of the 13th Working Conference on Reverse Engineering*, pp. 263-274, October 2006.
107. W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 1991, 21(7):739-755.
108. Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. *Proceedings of the 24th International Conference on Software Engineering*, pp. 513-523, 2002.
109. A.M. Zaremski and J.M. Wing. Signature matching: A key to reuse. *Proceedings of 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 182-190, 1993.
110. T. Zimmermann, S. Diehl and A. Zeller. How history justifies system architecture (or not). *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pp. 73, 2003.
111. ActiveAspect: <http://www.cs.ubc.ca/labs/spl/projects/activeaspect/>, 2007.
112. ArgoUML: <http://argouml.tigris.org/>, 2007.
113. Bugzilla: <http://www.bugzilla.org>, 2007.
114. CMEclipse: <http://www.lucas.lth.se/cm/cmeclipse.shtml>, 2007
115. Comparing and merging UML models in IBM Rational Software Architect: http://www-128.ibm.com/developerworks/rational/library/05/712_comp/, 2007.
116. Eclipse: <http://www.eclipse.org>, 2007.
117. Eclipse Graphical Editing Framework: <http://www.eclipse.org/gef/>, 2007.

118. Diffutils: <http://www.gnu.org/software/diffutils/>, 2007.
119. Help – Eclipse SDK: <http://help.eclipse.org>, 2007.
120. IBM Jazz Project: <http://www.jazz.net>, 2007.
121. MSDN – Microsoft Visual C++: <http://msdn2.microsoft.com/en-us/visualc/aa336429.aspx>, 2007.
122. Microsoft Visual Studio Team System: <http://msdn.microsoft.com/teamssystem>, 2007.
123. HTMLUnit: <http://htmlunit.sourceforge.net/>, 2007.
124. IntelliJ IDEA: <http://www.jetbrains.com/idea/features/refactoring.html>, 2007.
125. JDevAn: <http://www.cs.ualberta.ca/~xing/jdevan.html>, 2007.
126. JFreeChart: <http://www.jfree.org/jfreechart>, 2007.
127. JCommon: <http://www.jfree.org/jcommon>, 2007.
128. Mosell EDM Ltd: <http://www.deltaxml.com>, 2007.
129. Refactoring example: <http://www.cs.unc.edu/~stotts/COMP204/refactor>, 2007.
130. Weka: <http://www.cs.waikato.ac.nz/~ml/weka>, 2007.

Appendix A: UML meta-model

Table A-1. The UML model elements

| Metaclass <<stereotype>> | Description |
|---|---|
| Subsystem | A subsystem is a grouping of model elements that represents a behavioural unit in a physical system. |
| Package | A package is a grouping of model elements. |
| Class | A class declares a collection of attributes, operations and methods that fully describe the structure and behavior of a set of objects. A class acts as the namespace for various kinds of contained elements defined within its scope, including classes and interfaces. |
| Interface | An interface is a named set of operations that characterize the behavior of an element. |
| DataType | A data type is a type whose values have no identity. |
| Attribute | An attribute is a named piece of the declared state of a classifier, which refers to a static feature of a model element. An attribute may have an <code>initValue</code> specifying the value of the attribute upon initialization. |
| Operation <<create>> <<initialize>> | An operation is a service that can be requested from an object to effect behavior, which refers to a dynamic feature of a model element. |
| Method | A method is the implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation. |
| Parameter | A parameter is a declaration of an argument to be passed to, or returned from an operation. |
| Exception | An exception is a signal raised by behavioral features typically in case of execution faults. |
| Reception | A reception is a behavioral feature and declares that the classifier containing the feature reacts to the signal designated by the reception feature. |

Table A-2. The UML relations among model elements

| Metaclass «stereotype» | Description |
|---|---|
| Generalization | A generalization is a taxonomic relation between a more general element (parent) and a more specific element (child). |
| Abstraction «realize» | An abstraction is a dependency relation that relates two elements or sets of elements that represent the same concept at different levels of abstraction. |
| Usage «call» «instantiate» «send» «read» «write» | A usage is a dependency relation in which one element requires another element (or set of elements) for its full implementation or operation. |
| Association | An association is a declaration of a semantic relation between classifiers that can be of three different kinds: 1) ordinary association, 2) composite aggregate, and 3) shareable aggregate. There are three meta-composition and five ordinary meta-associations defined in the meta-model, which are described in Table A-3. |

Table A-3. The compositions and associations among model elements

| Metarelation | Description |
|-----------------------------|--|
| namespace – ownedElement | A namespace is a model element that can own other model elements. The element ownership is used for unstructured contents such as the contents of a package or a class declared inside the scope of another class. |
| owner – feature | A classifier declares a collection of features. The features are the inherent semantic parts of a classifier. |
| BehaviorFeature – parameter | An operation declares an ordered list of parameters. The parameters are the inherent semantic parts of an operation. |
| typedParameter – type | Designates a classifier to which an argument value of a parameter must conform. The type must be a class, interface, or datatype. |
| typedFeature – type | Designates a classifier as whose instances are values of the attribute. The type must be a class, interface, or datatype. |
| context – raisedSignal | Designates exceptions that may be raised by behavioral features, such as operations when execution faults happen. |
| reception – signal | Designates reception features that handle the signal. |
| method – specification | Designates an operation that the method implements. |

Table A-4. UMLDiff-specific tagged values attached to model elements

| Tagged values | Base metaclass | Description |
|----------------------|-----------------------|--|
| comment | ModelElement | Any documentation attached to the model element. |
| isFromModel | ModelElement | If the model element is imported from a model other than the current one, false. Otherwise, true. |
| deprecated | ModelElement | If the model element is obsolete and will be removed from the model in the future, true. Otherwise, false. |
| overloaded | ModelElement | If the operation is overloaded, true. Otherwise, false. |
| count | Usage | The number of times a usage dependency appears between the client and supplier elements. |

Appendix B: Reverse-engineering Java Software

Table B-1. Mapping Java language constructs to UML model elements

| Java constructs | UML metaclasses |
|--------------------------------|---|
| Java primitive type | ProgrammingLanguageDataType |
| Java array type | ProgrammingLanguageDataType |
| Java software subsystem | Subsystem |
| Java package | Package |
| Java class | Class |
| Java interface | Interface |
| Java field | Attribute |
| Java method | Operation |
| Java constructor | Operation<<create>> |
| Java class initializer | Operation<<initialize>> |
| Java field initializer | Attribute's initValue |
| Java parameter | Parameter |
| The return type of Java method | Parameter whose name='return' and kind=return |

Table B-2. Mapping Java relations to UML metarelations

| Java relations | UML metarelations |
|------------------------------|--|
| Contain | meta-composition [namespace – ownedElement] |
| Declare | meta-composition [owner – Feature] |
| Method/constructor parameter | meta-composition [BehaviorFeature – parameter] |
| extends | Generalization |
| implements | Abstraction<<realize>> |
| new XXX(...) | Usage<<instantiate>> |
| Use field | Usage<<read>> |
| Change field value | Usage<<write>> |
| Method/constructor call | Usage<<call>> |
| throw statement | Usage<<send>> |
| Field data type | meta-association [typedFeature – type] |
| Parameter type | meta-association [typedParameter – type] |
| Method return type | meta-association [typedParameter – type] for the parameter whose kind=return |
| throws clause | meta-association [context – raisedSignal] |
| catch clause | meta-association [reception – signal] |

Table B-3. Mapping Java modifiers to the attributes of UML metaclasses

| Java modifiers | The attributes of UML metaclasses |
|----------------------------|--|
| public, protected, private | visibility of ElementOwnership or Feature |
| static | ownerScope=classifier of Feature |
| final | isLeaf=true of GeneralizableElement or Operation |
| synchronized | concurrency=guarded of Operation |
| abstract | isAbstract=true of GeneralizableElement or Operation |
| transient | persistence=transitory of Attribute |

Table B-4. Mapping Java language features to *UMLDiff*-specific tagged values

| Java language features | <i>UMLDiff</i>-specific tagged values |
|---|--|
| Javadoc description before block tags | comment |
| Java construct belongs in the source code | isFromModel=true |
| Javadoc contains @deprecate tag | deprecated=true |
| Several methods/constructors with same identifier | overloaded=true |

Appendix C: JFreeChart

Table C-1. The number of model element and relation facts of JFreeChart

| Model elements | | Relations | |
|-----------------------|---------------|---------------------|----------------|
| Package | 698 | Contain | 205298 |
| Class | 12866 | Extend | 13695 |
| Interface | 1686 | Implement | 9458 |
| Field | 40829 | Read | 154465 |
| Method | 101311 | Write | 61036 |
| Constructor | 17908 | Call | 416073 |
| Parameter | 142635 | Class usage | 165415 |
| | | Class instantiation | 94704 |
| Total | 317933 | Total | 1120144 |

Table C-2. The summary of *UMLDiff* changes²³ in JFreeChart evolution

| | |
|-------------------------------------|-------------|
| Element renaming | 2180 |
| Element move ²⁴ | 957 |
| Extract operation | 533 |
| Inline operation | 95 |
| Data (return) type change | 1056 |
| Abstraction<<realize>> change | 1032 |
| Generalization change | 186 |
| Visibility change | 868 |
| Other attribute/tagged-value change | 607 |
| Total | 7514 |

²³ The changes to usage dependency between model elements are not included. Same for HTMLUnit and Eclipse case study.

²⁴ The moved methods may also involve identifier changes. Such instances are manually added during the inspecting session of *UMLDiff* results. Same for HTMLUnit and Eclipse case study.

Appendix D: HTMLUnit

Table D-1. The number of model element and relation facts of HTMLUnit

| Model elements | | Relations | |
|-----------------------|--------------|---------------------|---------------|
| Package | 95 | Contain | 36717 |
| Class | 2639 | Extend | 2639 |
| Interface | 128 | Implement | 497 |
| Field | 3239 | Read | 10504 |
| Method | 23779 | Write | 3963 |
| Constructor | 2718 | Call | 74244 |
| Parameter | 11739 | Class usage | 17666 |
| | | Class instantiation | 11486 |
| Total | 44337 | Total | 157716 |

Table D-2. The summary of *UMLDiff* changes in HTMLUnit evolution

| | |
|-------------------------------|-------------|
| Element renaming | 464 |
| Element move | 1098 |
| Extract operation | 254 |
| Inline operation | 27 |
| Data (return) type change | 105 |
| Abstraction<<realize>> change | 43 |
| Generalization change | 135 |
| Visibility change | 79 |
| Total | 2205 |

Appendix E: Eclipse

Table E-1. The number of model element facts of Eclipse

| | 2.0 | 2.1 | 2.1.3 | 3.0 | 3.0.2 | 3.1 | Total |
|--------------|--------------|--------------|--------------|--------------|--------------|--------------|---------------|
| Package | 138 | 144 | 144 | 177 | 177 | 188 | 968 |
| Class | 3546 | 4326 | 4332 | 5610 | 5612 | 6466 | 29892 |
| Interface | 692 | 768 | 769 | 935 | 935 | 1024 | 5123 |
| Array Type | 562 | 294 | 296 | 383 | 383 | 439 | 2357 |
| Field | 11440 | 14213 | 14245 | 18812 | 18862 | 29029 | 106601 |
| Method | 27623 | 33829 | 33878 | 42923 | 42927 | 49187 | 230367 |
| Constructor | 3929 | 4737 | 4751 | 6025 | 6027 | 6943 | 32412 |
| Total | 47930 | 58311 | 58415 | 74865 | 74923 | 93276 | 407720 |

Table E-2. The number of relation facts of Eclipse

| | 2.0 | 2.1 | 2.1.3 | 3.0 | 3.0.2 | 3.1 | Total |
|---------------------|---------------|---------------|---------------|---------------|---------------|---------------|----------------|
| Contain | 53623 | 65925 | 66034 | 84963 | 85076 | 105162 | 460783 |
| Extend | 3253 | 4003 | 4009 | 5134 | 5135 | 5921 | 27455 |
| Implement | 1449 | 1790 | 1792 | 2298 | 2300 | 2596 | 12225 |
| Read | 44583 | 54842 | 54954 | 73754 | 73827 | 98120 | 400080 |
| Write | 17781 | 21597 | 21638 | 27755 | 27815 | 32648 | 149234 |
| Call | 90924 | 117813 | 117815 | 151629 | 151858 | 179775 | 809814 |
| Class usage | 31658 | 39284 | 39362 | 51700 | 51830 | 61594 | 275428 |
| Class instantiation | 9915 | 12273 | 12315 | 16025 | 16123 | 19037 | 85688 |
| Total | 253186 | 317527 | 317919 | 413258 | 413964 | 504853 | 2220707 |

Table E-3. The summary of UMLDiff changes in Eclipse evolution

| Type of change | 2.1 - 2.0 | 3.0 - 2.1.3 | 3.1-3.0.2 | Total |
|-------------------------------|--------------|--------------|--------------|--------------|
| Element renaming | 809 | 2285 | 1488 | 4582 |
| Element move | 387 | 1244 | 684 | 2315 |
| Visibility change | 435 | 857 | 550 | 1842 |
| Data (return) type change | 245 | 718 | 561 | 1524 |
| Non-access modifier change | 167 | 484 | 425 | 1076 |
| Abstraction<<realize>> change | 190 | 391 | 274 | 855 |
| Generalization change | 33 | 109 | 162 | 304 |
| Entity addition | 7127 | 14095 | 17343 | 38565 |
| Entity removal | 1298 | 4157 | 2455 | 7910 |
| Total | 10691 | 24340 | 23942 | 58973 |

Appendix F: Refactoring-Detection Queries

In this appendix, we review queries implemented in JDEvAn for detecting Fowler-catalog refactorings [32]

F.1 The simple “standard” refactorings

Each of the simple refactorings of Table 4-1 consists of a single elementary design change reported by *UMLDiff*. Some of the simple refactorings are the direct output of *UMLDiff*, while others take into account the information about the containment context of the refactored elements. Consider, for example, the *extract operation* elementary change: depending on whether the newly extracted method belongs in the same class as the original, or two classes along an inheritance path, or two unrelated classes, the change aims at refactoring the class internals, pulling up (pushing down) a behavior into a superclass (subclass), or moving behavior to a class where it naturally belongs.

F.1.1 Dealing with containment hierarchy

The refactorings of *move subsystem, package, and class* are discovered directly through the *UMLDiff* process.

Given a moved class or interface, if it was originally contained in a class or interface and has subsequently moved into a package, an instance of *convert inner type to top-level refactoring* is reported. The *convert top-level type to inner refactorings* are identified with a similar, but inverse, query. The participants of these three types of refactorings include the moved subsystems, packages, or classes/interfaces and their containing model elements in two compared versions respectively.

F.1.2 Dealing with generalization

The key elementary changes of the refactorings in this category are the moved methods, the moved fields, and the extracted/inlined operations. Given such an elementary change, the generalization and abstraction relationship between the declaring classes or interfaces of the source and target method/field is examined. If the declaring class or interface of the

target method/field is the supertype (direct or transitive) of that of the source method/field, an instance of *pull-up method*, *pull-up field*, or *pull-up behavior* refactoring is reported, depending on the type of key elementary change being inspected.

Push-down method, field and behavior refactorings are identified in a way similar to the corresponding pull-up refactorings, with the difference that the condition must be revised to read “the declaring class or interface of the target method/field is the subtype (direct or transitive) of that of the source method/field”. The *pull-up constructor body* refactoring is a special case of the *pull-up behavior* refactoring. Instances of this type of refactoring are recognized through a similar query to *pull-up behavior* with one more condition that checks that “both the source and target operations are constructors”. The participants of these five types of refactorings include the refactored methods, constructors or fields and their declaring classes or interfaces in the two compared versions respectively.

F.1.3 Moving features between objects

Move method, *move field* and *move behavior* refactorings are recognized similarly to the corresponding pull-up and push-down refactorings above. The only difference is that they require a different query condition that “the declaring classes or interfaces of the source and target method/field have no inheritance relation”. The refactoring participants involve the refactored methods, constructors or fields and their declaring classes or interfaces in the two compared versions respectively.

F.1.4 Refactoring class internals

The nine refactorings in this category are all recognized directly through the *UMLDiff* process. *Renaming* a model element changes its declared name (identifier for method/constructor). For methods and constructors, their parameter lists can be modified through *adding or removing parameter*. The visibility of a model element can be modified to a more restrictive one (*information hiding*). The declared types of fields, method, and parameters can change to a more general (*generalize type*) or more specific (*downcast type*) one.

If the declaring types of the extracted (inlined) operations and the operations from (to) which they are extracted (inlined) are same, then the *extract/inline operation* refactorings become the ***extract method*** and ***inline method*** refactorings described in Fowler's refactoring catalog [32], which are essentially used to refactor class internals.

F.2 The composite “standard” refactorings

Composite refactorings are recognized as the composition of two or more elementary changes reported by *UMLDiff* and/or the instances of simple refactorings discussed in Section F.1. Although one may still understand how the software system has been refactored by examining a set of small, primitive or simple changes, we believe that, by combining the relevant elementary changes and/or simple refactorings into composite refactorings, it becomes easier to understand the specific intent of the change.

F.2.1 Dealing with containment hierarchy

Extract subsystem and package refactorings redistribute groups of features into newly introduced subsystems or packages. These refactorings are detected by examining the *UMLDiff* status of the original and new containing (direct or transitive) subsystem or package of the moved model elements and/or the extracted/inlined operations. If the original containing subsystem or package of the source model element is mapped and the containing subsystem or package of the target model element is newly added, an instance of *extract subsystem* or *extract package* refactoring is reported. ***Inline subsystem and package*** refactorings merge the contents of one subsystem or package into another. They are recognized through a similar query to the corresponding *extract subsystem/package* refactoring, with the difference that the condition must be revised to read “the original containing subsystem or package of the source model element is removed and the containing subsystem or package of the target model element is mapped”. The refactoring participants include the refactored model elements and their corresponding containing (direct or transitive) subsystem or package in the two compared versions respectively.

F.2.2 Dealing with generalization

The *extract interface* refactoring can be used to bring out the common interface of several classes or to define the operations that can be requested from an object. Given a mapped class, which starts implementing a newly introduced interface, if the cardinality of the intersection set between the methods defined in the new interface and the methods with the same signatures declared in the mapped class is greater than the user-specified threshold, an instance of *extract interface* refactoring is reported.

Note that, for the sake of efficiency, only the most distinct effects of some composite refactorings on the logic model are examined. In the case of *extract interface*, it examines only the amount of the same-signature methods defined and declared in the new interface and the mapped class respectively. It does not look into such changes as parameter type change, usage dependency change, etc., in their corresponding user objects.

For an *extract interface* refactoring, there may exist more than one pair of same-signature methods defined and declared in the new interface and the mapped class respectively. Furthermore, there may exist more than one mapped class, which start implementing the given new interface. The refactoring participants include the mapped classes, the newly introduced interface, and the pairs of same-signature methods defined and declared in them respectively.

Extract superclass or subclass and *inline superclass or subclass* are four refactorings that result in the reorganization of the class hierarchy. Queries for the instances of these four types of refactorings are defined based on the results of *pull-up* and *push-down* refactorings. For example, *extract superclass* refactorings are identified as follows: given a *pull-up field, method, or behavior*, if its original declaring class or interface is mapped and its current declaring class or interface is newly added, an instance of *extract superclass* refactoring is reported. For an *extract superclass* refactoring, there may exist more than one instance of pull-ups between the original and new declaring classes. Furthermore, there may exist more than one subclass, whose features are pulled up into the newly added superclass. The refactoring participants include the mapped subclasses whose features are pulled up, the newly added superclass, and the pull-up methods, fields and/or behavior.

Inline superclass, *extract subclass*, and *inline subclass* refactorings are identified similarly. They are based on *push-downs*, *push-downs*, and *pull-ups* respectively. For the inline refactorings, the condition also needs to be revised to read “the original declaring class or interface is removed and the current declaring class or interface is mapped”. They have a similar set of refactoring participants with *extract superclass* refactorings.

The intent of ***form template method*** refactorings is to *pull-up* the sequence of operations in the superclass and allows the subclasses to behave differently through polymorphism. The query for recognizing this type of refactorings is defined based on the *pull-up method* and *extract method* refactorings: given an instance of *pull-up method*, if there exist one or more extracted method in the same subclass as the origin of the *pulled-up* method and these extracted methods override and/or implement the methods called by the pulled-up method that provide the default behavior in the superclass, an instance of *form template method* refactoring is reported. For a *form template method* refactoring, there may exist more than one pulled-up method (from different sibling subclasses) whose subclass-specific behaviors are extracted and left in the individual subclasses. The refactoring participants include the pulled-up methods that define the common sequence of algorithm, the methods called by the pulled-up method that provide the default behavior in the superclass, and the extracted methods in the individual subclasses that override the default behavior and define the subclass-specific features.

The ***replace inheritance with delegation*** refactoring aims at favoring object composition (black-box reuse) over class inheritance (white-box reuse). The intent of ***replace delegation with inheritance*** is exactly opposite. The former refactoring is identified as follows: given a mapped class, which no longer extends or implements a particular type but declares a new field, if the data type of the newly declared field is the supertype (direct or transitive) of the type that the class no longer extends or implements, an instance of *replace inheritance with delegation* refactoring is reported. The refactoring participants of *replace inheritance (delegation) with delegation (inheritance)* include the mapped class, their no longer (new) supertype, and the new (removed) field and its corresponding data type.

F.2.3 Moving features between objects

Extract class and inline class refactorings redistribute the features between objects and adjust the collaborations among them. We have defined die-hard and legacy classes [102102]. A *die-hard* class is a class that is removed from the system but most of its functionalities are moved to other classes. An inlined class is a die-hard class. A *legacy* class is just the opposite of a *die-hard* class: it is introduced into the system as a placeholder for fields, methods, and behavior moved in from other classes. An extracted class is a legacy class. *UMLDiff* does not attempt to identify entities that have been renamed and moved at the same time. However, moving a class or interface and renaming it at the same time often results in a pair of die-hard and legacy classes being identified.

The identification of these three refactorings relies on the *move method*, *field* and *behavior* refactorings. Consider the *extract class* as an example. If the declaring class of the source element is mapped and the declaring class of the target element is newly added, an instance of the *extract class* refactoring is reported. To recognize *inline class* refactorings, the condition becomes “If the declaring class of the source element is removed and the declaring class of the target element is mapped”. The condition to recognize pairs of *die-hard and legacy classes* is “If the declaring class of the source element is removed and the declaring class of the target element is newly added”.

For a refactoring of type *extract class*, *inline class* or a pair of *die-hard and legacy classes*, there may exist more than one instance of method, field and behavior move between the original and new declaring classes. The refactoring participants of these three types of refactorings are the refactored methods, constructors or fields, and their declaring classes or interfaces in the two compared versions respectively.

F.2.4 Refactoring class internals

Let us now discuss how instances of the introduce factory method refactoring are detected. Given a newly added static method, which instantiates a particular type of object and whose return type is the type or supertype (direct or transitive) of the objects it instantiates, if the client method no longer creates the object directly but delegates to the newly added static method, an instance of *introduce factory method* refactoring is

reported. The participants of this refactoring are the newly added static method, the types of the object it instantiates, its return type, and the corresponding client methods.

Note that the newly introduced factory methods may instantiate more than one type of objects, all of which are the subtype (direct or transitive) of its return type. The refactoring-detection query does not care in which class the factory method is declared. The factory method may be declared in the class of its return type. However, this may not always be the case. For example, the factory method may be declared in a separate factory class. The query for *introduce factory method* is able to handle such cases. However, it does not handle the object instantiation through dynamic loading, such as using `Class.forName(String)`. There is no way to recognize which types of objects are actually created based on only static analysis.

To detect ***encapsulate field*** refactorings the field, its new encapsulation method, and their client methods are examined. Given a newly added method, which reads (writes for setter method) the mapped field in the same class and whose return type (one of its parameter's type for setter method) is same as the data type of field it reads (writes), or they are related through inheritance, if the client method no longer reads (writes) the field directly but calls the newly added method to access (change) the value of the field, an instance of *encapsulate field* refactoring is reported.

However, note that the query for *encapsulate field* refactoring does not check the visibility of the field changing to the more restrictive one. The most distinct effect of this refactoring on design is the introduction of the new access method and the usage dependency changes of the field's client method. In our experience with several case studies, the developers do forget to hide the encapsulated field sometimes or forget to update all its client sites so that some of them may still refer to the field directly. This is also an indicator that an automatic refactoring engine that helps the developers perform the refactoring and update all the corresponding references is desirable.

Refactorings of type ***introduce parameter object*** are to encapsulate several parameters that often go together into a single object. They are detected by examining the renamed method or constructor and its removed and newly added parameters and their corresponding types. Given a renamed method/constructor, which no longer declares some of its original parameters but declares a new parameter whose type is a newly

added class or interface, if the cardinality of the intersection set between the no longer declared parameters and the fields of the newly added parameter type is greater than the user-specified threshold, an instance of the refactoring *introduce parameter object* is reported, with the renamed method and the newly added type of its new parameter as refactoring participants.

Preserve whole object refactorings result in sending a complete object as a parameter of a method call instead of passing some of its fields' values. Such refactorings are detected by examining the renamed method or constructor, its removed and newly added parameters and their corresponding types, and the usage dependency changes of the renamed method and its client method. Given a renamed method, which no longer declares some parameters but declares a new parameter whose type is a mapped class or interface, it starts calling some methods, which are no longer called by some of its client method. If the cardinality of the intersection set between the types of the parameters no longer declared by the renamed method and the return types of the methods the renamed method starts calling is greater than the user-specific threshold, an instance of the refactoring *preserve whole object* is reported, with the renamed method, its client method, and the type of its new parameter as the participants.

Appendix G: Refactoring Reports

Table G-1. The refactorings in the evolution of HTMLUnit, JFreeChart and Eclipse

| Category | Type of refactoring | HTMLUnit | JFreeChart | Eclipse |
|---------------------------------|-------------------------------------|----------|------------|------------------|
| Dealing with containment | Convert inner type to top-level | 1 | 1 | 19 |
| | Convert top-level to inner | 0 | 1 | 20 |
| | Extract subsystem | 0 | 0 | 3 ⁺²⁵ |
| | Inline subsystem | 0 | 0 | 0 ⁺ |
| | Extract package | 1 | 15 | 16 ⁺ |
| | Inline package | 4 | 1 | 3 ⁺ |
| Dealing with generalization | Pull-up method/field | 1008 | 315 | 279 |
| | Push-down method/field | 22 | 52 | 53 |
| | Pull-up behavior | 141 | 165 | 0 ⁺ |
| | Push-down behavior | 6 | 7 | 0 ⁺ |
| | Pull-up constructor body | 3 | 9 | 0 ⁺ |
| | Extract interface | 8 | 55 | 33 |
| | Extract superclass | 119 | 11 | 15 ⁺ |
| | Extract subclass | 1 | 3 | 4 ⁺ |
| | Inline superclass | 0 | 2 | 4 ⁺ |
| | Inline subclass | 2 | 23 | 7 ⁺ |
| | Form template method | 1 | 3 | 0 ⁺ |
| | Replace inheritance with delegation | 1 | 5 | 2 |
| Moving features between objects | Extract class | 13 | 34 | 95 ⁺ |
| | Inline class | 2 | 0 | 31 ⁺ |
| | Die-hard/legacy classes | 0 | 8 | 95 ⁺ |
| | Convert anonymous class to nested | 1 | 0 | 12 ⁺ |
| | Move method/field | 32 | 216 | 1363 |
| | Move behavior | 71 | 102 | 0 ⁺ |
| | Deprecation + delegation | 14 | 83 | 0 ⁺ |
| Refactoring class-internals | Information hiding | 7 | 707 | 751 |
| | Generalize type | 6 | 140 | 177 |
| | Downcast type | 10 | 54 | 85 |
| | Introduce factory method | 0 | 0 | 19 |
| | Introduce parameter object | 0 | 0 | 4 |
| | Encapsulate field | 9 | 182 | 0 ⁺ |
| | Preserve whole object | 0 | 4 | 0 ⁺ |

²⁵ For the time being, we did not compute the complete set of usage differences for Eclipse. As a result, the detection of *Extract/Inline Operation* refactorings was also disabled. The lack of the usage differences and *Extract/Inline Operation* results affect the detection of several types of refactorings, which are marked with “+”.

Appendix H: API Migration Problems

Table H-1. The summary of API migration problems that Diff-CatchUp is able to handle

| Problem description | Removal/Renaming/Move of element | | | Element attribute change | | | Element relationship change | | | |
|------------------------------------|----------------------------------|---------|--------|--------------------------|------------|-------------|-----------------------------|----------|--------|-------------|
| | Package | RefType | Method | Field | Visibility | Deprecation | Modifiers | Datatype | Throws | Inheritance |
| ImportNotfound | ✓ | ✓ | | | | | | | | |
| CannotImportPackage | ✓ | ✓ | | | | | | | | |
| UndefinedType | | ✓ | | | | | | | | |
| UndefinedName | | ✓ | | ✓ | | | | | | ✓ |
| InvalidClassInstantiation | | ✓ | | | | | | | | |
| SuperclassMustBeA Class | | ✓ | | | | | ✓ | | | |
| SuperInterfaceMustBeAnInterface | | ✓ | | | | | | | | |
| UndefinedMethod/Constructor | | | ✓ | | | | | | | ✓ |
| ParameterMismatch | | | ✓ | | | | | | | |
| UndefinedField | | | | ✓ | | | | | | ✓ |
| UsingDeprecatedType | | | | | | ✓ | | | | |
| NotVisibleType | | | | | ✓ | | | | | |
| UsingDeprecatedMethod/Constructor | | | | | | ✓ | | | | |
| OverridingDeprecatedMethod | | | | | | ✓ | | | | |
| NotVisibleMethod/Constructor | | | | | ✓ | | | | | |
| OverridingNon VisibleMethod | | | | | ✓ | | | | | |
| MethodReduces Visibility | | | | | ✓ | | | | | |
| UsingDeprecatedField | | | | | | ✓ | | | | |
| NotVisibleField | | | | | ✓ | | | | | |
| ClassExtendFinalClass | | | | | | | | ✓ | | |
| FinalMethodCannotBeOverridden | | | | | | | | ✓ | | |
| CannotOverrideStaticMtdWithInstMtd | | | | | | | | ✓ | | |
| CannotHideInstMtdWithStaticMtd | | | | | | | | ✓ | | |
| StaticMethodRequested | | | | | | | | ✓ | | |
| NonStaticFieldFromStaticInvocation | | | | | | | | ✓ | | |
| FinalFieldAssignment | | | | | | | | ✓ | | |

