QUALITY ASSURANCE OF SOFTWARE MODELS

A STRUCTURED QUALITY ASSURANCE PROCESS SUPPORTED BY A FLEXIBLE TOOL ENVIRONMENT IN THE ECLIPSE MODELING PROJECT

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DR. RER. NAT.

PHILIPPS-UNIVERSITY MARBURG, GERMANY
FB 12 – MATHEMATICS AND COMPUTER SCIENCE

AUTHOR:
DIPL.-INF. THORSTEN ARENDT
BORN OCTOBER 13, 1973 IN ZIEGENHAIN, GERMANY

MARBURG AN DER LAHN, 2014

Angefertigt mit Genehmigung des Fachbereichs Mathematik und Informatik der Philipps-Universität Marburg (Hochschulkennziffer 1180).

Gutachter:

Prof. Dr. Gabriele Taentzer, Philipps-Universität Marburg
Prof. Dr. Harald Störrle, Technical University of Denmark

Prüfungskommission:

Prof. Dr. Manfred Sommer, Philipps-Universität Marburg
Prof. Dr. Gabriele Taentzer, Philipps-Universität Marburg
Prof. Dr. Harald Störrle, Technical University of Denmark
Prof. Dr. Bernhard Seeger, Philipps-Universität Marburg

Einreichungstermin: 11. April 2014.

Prüfungstermin: 12. Juni 2014.

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

— C.A.R. Hoare
The 1980 ACM Turing Award Lecture

Für Nele und Swenja
in Liebe

ACKNOWLEDGEMENTS

I thank all who in one way or another contributed in the completion of this thesis.

First and foremost I offer my sincerest gratitude to my supervisor, Prof. Dr. Gabriele Taentzer, who has supported me throughout my thesis with her patience and knowledge whilst allowing me the room to work in my own way. Furthermore, I thank Prof. Dr. Harald Störrle who immediately agreed to co-supervise this thesis and who gave me a lot of valuable and encouraging comments.

Sincere thanks are given to the following friends and colleagues for proofreading and their valuable comments: Kristopher Born, Mischa Dieterle, Stefan Jurack, Timo Kehrer, Florian Mantz, Kyriakos Poyias, Daniel Strüber, and Steffen Vaupel. Furthermore, I thank Jan Baart, Matthias Burhenne, Gerrit H. Freise, Florian Mantz, Pawel Stepien, and Alexander Weber for their fantastic work on the tooling.

Most importantly, none of this would have been possible without the love and patience of my family. The endless love given by Swenja and Nele provided my inspiration and was my driving force. Also thanks to Elke and Horst for their love and never ending support.

Last but not least, I would like to thank Siemens Corporate Technology for partially funding the research in this thesis.

ABSTRACT

The paradigm of model-based software development (MBSD) has become more and more popular since it promises an increase in the efficiency and quality of software development. In this paradigm, software models play an increasingly important role and software quality and quality assurance consequently leads back to the quality and quality assurance of the involved models.

The fundamental aim of this thesis is the definition of a structured syntax-oriented process for quality assurance of software models that can be adapted to project-specific and domain-specific needs. It is structured into two sub-processes: a process for the specification of project-specific model quality assurance techniques, and a process for applying them on concrete software models within a MBSD project. The approach concentrates on quality aspects to be checked on the abstract model syntax and is based on quality assurance techniques model metrics, smells, and refactorings well-known from literature. So far, these techniques are mostly considered in isolation only and therefore the proposed process integrates them in order to perform model quality assurance more systematically. Three example cases performing the process serve as proof-of-concept implementations and show its applicability, its flexibility, and hence its usefulness.

Related to several issues concerning model quality assurance minor contributions of this thesis are (1) the definition of a quality model for model quality that consists of high-level quality attributes and low-level characteristics, (2) overviews on metrics, smells, and refactorings for UML class models including structured descriptions of each technique, and (3) an approach for composite model refactoring that concentrates on the specification of refactoring composition.

Since manually reviewing models is time consuming and error prone, several tasks of the proposed process should consequently be automated. As a further main contribution, this thesis presents a flexible tool environment for model quality assurance which is based on the Eclipse Modeling Framework (EMF), a common open source technology in model-based software development. The tool set is part of the Eclipse Modeling Project (EMP) and belongs to the Eclipse incubation project EMF Refactor which is available under the Eclipse public license (EPL). The EMF Refactor framework supports both the model designer and the model reviewer by obtaining metrics reports, by checking for potential model deficiencies (called model smells) and by systematically restructuring models using refactorings. The functionality of EMF Refactor is integrated into standard tree-based EMF instance editors, graphical GMF-based editors as used by Papyrus

UML, and textual editors provided by Xtext. Several experiments and studies show the suitability of the tools for supporting the techniques of the structured syntax-oriented model quality assurance process.

ZUSAMMENFASSUNG

Das Paradigma der modellbasierten Softwareentwicklung (MBSD) erfreut sich immer zunehmender Beliebtheit, da es eine Steigerung von Effizienz und Qualität in der Softwareentwicklung verspricht. Folgedessen spielen Softwaremodelle eine immer wichtigere Rolle und die Themen Qualität und Qualitätssicherung von Software werden somit zurückgeführt auf die Themen Qualität und Qualitätssicherung der beteiligten Modelle.

Der grundlegende Inhalt dieser Arbeit ist die Definition eines strukturierten, syntaxorientierten und an projektspezifische bzw. domänenspezifische Bedürfnisse anpassbaren Prozesses für die Qualitätssicherung von Softwaremodellen. Dieser Prozess besteht aus zwei Teilprozessen. Im ersten Prozess werden projektspezifische Techniken für die Qualitätssicherung spezifiziert, die anschließend mit Hilfe des zweiten Prozesses an konkreten Softwaremodellen während eines MBSD Projektes angewendet werden können. Der Ansatz konzentriert sich dabei auf diejenigen Qualitätsaspekte, die auf der abstrakten Syntax des Modells überprüft werden können und benutzt die aus der Forschungsliteratur bekannten Qualitätssicherungstechniken Modellmetriken, Smells und Refactorings, die bis dato jedoch nur separat betrachtet wurden. Der vorgeschlagene Prozess integriert jetzt diese Techniken auf strukturierte Weise und ermöglicht so eine systematische Qualitätssicherung von Softwaremodellen. Drei ausgesuchte Beispiele mit unterschiedlichen Modellierungssprachen dienen als Proof-of-Concept Implementierungen des Prozesses und zeigen die Eignung, die Flexibilität und somit die Zweckmäßigkeit des Ansatzes.

Im Zusammenhang mit der Thematik Qualitätssicherung von Softwaremodellen beinhaltet die Arbeit zudem die folgenden zusätzlichen Beiträge: (1) die Definition eines Qualitätsmodells für Modellqualität, (2) Übersichten über Metriken, Smells und Refactorings für UML-Klassenmodelle inklusive strukturierter Beschreibungen dieser Techniken sowie (3) einen konzeptionellen Ansatz für die Spezifikation von komponierten Modell-Refactorings.

Der hohe Zeitaufwand und die potentielle Fehleranfälligkeit von manuell durchgeführten Modellanalysen erfordern eine weitgehende Automatisierung verschiedener Aktivitäten des vorgeschlagenen Qualitätssicherungsprozesses. Ein weiterer Hauptbeitrag dieser Arbeit ist die Entwicklung einer flexiblen Werkzeugumgebung für die Qualitätssicherung von Modellen, die auf dem Eclipse Modeling Framework (EMF), einer weit verbreiteten open-source Technologie im Bereich der modellbasierten Softwareentwicklung, basieren. Die Werkzeuge sind Teil des Eclipse Modeling Project (EMP) und gehören zum of-

fiziellen Inkubations-Projekt EMF Refactor, das unter der Eclipse Public License (EPL) zur Verfügung gestellt wird. Das Framework unterstützt Modellierer und Analysten bei der Erstellung von Metrikenberichten, dem Auffinden sogenannter Model Smells sowie der systematischen Restrukturierung der Modelle durch Refactorings. Die Funktionalität von EMF Refactor ist dabei in die baumbasierten EMF Instanzeditoren, in die auf GMF basierenden grafischen Editoren und in die von Xtext bereitgestellten textuellen Modelleditoren integriert. Verschiedene Experimente und Studien zeigen die Zweckmäßigkeit und die Eignung der Werkzeuge für die Unterstützung der Techniken in dem zuvor beschriebenen syntaxorientierten Qualitätssicherungsprozess für Softwaremodelle.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

Achieving high quality is one of today's challenges in product development processes. This is especially true for processes that are concerned with the development of software being intended to make people's lives easier. In modern software development processes, models become primary artifacts. Consequently, model quality assurance is of increasing importance for the development of high quality software. This leads to a number of general research questions respectively problem statements that should be addressed in this thesis:

- *What is model quality?* So far, there is no clear notion of model quality. Existing software quality standards are only partly applicable.
- *How can model quality be assured in a given project?* An adaptable process for performing quality assurance in a structured way is needed. This also includes the use of adequate tools in a flexible and integrated manner.
- *Which model quality assurance techniques exist and how do they correlate?* Correlation means both, relations within the same technique (such as combinations) and relations between different techniques (like implications).

In this first chapter, we motivate and discuss the goals of the work. Then, we summarize the main results as well as the author's publications being relevant to this thesis. Finally, we give an overview of the structure of the thesis and provide recommendations on how to read its constituent parts.

1.1 MOTIVATION AND GOALS

In the paradigm of model-based software development (MBSD), models play an increasingly important role and become primary artifacts in the software development process. In particular, this is true for model-driven software development (MDD) where high code quality can be reached only if the quality of input models is already high. As a consequence, software quality and quality assurance frequently

leads back to the quality and quality assurance of the involved software models.

Models are used for different purposes, e.g., specifying the software architecture and design, as input for code implementation or test case generation, or simply for communication purposes between several stakeholders within the project. Furthermore, model-based software projects are often part of the development of safety-critical embedded systems such as medical systems. In these cases, also safety aspects have to be addressed. The variety of scenarios demonstrates that the modeling purpose must be considered when selecting the corresponding model quality aspects of interest.

In the literature, well-known quality assurance techniques for models are model metrics and refactorings. They originate from corresponding techniques for software code by lifting them to models. Furthermore, the concept of code smells can be lifted to models, leading to model smells. However, these techniques are considered in isolation only, i.e., they address specific scenarios only without taking further techniques into account in order to provide a more global view on the quality of the model. Therefore, ***an integrated approach is needed in order to perform model quality assurance systematically***.

To evaluate the quality of a software product, the use of a well-defined quality model representing the characteristics of the product that describe the quality has been established for more than three decades. However, there is no clear notion of model quality in the literature and software quality standards like ISO/IEC 9126 and 25010 are only partly applicable. They are intended for complete software products and systems and not for development artifacts like software models. Several quality characteristics (e.g., *reliability*, *efficiency*, and *security*) that have a significant relevance on the quality of software products can not be considered when reasoning about model quality. As a consequence, ***there is still a lack of understanding in terms of what model quality exactly means***.

A widely accepted standard in software modeling is the Unified Modeling Language (UML). It provides 14 types of diagrams for both, structural and behavioral models. Here, class diagrams are the mostly used UML diagram type. Since its adoption in 1997, metrics, smells, and refactorings for UML models are in the scope of a variety of researchers. However, ***except for UML metrics, there are no structured surveys on these model quality assurance techniques available***.

Existing approaches for specifying model refactorings differ heavily in the way refactorings are specified. They mainly focus on smaller model changes, i.e., larger model refactorings are rarely considered. However, atomic refactorings are not always performed in isolation. Often, they are part of a group of refactorings that are all needed to perform a larger change. Drawing from the experience of code refactoring, it was soon clear that refactorings should be distinguished

into atomic ones performing primitive changes and composite refactorings that are built up from existing ones. Despite the multitude of model refactoring approaches, ***the specification of composite model refactorings is not yet sufficiently supported by existing approaches in the sense that composite refactorings are consequently built up from existing ones being developed independently***.

Software modeling is mainly performed using CASE tools such as MagicDraw or IBM Rational Software architect. Moreover, manually performing quality assurance tasks would be time-consuming and error-prone. Therefore, it is natural to ***support model quality assurance as effectively as possible by an appropriate tooling***. The tooling should be integrated into the used CASE tool in a way that all model quality assurance tasks can be performed directly within this IDE. This means, that (1) several kinds of editors such as graphical and textual editors are supported and (2) the user does not have to export the model and use third-party tools, for example for analyzing it. Finally, the tooling should be independent from both, the considered modeling language and the language used for specifying new quality assurance techniques.

The following section summarizes the main contributions of this thesis according to the goals and challenges discussed above.

1.2 CONTRIBUTIONS

The contributions of this thesis can be subdivided into two categories. On the one hand, the thesis provides several conceptual respectively theoretical results related to the integration of model metrics, smells, and refactoring into a well-defined model quality assurance process. On the other hand, the second category contains results concerning implementation and tooling issues. The following sections summarize these contributions.

1.2.1 *Conceptual results*

A major contribution of this thesis is the

```
definition of an approach for the integration of model
metrics, model smells, and model refactoring into
a structured quality assurance process for software
models that considers project-specific needs.
```

This syntax-oriented process consists of two sub-processes: First, dependent on the modeling language and the modeling purpose, specific quality goals, and hence project- and domain-specific quality checks and refactorings have to be defined. Quality checks are formulated using model smells which can be specified in terms of model

metrics and anti-patterns. Afterwards, the specified quality assurance process is applied to concrete software models. Based on the outcome of a static model analysis using the pre-defined model metrics and smells, appropriate model refactoring steps can be performed. The techniques should be applied as long as needed in order to obtain a reasonable model quality. Three scenarios for performing this model quality assurance process serve as proof-of-concept implementations and show its applicability, its flexibility, and hence its usefulness.

In our approach, we concentrate on quality aspects to be checked on the model syntax. These include not only the consistency with the language syntax definition, but also the conformity with modeling conventions often defined and adapted to specific software projects. As a conceptual basis for a Goal-Question-Metrics approach to our quality assurance process, we refer to six classes of quality goals for software models identified in a systematic literature review. Based on these so-called 6C goals we present the

```
definition of a quality model for model quality
```

consisting of high-level quality attributes and low-level characteristics. This model represents a further contribution of this thesis.

Since the UML is a widely accepted standard in software modeling and subject of a number of research activities, this thesis further provides an

```
overview on metrics, smells, and refactorings for UML
class models discussed in the literature, including
structured descriptions of each technique.
```

Besides the discussion on the various relations to the defined quality model, we also discuss relationships between selected UML refactorings and UML smells. Due to a pragmatic search strategy, we do not claim the surveys to be complete. However, they are quite comprehensive and represent another contribution of this thesis.

As a further contribution of this thesis, we present an

```
approach for composite model refactoring addressing
the specification of refactoring composition.
```

The main idea of the approach is to specify composite model refactorings by a hierarchy of so-called refactoring units defining some kind of control structure of a composite and with parameter passing between different units.

1.2.2 *Implementation and tooling*

A common and widely-used open source technology in model-based software development is the Eclipse Modeling Framework (EMF). It extends Eclipse by modeling facilities and allows for defining (meta) models and modeling languages by means of structured data models. Furthermore, EMF comes with a very active community providing a variety of helpful tools. Also due to the comprehensive knowledge in this domain, another major contribution of this thesis is the

```
development of a flexible framework for model
quality assurance based on the Eclipse Modeling
Framework (EMF).
```

The framework has been designed to support a syntax-oriented model quality assurance process that can be easily adapted to specific needs in model-based projects (see major conceptual result above). The entire tool set presented belongs to the Eclipse incubation project *EMF Refactor* [47] and is available under the Eclipse public license. We evaluated the suitability of the tools for supporting the techniques of the model quality assurance process by performing and analyzing several experiments and studies.

EMF Refactor supports both the modeler and the reviewer by generating metrics reports, checking for potential model deficiencies respectively smells, and systematically restructuring models using refactorings. Quick fixes such as automatic proposition of refactoring for occurring smells and information on implications of a selected refactoring concerning new model smells widen the provided functionality and support an integrated use of the quality assurance tools.

The main functionality of EMF Refactor is integrated into several editors. Here, not only standard tree-based EMF instance editors are supported, but also graphical GMF-based editors as used by Papyrus UML and textual editors provided by Xtext. Among other functionalities, each version provides a highlighting of model elements for smells in the corresponding model view and a preview of upcoming model changes when performing a refactoring.

Model checks and refactorings can be specified by several specification mechanisms. The current version of EMF Refactor supports Java, OCL, and the model transformation language Henshin as possible specification approaches. Further specification languages can be inserted using suitable adapters. Finally, metrics can be composed to more complex metrics and refactorings can be composed by using a dedicated language named CoMReL (Composite Model Refactoring Language) based on the fourth conceptual contribution (see above).

The following papers and articles related to this thesis were published during the doctoral project of the author (in chronological order).

1. Thorsten Arendt, Florian Mantz, Lars Schneider, and Gabriele Taentzer: *Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study.* Proceedings of Model-Driven Software Evolution, Workshop Models and Evolution (MoDSE-MCCM 2009), co-located with MoDELS 2009, October 4 2009 in Denver, CO, USA.
   $\Longrightarrow$ Section 10.3 is an adapted version of this paper.

2. Thorsten Arendt, Pawel Stepien, and Gabriele Taentzer: *EMF Metrics: Specification and Calculation of Model Metrics within the Eclipse Modeling Framework.* Proceedings of 9th BElgian- NEtherlands software eVOLution seminar (BENEVOL 2010), December 17 2010 in Lille, France.
   $\Longrightarrow$ Sections 12.2 and 13.2 are based on this paper.

3. Thorsten Arendt, Matthias Burhenne, and Gabriele Taentzer: *Defining and Checking Model Smells: A Quality Assurance Task for Models based on the Eclipse Modeling Framework.* Proceedings of 9th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010), December 17 2010 in Lille, France.
   $\Longrightarrow$ Sections 12.3 and 13.3 are based on this paper.

4. Thorsten Arendt, Florian Mantz, and Gabriele Taentzer: *EMF Refactor: Specification and Application of Model Refactorings within the Eclipse Modeling Framework.* Proceedings of 9th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010), December 17 2010 in Lille, France.
   $\Longrightarrow$ Sections 12.4 and 13.4 are based on this paper.

5. Thorsten Arendt, Sieglinde Kranz, Florian Mantz, Nikolaus Regnat, and Gabriele Taentzer: *Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support*. Proceedings of Software Engineering 2011, February 21-25 2011 in Karlsruhe, Germany. Volume 183 of LNI, pages 63-74, GI, 2011.
   $\Longrightarrow$ Chapters 3 and 12 are adapted versions of this paper.

6. Thorsten Arendt and Gabriele Taentzer: *Integration of Smells and Refactorings within the Eclipse Modeling Framework.* Proceedings of Fifth Workshop on Refactoring Tools (WRT 2012) co-located with ICSE 2012, June 1 2012 in Rapperswil, Switzerland.
   $\Longrightarrow$ Sections 12.4 and 13.5 use parts of this paper.

7. Thorsten Arendt and Gabriele Taentzer: *Besser modellieren: Qualitätssicherung von UML-Modellen.* Article in magazine Objektspektrum, 06 2012, SIGS DATACOM.
   $\Longrightarrow$ Chapters 4 and 12 are elaborated versions of this article.

8. Thorsten Arendt and Gabriele Taentzer: *Composite Refactorings for EMF Models.* Technical report, Philipps-Universität Marburg, FB 12 - Mathematik und Informatik, Marburg, Germany, 2012.
   $\Longrightarrow$ Chapter 7 is an adapted versions of this report.

9. Gabriele Taentzer, Thorsten Arendt, Claudia Ermel and Reiko Heckel: *Towards refactoring of rule-based, in-place model transformation systems*. Proceedings of the First Workshop on the Analysis of Model Transformations (AMT) co-located with MoDELS 2012, October 2 2012 in Innsbruck, Austria.
   $\Longrightarrow$ Section 6.3 is an adapted version of this paper.

10. Thorsten Arendt and Gabriele Taentzer: *A tool environment for quality assurance based on the Eclipse Modeling Framework.* Journal Automated Software Engineering, Volume 20, Issue 2 (2013), Page 141-184, Springer, New York.
    $\Longrightarrow$ This article is a condensed version of Sections 3.2, 4.2, and 6.1, as well as several sections of Part II.

11. Thorsten Arendt, Gabriele Taentzer and Alexander Weber: *Quality Assurance of Textual Models within Eclipse using OCL and Model Transformations.* Proceedings of 13th International Workshop on OCL, Model Constraint and Query Languages (OCL) co-located with MoDELS 2013, September 30 2013 in Miami, CA, USA.
    $\Longrightarrow$ Section 6.2 and Chapter 13 are extended versions of this paper.

Furthermore, the author presented the EMF Refactor at the following conference events:

1. Thorsten Arendt and Gabriele Taentzer: *Improving the Quality of EMF models using metrics, smells, and refactorings.* Tutorial at 8th European Conference on Modelling Foundations and Applications (ECMFA 2012), July 2 2012 in Lyngby, Denmark.
   $\Longrightarrow$ Section 6.1 and Chapter 12 are extended versions of this tutorial.

2. Thorsten Arendt: *Improve the Quality of your EMF-based Models!* Talk at EclipseCon Europe 2012, October 22 2012 in Ludwigsburg, Germany.
   $\Longrightarrow$ Chapter 12 is an elaborated version of this talk.

3. Thorsten Arendt, Timo Kehrer and Gabriele Taentzer: *Understanding Complex Changes and Improving the Quality of UML and Domain-Specific Models.* Tutorial at ACM/IEEE 16th International

Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), September 30 2013 in Miami, CA, USA.
$\Longrightarrow$ Sections 6.1 and 6.2 as well as Chapters 12 and 13 are extended versions of this tutorial.

Finally, Chapter 5 is a results of three tasks within the project *SPES 2020 Software Platform Embedded Systems* [2, 88] funded by the German Federal Ministry of Education and Research from 2009 to 2012.

1.4 How to Read This Thesis

This thesis is subdivided into two main parts. Part I comprises the conceptual contributions presented in Section 1.2.1. Here, the structured process for quality assurance of software models that can be adapted to project-specific and domain-specific needs is presented. Moreover, several topics related to this process and corresponding results are included in this part. Part II presents the second main contribution of this thesis, i.e., the provided tool environment for model quality assurance in Eclipse (see Section 1.2.2). This part includes topics related to development and evaluation of the tooling as well as examples for applying and specifying new model quality assurance techniques. A brief description of each section can be found in the introductory chapter of the corresponding part. Finally, this thesis contains several appendices containing comprehensive catalogs with structured descriptions of metrics, smells, and refactorings for UML class models as well as study material concerning the evaluation.

Though this thesis can be read in chronological order, different kinds of readers may read only parts of it without losing the overall context. However, each reader should start reading Chapter 3 since the defined process represents the foundation for the subsequent chapters. Afterwards, the reader may continue as follows:

Project managers respectively ***quality assurance managers*** may continue reading Chapter 4 and (if interested in examples) Chapters 5 and 6. Furthermore, the practical Chapters 12 and 14 may be of interest to this kind of readers.

(Modeling) language designers may continue reading Chapter 4 and the examples in Chapter 5. In particular, the specification Chapters 7 and 13 are of specific interest to this kind of readers.

(UML) modelers may continue reading Chapter 4 and the UML examples in Chapters 5 and 6. Moreover, the practical Chapters 12, 13, and 14 may be of interest to this kind of readers.

Finally, ***Eclipse developers*** respectively ***EMF developers*** may continue reading the practical chapters of Part II in arbitrary order.

Part I

A STRUCTURED QUALITY ASSURANCE PROCESS FOR SOFTWARE MODELS

2

INTRODUCTION TO PART I

The paradigm of model-based software development (MBSD) has become more and more popular since it promises an increase in the efficiency and quality of software development. In this paradigm, models play an increasingly important role and become primary artifacts in the software development process. In particular, this is true for model-driven software development (MDD) where models are used directly for automatic code generation. Here, high code quality can be reached only if the quality of input models is already high. Therefore, software quality and quality assurance frequently leads back to the quality and quality assurance of the involved models.

Developers use models for different purposes, e.g., for specifying the software architecture and design, as input for code implementation or retrieving information for tests and test case generation. Often these developers are not on one site, i.e., architects may be located in Germany and the implementers in India. In these cases, models are an essential part of developer communication and their quality influences the quality of the final product to a great extent. In addition, model-based software projects are often part of the development of safety critical systems where safety-related aspects need to be addressed. For example, the safety standard IEC 62304 [39] requires that for a medical system all intermediate results during the development process including software models must be of an appropriate quality.

A widely accepted standard in software modeling is the Unified Modeling Language (UML) [123], a general-purpose language managed by the Object Management Group (OMG). It is a very comprehensive and powerful language, but does not cover any particular method and comes without built-in semantics. On the one hand, this allows a flexible use. On the other hand, this includes a high risk of misunderstanding. Without tailoring a project-specific usage of UML before starting development, practical experience showed that models can be difficult to understand or even misinterpreted.

In the literature, well-known quality assurance techniques for models are model metrics and refactorings. They origin from corresponding techniques for software code by lifting them to models. Especially class models are closely related to class structures in object-oriented programming languages such as C++ and Java. For behavior models,

the relation between models and code is less obvious. Finally, the concept of code smells can be lifted to models, leading to model smells.

However, these techniques are often considered in isolation, i.e., they address specific scenarios only without taking further techniques into account in order to provide a more global view on the quality of the model. Therefore, an integrated approach is needed in order to perform model quality assurance systematically. To satisfy this need, this first part of the thesis presents a **`structured process for quality assurance of software models`** that can be adapted to both project-specific and domain-specific needs. Moreover, we discuss several topics related to this process and present corresponding results.

The chapters of Part I contain the following:

Chapter 3 gives an overview on model quality assurance and model quality assurance techniques and defines two processes for model quality assurance: a process for the application of project-specific model quality assurance techniques in ongoing projects and a process to specify these project-specific techniques in a structured way.

Chapter 4 defines a quality model for model quality consisting of high-level quality attributes and low-level characteristics based on a discussion on quality models for software products in general, their adaptation to models, and comprehensive definitions of quality aspects and characteristics extracted from selected review articles.

Chapter 5 presents an (incomplete but comprehensive) overview on metrics, smells, and refactorings for UML class models discussed in literature. For each technique, we cite the main sources, present structured descriptions of selected examples, and discuss relations to the quality model discussed in Chapter 4.

Chapter 6 demonstrates three example applications serving as proof-of-concept evaluation of the quality assurance process defined in Chapter 3. The example cases include UML as general-purpose language, a domain-specific language for developing simple web applications, and a DSL for rule-based model transformation systems.

Chapter 7 discusses an approach for the specification of refactoring composition. It is motivated by the fact that the specification of composite model refactorings is not yet sufficiently supported by existing approaches, in the sense that composite refactorings are consequently built up from existing ones being developed independently.

Finally, ***Chapter 8*** concludes and discusses directions for future work on systematic model quality assurance.

3

A STRUCTURED MODEL QUALITY ASSURANCE PROCESS

Achieving high quality is one of the major challenges on today's product development processes. This is especially true for those processes that are concerned with the development of software being intended to make people's lives easier. According to Sommerville [143], software quality management aims at managing the quality of software and its development process. It can be subdivided into three main tasks: assuring, planning, and controlling of software quality.

This chapter deals with the definition of a structured model quality assurance process that concentrates on the syntactical dimension of model quality. It is structured as follows: first, we give an overview on model quality assurance and selected model quality assurance techniques in Section 3.1. The following Section 3.2 describes the defined quality assurance process. Here, Section 3.2.1 presents a process for the application of project-specific model quality assurance techniques in ongoing projects. In order to specify these project-specific techniques in a structured way, we define such a process in Section 3.2.2.

3.1 MODEL QUALITY ASSURANCE

For defining a structured model quality assurance process we first give an overview on model quality assurance and selected model quality assurance techniques.

Quality management can be either product-oriented or process-oriented. The former perspective means that software artifacts (but also intermediate results) are checked against predefined quality aspects whereas the latter perspective addresses artifacts that are related to the software development process like methodologies, tools, guidelines, and standards.

A prominent example for process-oriented quality management is CMMI (Capability Maturity Model Integration) [25]. CMMI is a model and de-facto industry standard that consists of best practices that address the development and maintenance of products and services. It covers the life cycle of a product from conception through delivery to maintenance and integrates essential bodies of knowledge for developing products, such as software engineering, systems en-

gineering, and acquisition. Furthermore, CMMI is the successor of the capability maturity model (CMM) for Software that was developed from 1987 to 1997 at Carnegie Mellon Software Engineering Institute (SEI) [152]. However, since we consider the quality of software models, we do not refer to process-oriented quality management techniques. Instead, this thesis uses product-oriented model quality assurance tasks and considers software models as artifacts under observation.

Since modeling languages often provide at least one graphical view, model quality can be either considered for this visual representation(s), i.e., the concrete model syntax, or for the underlying structure, i.e., the abstract syntax of the model.

Several research papers dealing with the quality of software models address the concrete syntax level. Here, especially models of the Unified Modeling Language (UML) [123] are considered since it provides the concept of views on selected parts of the model, called diagrams. In [154] for example, van Elsuwe and Schmedding discuss three generic metrics being usable to assess information on diagrams that help on reasoning about their quality. The metrics address the informative content of the diagram, its visual size as well as its complexity. In another work, Störrle reports on the results of three controlled experiments using compound layouts on requirements analysis models, i.e., on UML use case, class, and activity diagrams [145]. He observed (1) that the impact of layout quality should be more apparent in models and diagram types used in earlier life cycle phases and (2) that good layouts use many different heuristics simultaneously instead of using them in isolation. Furthermore, Störrle noticed that novice modelers benefit far more from good layouts than advanced modelers. However, this thesis concentrates on on the syntactical dimension of model quality, i.e., on those quality aspects which can be checked on the model syntax only.

Quality assurance is the definition of processes and standards that should lead to high-quality products and the introduction of quality processes into the manufacturing process [143]. A variety of model quality assurance techniques exist and are subject of several research activities. They can be subdivided into two categories. *Analytical* techniques measure the current quality level of the artifact (in our case the software model) whereas *constructive* techniques guarantee for higher quality when applied during the development of the artifact (in our case the modeling activity).

In this thesis, we consider the analytical model quality assurance techniques model metrics and model smells. Metrics can be used to obtain quantitative information about processes or artifacts like software models. Especially for evaluating quality issues metrics are very helpful. In this context, the use of Goal-Question-Metrics paradigm (GQM) presented by Basili et al. [9], a mechanism for defining and

evaluating a set of quality goals by using measurements (metrics), has been proven well for the last 20 years. The concept of code smells has been coined by Kent Beck and Martin Fowler [11, 64] and has been lifted to models leading to model smells. They represent suspicious model parts that are potential candidates for improvements, i.e., they are not synonyms for problems but are worthy of an inspection.

Special kinds of model smells are model clones representing similar or identical fragments in a model. Again, this concept originates from the corresponding counterpart on the code side, that is code respectively software clones. A fair bit of research has been done addressing several topics on code clones, for example clone prevention, detection, and deletion. A comprehensive survey of research on software clones can be found in [92]. Here, Koschke discusses (among others) several notions of redundancy and similarity as well as various categorizations of clone types. A representative research on the model side is presented in [146]. Here, Störrle analyzes the concept of clones in UML domain models, i.e., use case, class, activity, and state machine models. He establishes a practical definition of model clones based on a structural analysis of real clones, develops heuristics and a clone detection algorithm, and presents an implementation of the approach. However, we do not further follow the concept of model clones in this thesis in order to narrow its scope to be manageable.

Constructive model quality assurance techniques can be used to improve the quality of software models. In the following, we discuss two kinds of constructive techniques, namely modeling guidelines and model refactoring. A further constructive quality assurance technique is the structured use of software design patterns [85] for providing a general reusable solution to a commonly occurring problem in software design possibly combined with providing syntax-oriented complex editing operations to increase the convenience of the corresponding model editor [149].

In order to avoid problems with respect to model quality, in particular model smells, the use of modeling conventions analogue to coding conventions are appropriate. In [98], Lange et. al. define modeling conventions as conventions to ensure a uniform manner of modeling and to prevent for defects. They report on the results of a controlled experiment to explore the effectiveness of modeling conventions for UML models with respect to prevention of defects. The results indicate (1) that decreased defect density is attainable at the cost of increased effort when using modeling conventions, (2) that this tradeoff is stressed if tool support is provided, and (3) that efficient integration of convention support in the modeling process forms a promising direction towards preventing defects. Since in this thesis we do not consider the prevention of inserting model smells, we do not use this technique in the following. Instead, we consider another construc-

tive model quality assurance technique that is usable to correct model smells in order to improve the model's quality: model refactoring.

'Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure' [64]. Basically introduced to software code, refactoring has been successfully lifted to the level of software models, especially for (UML) class models being closely related to class structures in object-oriented programming languages such as C++ and Java. For behavior models, the relation between models and code, and therefore the adoption of code refactorings to model refactorings, is less obvious.

In summary, we use the following model quality assurance techniques in the remainder of this thesis:

- *model metrics* and *model smells* as analytical model quality assurance techniques for detecting quality defect, and
- *model refactoring* as constructive model quality assurance technique for correcting them.

However, the problem concerning these techniques (respectively their discussions in literature) is that metrics, smells, and refactorings are used in different kinds of model quality assurance tasks, i.e., are considered in isolation. So, the challenge is to combine them to provide an integrated usage within a structured quality assurance process for software models. The following section defines such an integrated process.

3.2 PROCESS DEFINITIONS

The increasing use of model-based or model-driven software development processes induces the need for high-quality software models. Therefore, we propose a model quality assurance process that consists of two sub-processes: a process for the specification of project-specific model quality assurance techniques, and a process for applying them on concrete software models during a model-based software development process.

3.2.1 *Application process*

Figure 3.1 illustrates the process for the application of project-specific model quality assurance techniques. For a first rough model overview, a report on model metrics might be helpful. Furthermore, a model can be checked against the existence (respectively absence) of specified model smells. Each model smell found has to be interpreted in order to evaluate whether it should be eliminated by a suitable model modification (either by a manual model change or a refactoring). However, we have to take into account that also new model

smells can be induced by refactorings and care should be taken to minimize this effect. This check-improve cycle should be performed as long as needed to get a reasonable model quality.

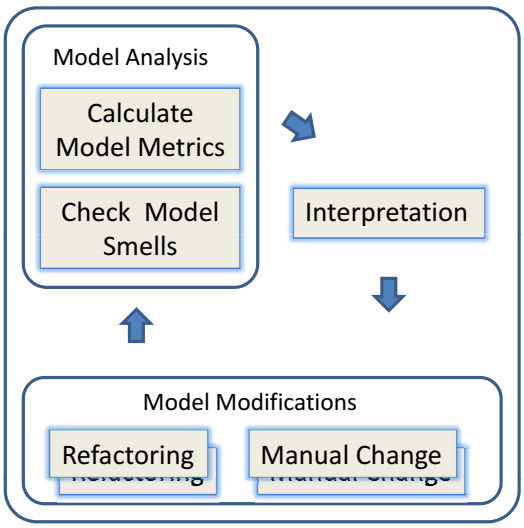


Figure 3.1: Process for the application of project-specific model quality assurance techniques

The application process can be embedded into several kinds of software development process models, for example:

- According to the traditional waterfall model, the application process can be embedded at several points in time during the design phase. However, it must be completed before the subsequent implementation phase starts.
- In iterative process models such as the spiral model [18], the application process should be embedded after each iteration step in order to analyze and improve the design of the current increment. The same applies to agile methods such as extreme programming (XP) [10] and Scrum [139].
- In model-driven software development processes, the application process should be embedded definitely before the generation process starts, either the generation of models of the subsequent stage, of documentation artifacts, or even code.

Ideally a quality assurance process is fully specified before using it within model-based software development projects. However, it is not seldom that the process has to be adapted during the model development phase. Our process allows the straight adaptation to new model checks and refactorings.

3.2.2 *Specification process*

Figure 3.2 shows the process for specifying new model quality assurance techniques. After having identified the intended modeling purpose the most important quality goals are selected. Here, we have to consider several conditions that influence the selection of significant quality aspects being the most important ones for modeling in a specific software project. The first issue to consider is that the selection of significant quality aspects highly depends on the modeling purpose. There is a variety of purposes for modeling in software projects. For example, models can be used for communication purposes between stakeholders, being customers and requirements engineers or project managers and software designers. In other projects, software models may be used for code generation purposes, to generate the application code and/or code that is used in tests for implemented software components. Since modeling purposes are quite different and vary in several software projects, a quality aspect that is very important in one software project might be less important in other ones. For instance, in projects that use software modeling for communication purposes the *comprehensibility* of the model might be the most relevant quality aspect whereas aspects *correctness* and *completeness* are more important for models that are used for the generation of application or test code, respectively.

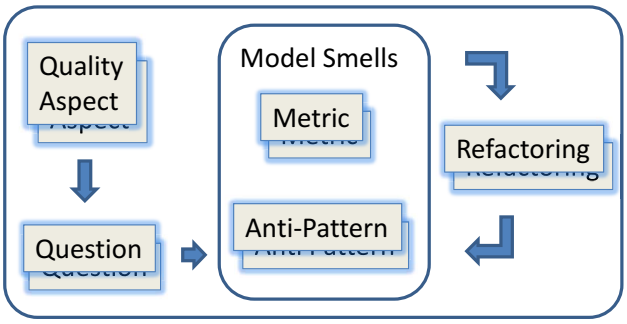


Figure 3.2: Process for the specification of project-specific model quality assurance techniques

Another factor that influences the significance of a model quality aspect is the corresponding application domain. This means that software models are used in various domains like web applications or embedded systems having different impacts on the significance of a certain model quality aspect. For example, for models of safety-critical embedded systems, *correctness* is more important than models of usual web applications.

The preceding discussions show that it is appropriate to set up a specific model quality assurance process for each software project be-

ing dependent on the modeling purpose as well as the corresponding modeling domain.

In the next step, static syntax checks for these quality aspects are defined. This is done by formulating questions that should lead to so-called model smells hinting to model parts that might violate a specific model quality aspect. Here, we adopt the goal-question-metrics approach (GQM) that is widely used for defining measurable goals for quality and has been well established in practice [9]. In our approach, we consider the syntax of the model in order to give answers to these questions. Some of these answers can be based on metrics. Other questions may be better answered by considering specific patterns which can be formulated on the abstract syntax of the model. However, further static analysis techniques could be incorporated to find out additional potential model smells. Furthermore, the project-specific process can (re-)use general metrics and smells as well as special metrics and smells specific for the intended modeling purpose.

Refactoring is the technique of choice for fixing a recognized model smell. A specified smell serves as precondition of at least one model refactoring that can be used to restructure models in order to improve model quality aspects but appreciably not influence the semantics of the model. In this context, it is also recommended to analyze the specified refactorings whether the application of a certain refactoring may cause the occurrence of a specific model smell.

Since the process of manual model reviews is very time consuming and error prone, several tasks of the proposed project-specific model quality assurance process should be automated as effectively as possible. The following tasks of the process can be automated:

- Support for the implementation of new model metrics, smells, and refactorings using several concrete specification languages.
- Calculation of implemented model metrics, detection of implemented model smells, and application of implemented model refactorings.
- User-friendly support for project-specific configurations of model metrics, smells, and refactorings.
- Generation of model metrics reports.
- Suggestion of suitable refactorings in case of specific smell occurrences.
- Provision of warnings in cases where new model smells come in by applying a certain refactoring.

Part II of this thesis presents a flexible tool environment for model metrics reports, smell detection, and refactoring for models that is based on the Eclipse Modeling Framework (EMF) [144, 44].

4

MODEL QUALITY AND MODEL QUALITY ASPECTS

To evaluate the quality of a software product the use of a well-defined quality model representing the characteristics of the product that describe the quality has been established for more than three decades. The objective of this chapter is to define a quality model for model quality consisting of high-level quality attributes and low-level quality characteristics which condenses the mainly discussed topics in research literature. In the following sections, we first discuss quality models for software products in common and their adaptation to software models. Then, we present comprehensive definitions of quality aspects and characteristics extracted from selected review articles. Finally, we develop a quality model for model quality based on these articles which serves as basis for several parts in the remaining chapters of this thesis.

4.1 FROM SOFTWARE QUALITY TO MODEL QUALITY

In this section, we first present several quality models for software products. Afterwards, we discuss related work on the quality of software models, especially models of the Unified Modeling Language (UML) [123].

4.1.1 *Software quality models*

The development of high-quality products is a broadly discussed research topic throughout the last 40 years. In the 1970s and 1980s, several quality models for software products gradually evolved (for example, McCall Model, Boehm Model, and FURPS/FURPS+ Model) resulting in the quality model specified in the ISO/IEC 9126 standard which was first established in 1991. The following paragraphs shortly describe the core concepts of these quality models and summarize the explanations taken from [74].

The McCall Model

One of the first quality models for software products representing a corner stone for today's quality models was established by Jim Mc-

Call et al. in 1977 [112]. It consists of 11 high-level quality factors (*maintainability*, *flexibility*, *testability*, *portability*, *reusability*, *interoperability*, *correctness*, *reliability*, *efficiency*, *integrity*, and *usability*) reflecting both the user's and the developer's view which are classified in three major types (product *revision*, *transition*, and *operations*).

Each quality factor is positively influenced by a set of quality criteria whereby each criterion in turn can influence a number of quality factors. The quality model defines altogether 23 quality criteria like *modularity* and *storage efficiency* and 32 relations between high-level quality factors and low-level criteria. Further major contributions of the McCall Model are the relationships created between quality characteristics and metrics to concretely measure quality criteria.

The Boehm Model

In 1978, Barry Boehm et al. [19] presented a quality model for software products that also addresses hardware characteristics. Similar to the McCall Model this model specifies quality characteristics in a hierarchical structure. The focus of the Boehm Model is on *maintainability* being one of the three high-level quality characteristics (*as-is usability* [1], *maintainability*, and *portability*).

On the intermediate level, the Boehm Model includes seven quality factors (*portability*, *reliability*, *efficiency*, *usability*, *testability*, *understandability*, and flexibility) which are further sub-divided into 15 primitive characteristics (for example, *device independence* and *structuredness*) to provide the foundation for defining quality metrics.

The FURPS/FURPS+ Model

Another quality model for software products is the FURPS/FURPS+ Model originally presented by Robert Grady at Hewlett Packard [77, 76]. This model is organized quite differently from either McCall Model or Boehm Model.

The FURPS/FURPS+ Model provides five general categories being of two different types according to the user's requirements: functional requirements defined by input and expected output (***F**unctionality*) and non-functional requirements (***U**sability*, ***R**eliability*, ***P**erformance*, and ***S**upportability*). Finally, more than 30 quality characteristics (like *aesthetics* and *modifiability*) again form a hierarchy similar to the models discussed before.

The ISO/IEC 9126-1 (ISO/IEC 25010) Quality Model

To overcome problems and uncertainties with the diversity of the described quality models the International Organization for Standardization (ISO) [42] and the International Electrotechnical Commission

1 This is the main focus of the McCall Model.

(IEC) [26] developed a standard quality model for software products. The ISO/IEC 9126-1 standard *Software engineering – Product quality – Part 1: Quality model* [41] was first published in 1991 and slightly enhanced in 2001.

Characteristic	**Description**
Functionality	This quality characteristic describes the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions (what the software does to fulfill needs).
Reliability	This quality characteristic describes the capability of the software product to maintain its level of performance under stated conditions for a stated period of time.
Usability	This quality characteristic describes the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions (the effort needed for use).
Efficiency	This quality characteristic describes the capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.
Maintainability	This quality characteristic describes the capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in the environment and in the requirements and functional specifications (the effort needed to be modified).
Portability	This quality characteristic describes the capability of the software product to be transferred from one environment to another. The environment may include organizational, hardware or software environment.

Table 4.1: High-level quality characteristics of the ISO/IEC 9126-1 quality model (taken from [101])

In the ISO/IEC 9126-1 quality model, the totality of software product quality attributes are classified in a hierarchical tree structure of characteristics and sub characteristics. The standard specifies six independent, not directly measurable high-level quality characteristics as described in Table 4.1. They are further divided into 21 sub characteristics (respectively quality criteria) as shown in Figure 4.1.

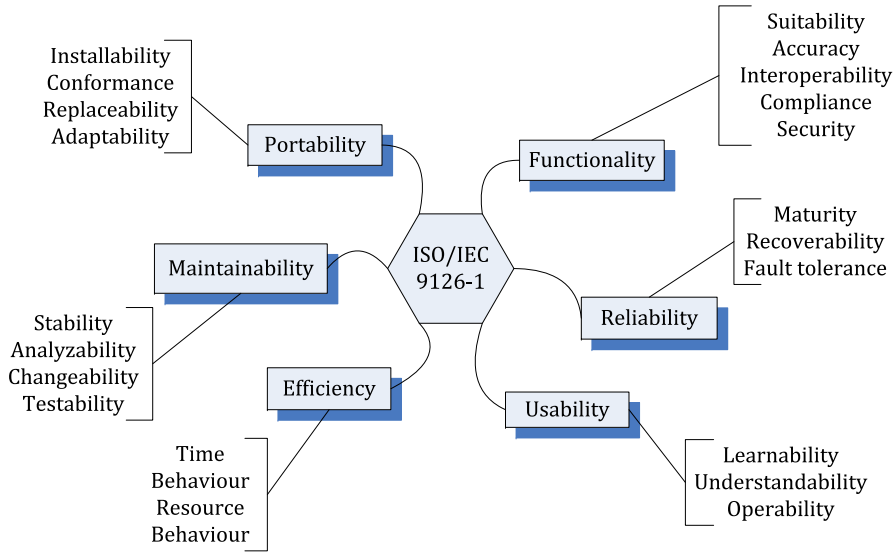


Figure 4.1: The ISO/IEC 9126-1 quality model

In 2011, ISO/IEC 9126-1 has been revised by the new ISO/IEC 25010:2011 standard *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models* [40]. However, the vast majority of research literate and software economy still focus on the ISO/IEC 9126-1 quality model since it represents the commonly accepted state-of-the-art of software product quality specifications.

4.1.2 *Software model quality*

As demonstrated in the previous section, there has been a great deal on research on software quality, especially on the quality on software code. Since in modern software development processes models become the main artifacts, reasoning about software quality often leads back to reasoning about the quality of the involved software models. However, there has been relatively little work on model quality basically caused by the lack of an understanding what model quality exactly means.

Existing knowledge on software quality as presented in the previous section can be applied on model quality to a limited extent only. There are significant differences between source code and software models, for example different levels of abstraction or a different use with respect to execution (mostly, models are not intended to be executable). Therefore, several quality characteristics having a significant relevance for the quality of software products can not be considered when reasoning about model quality (for example, quality characteristics *Reliability*, *Efficiency*, and *Security*).

The following paragraphs give an overview on the related work on model quality in model-based software development in general and specifically when modeling with the Unified Modeling Language (UML) [123].

Model quality in model-based software development

In a systematic literature review (SLR) following the guidelines presented in [87], Mohegheghi et al. analyzed 40 primary studies focused on model quality within the domain of model-based and model-driven software development [116]. The analyzed studies have been published between 2000 and 2007; their sources include books, journals, conference and workshop proceedings, PhD theses, and online publications. The following listing shows the main results of the SLR:

- The authors identified six main quality goals: *Correctness*, *Completeness*, *Consistency*, *Comprehensibility*, *Confinement*, and *Changeability*. We discuss these so-called *6C Goals* in more detail in Section 4.2.1.
- Best practices with respect to the *development process* are:
  - Use of a *model-based* development process with ongoing model analyses during *reviews* using mechanisms like *metrics*. The quality assurance process presented in Chapter 3 refers to this recommendation.
  - Use of *modeling guidelines* to avoid syntactical and semantical errors and verification of the proper use, for example by using *checklists*.
  - Use of a *One-Diagram-Strategy* when using modeling languages which provide multiple diagram views like the UML to avoid misunderstandings, incomprehensibilities, and inconsistencies.
- Best practices with respect to *formal methods and automation* are:
  - Use of *formal models* to provide precise modeling, formal analyses and proofs, execution, and generation.
  - Use of *domain-specific languages* or *UML extensions* to avoid incorrect models by encoding domain-specific concepts and rules.
  - Use of *generation facilities* to generate models from other ones in order to improve consistency between models and completeness of the resulting model.
- The most models being discussed in literature are UML models.

The last result leads to another SLR present in the following section.

Quality of UML models

A SLR specifically dedicated to the quality of conceptual models written in UML is presented by Genero et al. in [73]. In this SLR, the authors extracted and analyzed altogether 266 peer-reviewed papers published between 1997 and 2009. The main results of this SLR are:

- There is no clear view of the real state of the field, " . . . although quality of models . . . is a 'hot topic' that needs further investigation [157]."
- More than half of the research concentrates on semantic consistency (113 out of 266 papers). We discuss this result in more detail later on in this section.
- The application of quality assurance techniques being well established for software code (like testing, analysis, and inspection) in the context of UML models is still in an 'embryonic phase'. However, more than 75% of the extracted papers deal with UML quality assurance techniques and the evaluation of UML model quality.
- The type of UML diagram that has been studied most is the class diagram (83 out of 163 papers dealing with models of specific diagram types; being 50.9%).

One research question examined in the SLR addresses the type of quality which is covered in the corresponding paper(s). However, the authors do not use a quality model specific to UML. Instead, they use a mixture of quality characteristics from ISO/IEC 9126 and those drawn from selected papers (without giving concrete references).

As mentioned above, one result of the SLR is that semantic consistency is the quality characteristic which has been researched most. An overview on this specific topic can be found in the SLR performed by Lucas et al. [102]. Here, the authors reviewed 44 papers published between 2001 and 2007 focusing only on consistency within UML models. Their conclusion is that UML consistency is a 'highly active and promising line of research' but there are still some gaps being not addressed in literature.

One of the prominent researches in this field is done in the work of Alexander Egyed [33, 34]. Here, mainly (in)consistencies across two or more UML diagrams that makes up a complete UML model are addressed. However, this work differs on the approach presented in this thesis in several facts. First, the majority of considered inconsistencies represent either violated well-formedness rules respectively constraints (like a message in a sequence diagram having no corresponding class operation). Second, several considered inconsistencies should be better addressed by the UML language definition (for example, the need for a referenced concrete class when modeling a life-

line in a sequence diagram) leading back to violated constraints mentioned above instead of representing model smells in the sense of this thesis. Finally, the author concentrates on an approach for 'quickly ... deciding what consistency rules to evaluate when a model changes' while the smell analysis presented in this thesis is neither performed during modeling nor is it time critical.

The results of the SLR performed by Genero et al. [73] also show that more than 90% of the addressed quality types correspond to five of the main model quality characteristics extracted in the SLR performed by Mohagheghi et al. [116] as presented in the previous section (the *6C Goals*). These are (in the order of significance): *Consistency*, *Comprehensibility*, *Correctness*, *Changeability*, and *Completeness*.

As a consequence of using a mixture of model quality characteristics (see above), Genero et al. state that there is no consensus on the quality characteristics addressed nor on their definitions (even not in the book by Bhuvan Unhelkar [151]). They extracted only one quality model for UML model quality that has been proposed in literature leading us to another related work in the field of model quality.

In the work of Christian F.J. Lange [95, 97], a quality model is defined which is specific for UML modeling. The purpose of this quality model is to provide guidance in selecting metrics and rules to assess the quality of UML models. These concepts are closely related to the concepts used in this thesis, however the work (1) concentrates on the quality of UML models only and (2) does not use any techniques like refactorings to eliminate quality defects respectively model smells.

The quality model presented in [97] consists of four levels. The first two levels represent two `Usages` (*development* and *maintenance*) and eight `Purposes` of software models (like *communication* and *code generation*). The third level of the quality model contains 12 inherent quality characteristics (like *complexity*, *detailedness*, and *aesthetics*) whereas the fourth level represents metrics and rules to measure these characteristics. In the resulting quality model, purposes are related to usages, quality characteristics are related to purposes, and metrics respectively rules are related to quality characteristics. However, the quality model implies several imprecise facts which are hard to understand and therefore at least worthy to discuss. First, the relations between the purpose of modeling and the use of the models form a total function in a sense that each purpose is related to exactly one use. This is quite unusual since, for example, models which are used to comprehend a system (purpose *comprehension*) my occur in both the development and the maintenance phase. Second, also the relation between quality characteristics and purposes seems to be incomplete. Here, the missing relation between characteristic *communicativeness* and purpose *communication* is an obvious example. Third, several metrics are considered concretely (like *DIT* and *NCU*) whereas other metrics are clustered (like *ratios* and *code matching*) making a relation between

such a cluster and the quality characteristic which can be measured by it very hard. Similar observations can be made when looking at the rules considered in the fourth level of the quality model. Here, concrete defects like *Multiple Definitions of Classes with Equal Names* as well as clustered defects like *Adherence to Naming Conventions* are addressed. Nevertheless, the work by Christian F.J. Lange give inspirations within several parts of this thesis.

4.2 MODEL QUALITY ASPECTS

This section presents the core concepts of two articles concerning model quality characteristics in the context of a model-based software development process. These concepts serve as a basis for the development of a quality model for model quality which is presented in the subsequent section.

4.2.1 *6C model quality goals*

In [116], Mohegheghi et al. present the results of a systematic literature review (SLR) discussing model quality in model-based software development according to the guidelines presented in [87]. Among others, the purpose of the SLR was to identify what model quality means, i.e., which quality goals are defined in literature. The review was performed systematically by searching relevant publication channels for papers published from 2000 to 2007. From 40 studies covered in the review, the authors extracted six classes of quality goals in model-based software development, the so-called *6C goals*. They state that other quality goals discussed in literature, like *conformity* and *simplicity*, can be satisfied if the 6C goals are in place. The remainder of this section shortly introduces the identified 6C goals.

CORRECTNESS: A model is **correct** if it includes the right elements and correct relations between them and, what is most important, if it includes correct statements about the domain. Furthermore, a model must not violate rules and conventions. This definition includes *syntactic correctness* relative to the modeling language as well as *semantic correctness* related to the understanding of the domain.

COMPLETENESS: A model is **complete** if it has all necessary information that is relevant, and if it is detailed enough according to the purpose of modeling. For example, requirement models are said to be complete when they specify all the black-box behavior of the modeled entity, and when they do not include anything that is not in the real world.

CONSISTENCY: A model is **consistent** if there are no contradictions within. This definition covers *horizontal consistency* concerning models/diagrams on the same level of abstraction, *vertical consistency* concerning modeled aspects on different levels of abstraction as well as *semantic consistency* concerning the meaning of the same element in different models or diagrams.

COMPREHENSIBILITY: A model is **comprehensible** if it is understandable by the intended users, either human users or tools. In most of the literature, the focus is on comprehensibility by humans including aspects like aesthetics of a diagram, model simplicity or complexity, and the use of the correct type of diagram for the intended audience. Several authors also call this goals *pragmatic quality*.

CONFINEMENT: A model is **confined** if it agrees with the modeling purpose and the type of system. This definition also includes relevant diagrams on the right abstraction level. Furthermore, a confined model does not have unnecessary information and is not more complex or detailed than necessary. Developing the right model for a system or purpose of a given kind also depends on selecting the right modeling language.

CHANGEABILITY: A model is **changeable** if it can be evolved rapidly and continuously. This is important since both the domain and its understanding as well as requirements of the system evolve with time. Furthermore, changeability should be supported by modeling languages and modeling tools as well.

4.2.2 *A taxonomy of model quality characteristics*

In another article concerning model quality in model-based software development, Fieber et al. present a taxonomy of model quality aspects [38]. For a better differentiation from the 6C goals recalled in the previous section, the quality aspects presented by Fieber et al. are referred to as *quality characteristics* in the following. The following paragraphs describe selected model characteristics similar to the 6C goals described above.

PRESENTATION: How good is the visual perception and acceptance by the user? How good is the layout of a diagram? How many elements are displayed in a diagram?

SIMPLICITY: Is a model too complex? Is it possible to simplify model structures? Is the model complexity necessary? Simplicity addresses the aspect of how complex something is modeled. A model should not be more complex than required. Some features of a model can be expressed using different kinds of structures without changing the semantics or precision. In case of

behavior models, simplicity can also be understood as the opposite of control flow complexity. Another interpretation of this quality aspect is related to the purpose of the model. Anything that does not contribute to proper modeling purpose should not be displayed.

CONFORMITY: Are all naming conventions respected? Are any modeling conventions violated? Conformity means the conformance to modeling standards, e.g., all attributes in a class diagram have to be named in *camel case*[2].

COHESION/MODULAR DESIGN (MODULARITY): Does each model element have a well-defined responsibility? Are modeled features reusable in other projects? Cohesion and modular design are strongly related to the coupling of model elements. While cohesion is related to dependent system aspects, the modular design is related to technical independent aspects or independent aspects with regards to content. For example, the fact that security is modeled in one component is addressed by the cohesion aspect. In contrast, modular design requires that each class has only one role of responsibility.

REDUNDANCY: Is the used redundancy in the model mandatory? On the one hand, redundancy in models should be reduced, because redundancy is always error-prone. On the other hand, some controlled redundancy can be useful, e.g., for test-code generation. Fieber et al. give an example where they use state charts as input for the application-code generator and sequence diagrams as input for the test-code generator.

SEMANTIC ADEQUACY: Does the model use a proper modeling language? Are adequate elements or diagrams used for modeling a specific aspect? Some aspects of a model can be modeled using different kinds of diagrams or modeling languages. If this kind of diagram fits the modeled aspect, it is a question of semantic adequacy.

CORRECTNESS: Is a model semantically and syntactically correct? For example, if an object model uses an instance of an abstract class this violates the correctness. If a model element is useless, that may be a semantical error in the model.

PRECISION: How detailed are the relevant aspects of the system described? Precision of a model concerns how only relevant features of the domain or other artifacts are addressed by the model. In a precise model each omitted feature of the original

2 A naming convention which is common practice in Java. It uses medial capitals, one example is: *Window*A*dapter*.

aspect is in fact irrelevant for the current development phase or modeling purpose.

COMPLETENESS WRT. PRECEDING PHASES Are all requirements completely covered by the model? Are all information from the preceding phase in the model chain completely transfered to the correct phase?

COMPLETENESS WRT. SUBSEQUENT PHASES This model characteristics means that the model contains all necessary information to deduce or generate the artifacts of the subsequent phase. In fact, this is a semantic characteristic related to models of one level in the model chain.

TRACEABILITY Traceability is a relationship between models across multiple stages. Changes in corresponding artifacts can be traced in order to get statements about effect and effort of necessary changes. By using traceability statements, the completeness wrt. preceding phases can be determined, for example.

CHANGEABILITY Is is possible to change respectively evolve the model in an easy way? Is it possible to reuse the model respectively parts of the model in other projects? Changeability can further be sub-divided in aspects maintainability, extensibility, and reusability.

4.3 A QUALITY MODEL FOR MODEL QUALITY

In this section, we develop a quality model for model quality based on the concepts presented in the previous section. We use this quality model later on in this thesis, for example in Chapters 5 and 6.

To combine the concepts of the presented articles, we analyzed the model characteristics presented in [38] with respect to their relationships to the 6C goals defined in [116]. General relation: Our perception is that model characteristics by Fieber et al. are more specific than corresponding 6C goals. In more detail, we detected that some model characteristics represent partial definitions of certain 6C goals whereas other just influence some of them. Table 4.2 presents the results of this analysis. An entry `D` in cell *(i,j)* denotes that model characteristic *j* represents a (partial) definition of goal *i*. An entry `I` in cell *(i,j)* indicates that model characteristic *j* influences goal *i*. This matrix is helpful in matching quality assurance techniques to specific model quality goals as presented in Chapter 5.

As shown in Table 4.2 there are some characteristics used by Fieber et al. that are synonyms or partial definitions of corresponding 6C goals. *Conformity* in the sense of Fieber et al. can be a synonym for *syntactic correctness*. By *semantic adequacy* Fieber et al. address *confinement* as goal. Furthermore, there are characteristics which are com-

parable to equally called quality goals (*correctness*, *completeness*, and *changeability*).

	Quality Characteristics											
6C Goals	*Presentation*	*Simplicity*	*Conformity*	*Cohesion/Modular Design (Modularity)*	*Redundancy*	*Semantic Adequacy*	*Correctness*	*Precision*	*Completeness wrt. preceding phases*	*Completeness wrt. subsequent phases*	*Traceability*	*Changeability*
Correctness			D	I	I		D	I				
Completeness								I	D	D	I	
Consistency					I						I	
Comprehens.	I	I		I	I						I	
Confinement		I				D		I				
Changeability		I			I						I	D

Table 4.2: Relationships of quality characteristics presented by Fieber et al. [38] to 6C quality goals defined by Mohagheghi et al. [116]

Nevertheless, Fieber et al. define model characteristics which do not define but just influence 6C goals. First, it is obvious that characteristic *presentation* only influences goal *comprehensibility*. *Simplicity* however influences *comprehensibility* since a complex model is hard to understand, goal *confinement* since a complex model might be a hint to a wrong level of abstraction or a wrongly selected modeling language, and goal *changeability* since complex models are hard to evolve. Fieber's *Cohesion/Modular Design (Modularity)* represents a technique that influences *correctness* since an incorrect assignment of a model element to a module might not reflect the real world's aspect and goal *comprehensibility* since an incorrect assignment of a model element to a module might lead to misunderstandings. *Redundant* model parts can be seen as incorrect modeling since they do not reflect the modeled aspect of the real world when single aspects are represented multiply. They may lead to contradictions in the model and may be harder to understand and to maintain. An *imprecise* model might be neither correct nor complete. Furthermore, this may be a hint to a disagreement with the modeling purpose or the current level of ab-

straction. Last but not least, model parts that can not be *traced* may be hints to an incomplete model and may lead to contradictions and misunderstandings and are harder to maintain.

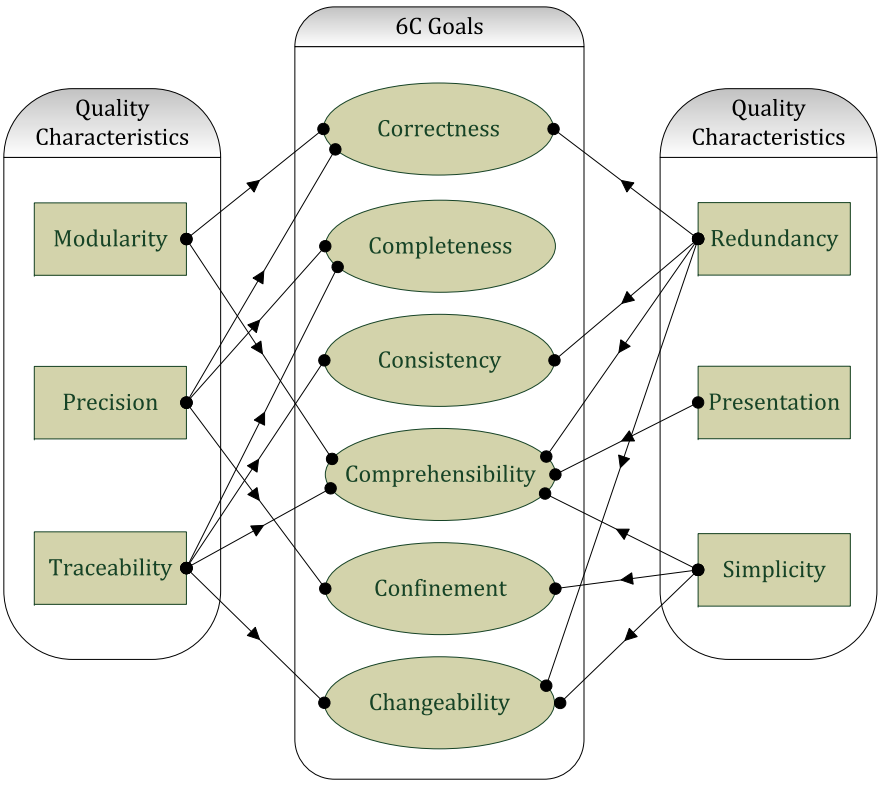


Figure 4.2: A quality model for model quality

Based on Table 4.2 and the discussion above a corresponding quality model for model quality can be derived as shown in Figure 4.2. Here, the 6C goals represent the center of the model while the quality characteristics on their right and left influence the associated goal(s) to some extent. For simplicity reasons, synonyms and partial definitions as discussed above are left out.

The relations presented in Table 4.2 and Figure 4.2 show that the 6C goals are not disjoint in nature. For example, redundant model parts influence both, goal *comprehensibility* and goal *changeability*. Furthermore, the 6C goals my influence each other. For example, it is easy to see that incorrect model parts are also hard to understand. In this case, a reduced quality with respect to a certain quality goal (*correctness*) also influences another one (*comprehensibility*) in the same negative way (type A). On the other hand, the more complete a model is, the more complex it gets. In this case, improving a certain quality goal (*completeness*) may influence another one (*comprehensibility*) in the opposite direction (type B).

Figure 4.3 illustrates these potential dependencies between 6C goals from an abstract view. It shows that for dependencies of type A the

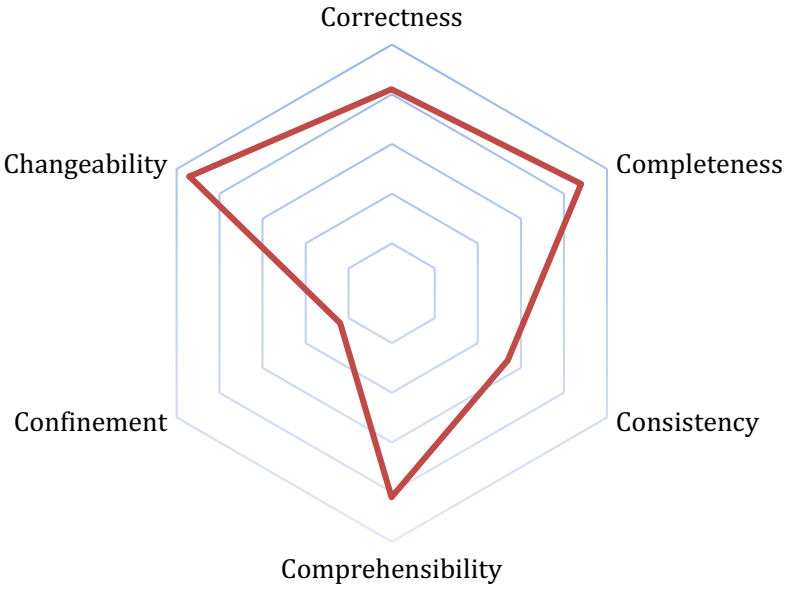


Figure 4.3: Abstract illustration of mutual dependencies between 6C goals

overall quality of the model is reduced (respectively improved), i.e., the inner area is getting smaller (respectively bigger). The influence of dependencies of type B (one edge is shifted outwards, the other one is shifted inwards) on the overall model quality can not exactly be determined. For this reason, it is recommended to prioritize the quality goals with respect to the modeling purpose and domain as discussed in Section 3.2.2.

5

SELECTED MODEL QUALITY ASSURANCE TECHNIQUES

The Unified Modeling Language (UML) [123] is a general-purpose modeling language in the field of object-oriented software engineering that is standardized by the Object Management Group (OMG). Since its adoption in 1997 it has become the mostly used MOF-based modeling language [140]. The UML provides 14 types of diagrams divided into two categories. Seven diagram types represent structural information, and the other seven represent general types of behavior, including four that represent different aspects of interactions.

The structured quality assurance process presented in Chapter 3 uses the quality assurance techniques *model metrics* and *model smells* for analyzing quality aspects, and *model refactorings* for improving them while not changing the semantics respectively the meaning of the model. Metrics, smells, and refactorings for UML models are in the scope of a variety of researchers during the last 15 years.

In this section, we give an overview on these techniques for UML class models being developed and discussed in literature from 1997 to 2009. We concentrate on class models for two reasons: first, because class diagrams are the mostly used UML diagram type [29], and second, since the literature search has been performed as part of a collaboration with Siemens Corporate Technology [142] in the context of the *SPES 2020 Software Platform Embedded Systems* [2, 88] project funded by the German Federal Ministry of Education and Research from November 2009 to January 2012. We start with an overview on metrics, continue with a summary on smells, and finally present a survey on refactorings for UML class models. Note that since these research fields are very broad and active, we consequently do not claim the overviews to be complete.

For each technique, we first present an overview on the total numbers and main sources of the corresponding technique. Then, we present structured descriptions of selected techniques and discuss relations to the quality model discussed in Section 4.3. Comprehensive catalogs with structured descriptions of all techniques found in literature can be found in Appendices A to E of this thesis.

The use of metrics to obtain quantitative information about software development processes and artifact has been proven well for the last 30 years. Especially for evaluating quality issues metrics are very useful which lead to a systematic approach (Goal-Question-Metrics Approach – GQM) presented by Basili et al. in 1994 [9]. Here, they start with the following motivation for the use of metrics:

> As with any engineering discipline, software development requires a measurement mechanism for feedback and evaluation. Measurement is a mechanism for creating a corporate memory and an aid in answering a variety of questions associated with the enactment of any software process.

In this section, we give a brief overview on the research on metrics being discussed in literature for measuring quality issues of UML class models. We further present some descriptions of selected metrics and discuss potential relations to the quality aspects discussed in Chapter 4 of this thesis.

5.1.1 *An overview on UML class model metrics*

The evaluation of UML design models using appropriate metrics is a very active research field in the the 2000s. Here, the majority of UML metrics are based on the object-oriented design metrics developed during the 1990s, for example by Chidamber and Kemerer [24] but also in [100, 109, 23, 110].

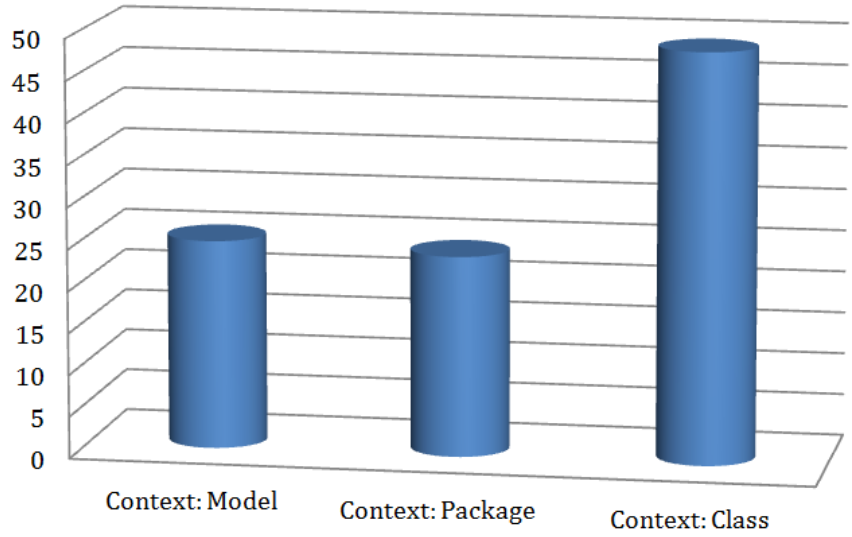


Figure 5.1: Extracted UML class metrics with respect to the contextual type

In a (non-systematic) literature search we concentrated on metrics for UML class diagrams and extracted altogether 98 metrics. These metrics are classified with respect to the contextual type, i.e., the UML meta model element type the metric is calculated on. As can be seen in Figure 5.1, the majority of UML class model metrics is defined for the context type *class* (49 metrics, respectively 50%).

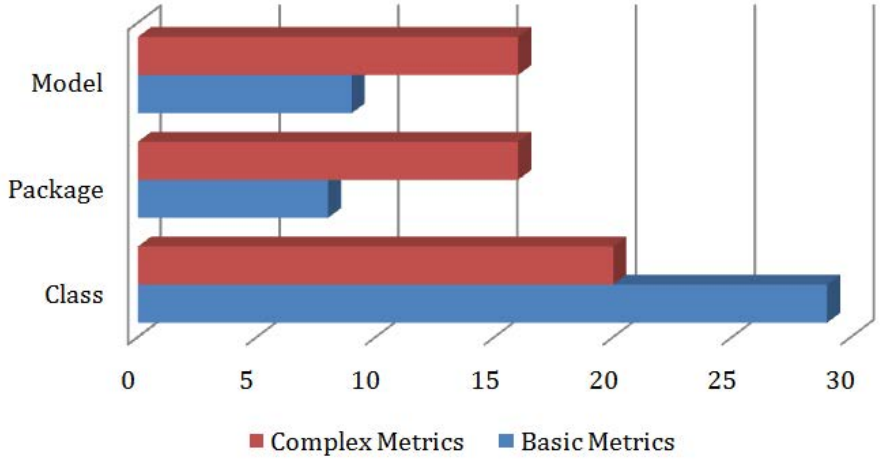


Figure 5.2: Extracted basic and complex UML class model metrics

Furthermore, we distinguish between basic and complex metrics. This means that the definition of a complex metric might rely on one or more basic metrics. Appendix A presents all 46 basic and 52 complex metrics in a comprehensive catalog. Figure 5.2 shows an overview on the basic and complex UML class model metrics.

One of the main research on UML metrics is done by Marcela Genero et al. [69, 70, 71, 72]. In this work, more than 60 metrics for UML class models are discussed. Further main sources concerning UML class model metrics are [105, 84, 86]. During the literature search we further identified 6 metrics for UML use case models and 6 metrics for UML state machines. Here, the main sources are [105] (for use case models) and [115] (for state machines).

5.1.2 *Selected UML class model metrics*

In this section, we present a selection of complex model metrics found in literature. We describe one metric for each context type presented in Figure 5.1 (model, package, and class) as representative. For each metric its name, the context type, a short description, the range of values, and a potential interpretation (including assignments to quality aspects they can measure) are given. The definition of a complex metric might rely on one or more basic metrics. Detailed descriptions of the complex metrics as well as a comprehensive list of basic metrics can be found in the corresponding catalog in Appendix A.

UML class model metric AvsC

CONTEXT: Model

DESCRIPTION: Relation between the number of attributes and number of classes [71]. It is defined as $AvsC = (\frac{NAM}{NAM+NCM})^2$ where NAM is the total number of attributes in the model, NCM is the total number of classes in the model, and $(NAM + NCM) > 0$.

RANGE: $0 \leqslant AvsC < 1$

INTERPRETATION: If the value is higher, model classes have more attributes and the model can be considered to be more complex (affected quality attribute *Comprehensibility*). It is also possible that the model contains unnecessary information and does therefore not correspond to the modeling purpose (*Confinement*). On the other hand, a lower value could be a hint for relevant but missing information (affected quality attribute *Completeness*).

UML class model metric A

CONTEXT: Package

DESCRIPTION: Ratio between number of abstract classes (and interfaces) and total number of classes within the package (abstractness) [84, 109, 110]. It is defined as $A = \frac{NACP+NIP}{NCP+NIP}$ where NACP is the number of abstract classes within the package, NIP is the number of interfaces within the package, and NCP is the number of classes within the package.

RANGE: $0 \leqslant A \leqslant 1$

INTERPRETATION: A higher value indicates a heavier use of abstract classes and interfaces making the model harder to understand (affects quality attribute *Comprehensibility*). This could be interpreted differentially. First, the modeler(s) could use the UML language feature of abstract classes respectively interfaces too exhaustively and therefore not in sync with the modeling purpose (affected quality attributes *Consistency* and *Confinement*). Second, classes could be marked as abstract by mistake (*Correctness*). Third, a high abstractness value could be a hint for relevant but missing concrete classes (affected quality attribute *Completeness*).

UML model metric CBC

CONTEXT: Class

DESCRIPTION: Number of attributes and associations with class type (Coupling between classes) [86]. It is defined as $CBC = DAC + NAC$ where DAC is the number of attributes having another

class as type and NAC is the number of associations to other classes.

RANGE: $0 \leqslant \text{CBC} \leqslant (\text{NATC} + \text{NASC})$ (total number of attributes + total number of associations)

INTERPRETATION: A higher value indicates that the class is stronger coupled to other classes leading to a more complex part of the model which is harder to understand than other ones (quality attribute *Comprehensibility*). Since such a class bears special responsibilities it is also harder to maintain (*Changeability*). However, a high CBC value could be a hint that these responsibilities are modeled by mistake (quality attribute *Correctness*).

5.1.3 *Affected quality aspects*

The following Tables 5.1 to 5.3 show the assignments of the 52 complex class model metrics to the quality aspects they measure.

	6C quality attributes					
UML metric; context: Model	*Correctness*	*Completeness*	*Consistency*	*Comprehensib.*	*Confinement*	*Changeability*
01. AGvsC				×	×	
02. ANA			×	×	×	×
03. AvsC		×		×	×	
04. AScsC	×	×	×	×	×	×
05. DEPvsC	×		×	×	×	
06. GEvsC		×		×	×	
07. MaxDIT				×		
08. MaxHAgg				×		
09. MEvsC		×		×	×	
10. MGH				×	×	
11. MMI			×	×	×	
12. OA3				×		×
13. OA4				×		×
14. OA5				×		×
15. OA6				×		×
16. OA7				×		×

Table 5.1: 6C quality aspects affected by UML metrics (context: Model)

UML metric; context: Package	6C quality attributes					
	Correctness	*Completeness*	*Consistency*	*Comprehensib.*	*Confinement*	*Changeability*
01. A	×	×	×	×	×	
02. AHF	×				×	×
03. AIF		×		×	×	×
04. Ca	×		×	×	×	
05. Ce	×		×	×	×	
06. DN	×			×		
07. DNH				×	×	
08. H	×	×		×		
09. I	×		×	×	×	
10. MHF	×				×	×
11. MIF		×		×	×	×
12. NAVCP	×	×	×	×	×	×
13. PF			×	×	×	
14. PK1	×		×	×	×	
15. PK2	×		×	×	×	
16. PK3	×		×	×	×	

Table 5.2: 6C quality aspects affected by UML metrics (context: Package)

Note that these assignments are not substantiated by solid evidence, for example by performing empirical studies. Such evaluation is out of scope of this thesis. Here, we refer to the corresponding sources of the extracted UML class model metrics.

We summarize the assignments of the 52 complex metrics extracted from literature to quality aspects they can measure in Figure 5.3 on page 42. The most affected quality aspect is *Comprehensibility* (46 metrics respectively 88.5%). This is due to the fact that the majority of metrics is concerned with the complexity of the corresponding model element. The higher the complexity is, the harder is it to understand the model respectively the modeled part.

	6C quality attributes					
UML metric; context: Class	*Correctness*	*Completeness*	*Consistency*	*Comprehensibility*	*Confinement*	*Changeability*
01. APPM		×		×	×	×
02. CBC	×			×		×
03. CBO	×	×		×		×
04. CL1				×		
05. CL2				×		
06. DAM	×		×		×	
07. DCC	×	×		×		×
08. DIT				×		
09. HAgg				×		
10. MAgg				×		
11. MFA		×		×	×	×
12. NASC		×	×	×		×
13. NATC2	×				×	×
14. NDepIn	×		×	×	×	
15. NDepOut	×		×	×	×	
16. NP				×		
17. NW				×		
18. RFC	×		×		×	
19. SIX	×			×	×	×
20. WMC	×				×	×

Table 5.3: 6C quality aspects affected by UML metrics (context: Class)

5.2 SMELLS FOR UML CLASS MODELS

Model smells occur in model parts that are potential candidates for improvements, i.e., they are not synonyms for problems but are worthy of an inspection. The term *smell* is adopted from the concept of *code smell* and lifted to models leading to model smells. The concept

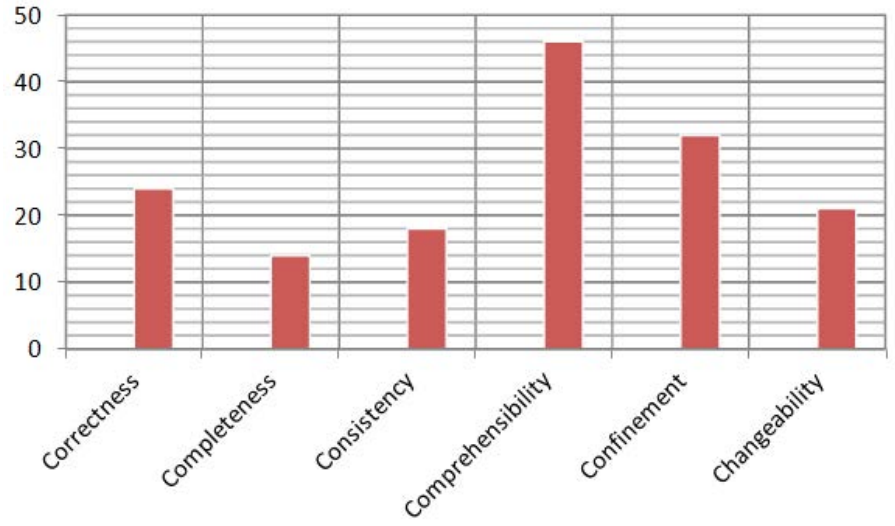


Figure 5.3: Summary of affected quality attributes when interpreting complex UML class model metrics

of code smells has been coined by Kent Beck and Martin Fowler [11]. A useful definition is given on the website of Martin Fowler [65]:

> ... smells don't always indicate a problem. Some long methods are just fine. You have to look deeper to see if there is an underlying problem there - smells aren't inherently bad on their own - they are often an indicator of a problem rather than the problem themselves.

In this section, we concentrate on model smells for class models being the mostly used UML diagram type [29]. After presenting an overview on smells found in literature we give structured descriptions of selected ones. Finally, we discuss potential impacts of UML class model smells on the quality aspects discussed in Chapter 4.

5.2.1 *An overview on UML class model smells*

We extracted altogether 26 smells for UML class models discussed in literature. They are mostly adopted from corresponding code smells presented in [64] and [133]. Further sources are [127] and [97]. A catalog of the identified UML class model smells can be found in Appendix B of this thesis. In this catalog, each model smell is presented by its name and a short description.

Besides smells for UML class models we also identified 4 smells for use case models, 6 smells for sequence diagrams, and 5 smells for state machines. Here, the main sources are [97] (for sequence diagram smells) and [3] (for use case diagram and state machine smells).

Searching for UML model smells in literature is not a straight forward task. On the one hand, the term *model smell* is not commonly

used. Synonyms are for example *inconsistencies* and *defects* (as used by Lange [97]). On the other hand, model smells are seldom main subjects of research. However, since smells are strongly coupled to refactorings, several smells can be derived from appropriate literature in this area (like for example from [127]). Therefore, we do not claim that the catalog Appendix B is complete.

Moreover, we have to filter located smells whether they fit to the right dimension, i.e., whether they affect the abstract model syntax being the basis of the quality assurance process defined in Chapter 3. Therefore, the catalog in Appendix B does not include smells which (1) affect the concrete syntax (like smell *Prominent Attribute*), and which (2) affect the semantic of the model (like smell *Inverted Operation Name*), both taken from [27], and which (3) do not fit to the definition of model smell but instead represent inconsistencies, i.e., violated constraints and well-formedness rules (like *Unnamed Use Case* [3] and *Message without Method* [97] [1]).

5.2.2 *Selected UML class model smells*

In this section we describe selected UML class model smells listed in Appendix B in a structured way. We recognized that several smells described in this catalog can be either specified by appropriate metrics or by corresponding patterns defined on the abstract syntax of the UML [124]. Therefore, we describe one model smell for each of these specification types as representative.

For each model smell a short description is given as well as possible indicators to detect this smell in a given model. Furthermore, we present a list of quality characteristics and quality goals affected by this smell according to the quality model defined in Section 4.3 of this thesis. Lists of refactorings being suitable for eliminating the smell and an example complete each model smell description. Further structured descriptions of UML class model smells can be found in Section 6.1.2 and Appendix D of this thesis.

Long Parameter List

DESCRIPTION An operation has a long list of parameters that makes it really uncomfortable to use the operation. Long parameter lists are hard to understand and difficult to use. Furthermore, using long parameter lists is not intended by the object-oriented paradigm. An operation should have only as much parameters as needed for solving the corresponding task. It is recommended to pass only those parameters that cannot be obtained by the owning class itself [11, 131].

1 See constraint [2] in the specification of meta element `Message` in [124].

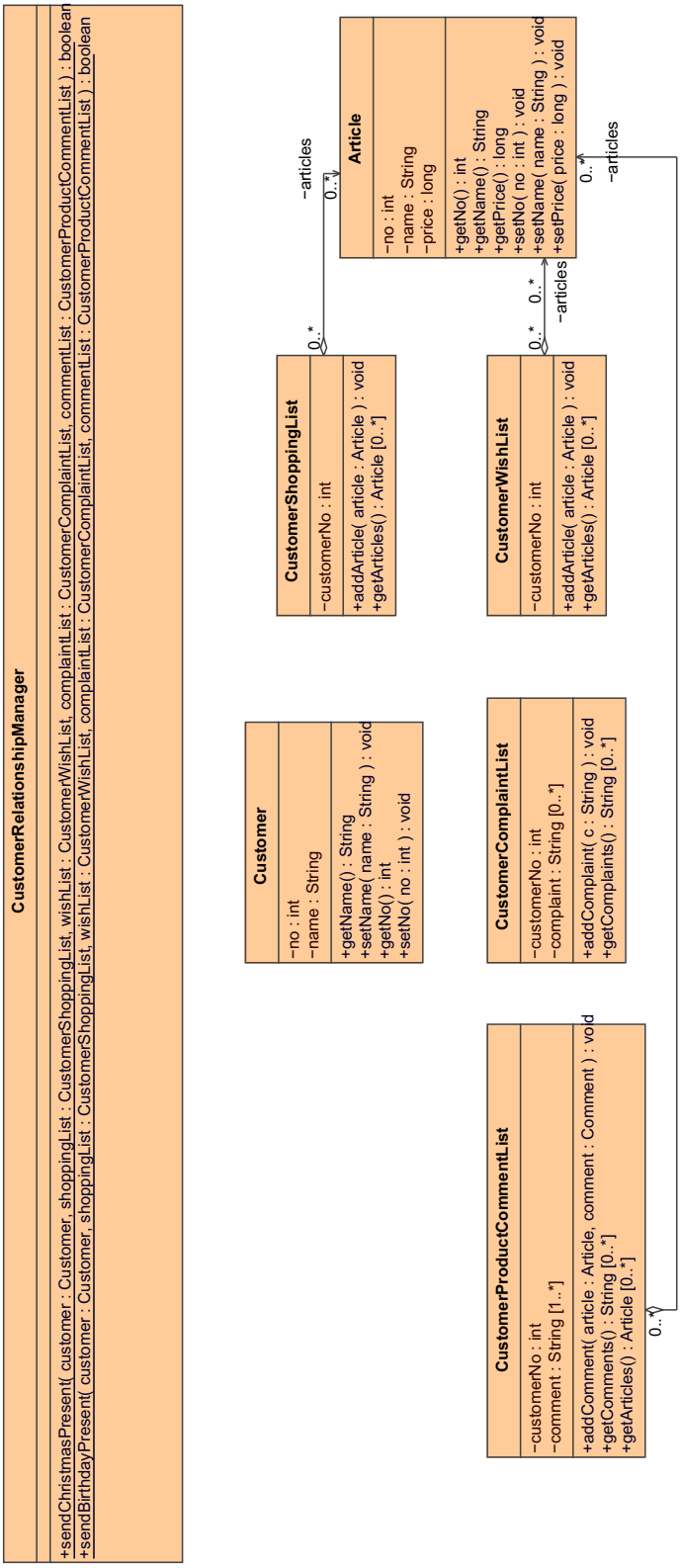


Figure 5.4: Example UML class model smell *Long Parameter List*

EXAMPLE Figure 5.4 shows class *CustomerRelationshipManager* that owns two operations each having a long parameter list. Here, this smell can easily be detected by observation.

DETECTION This smell can be simply detected by observing the model (see above) or by evaluating metric *Number of Input Parameters* and evaluating its value with respect to a predefined threshold value. Metric *Number of Input Parameters* can be specified by the OCL expression

```
self.ownedParameter
-> select(direction = ParameterDirectionKind::_in or
direction = ParameterDirectionKind::inout)
-> size()
```

that returns the number of owned parameters of a given operation with direction *in* respectively *inout*.

USABLE UML MODEL REFACTORINGS *Introduce Parameter Object* for extracting information to a new class. *Remove Parameter* for removing not needed information.

AFFECTED QUALITY CHARACTERISTICS AND GOALS Long parameter lists may be harder to understand and may contain redundant information. Presentation/Aesthetics, Simplicity, Cohesion/Modular Design → Comprehensibility, Changeability, Correctness

Specialization Aggregation

DESCRIPTION The association is a specialization of another association. This means, that there is a generalization relation between the two involved associations. People are often confused by the semantics of specialized associations. The suggestion is therefore to model any restrictions on the parent association using constraints [119].

EXAMPLE Figure 5.5 shows class *Journey* that is subclassed by class *AirJourney*. Also there is a similar class inheritance hierarchy including classes *Route* and *AirRoute*. Furthermore, there is an association between both subclasses *Journey* and *Route*. This association is also specialized by a corresponding association. In fact, this association hierarchy might be confusing.

DETECTION This smell can be detected by matching a corresponding (anti-) pattern based on the abstract syntax of UML. Figure 5.6 shows such a specification. It defines two UML `Associations` (named *assoc_1* and *assoc_2*) which are related by a corresponding `Generalization` relationship.

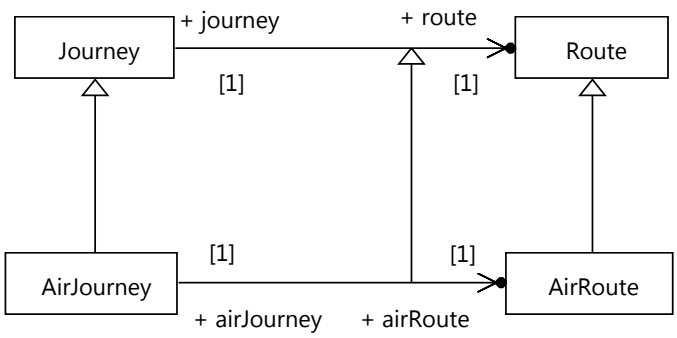


Figure 5.5: Example UML model smell *Specialization Aggregation*

USABLE UML MODEL REFACTORINGS No existing model refactoring can be used to eliminate this smell. Either a new one has to be developed, or the smell has to be eliminated directly, for example by restructuring the model considering this specific aspect.

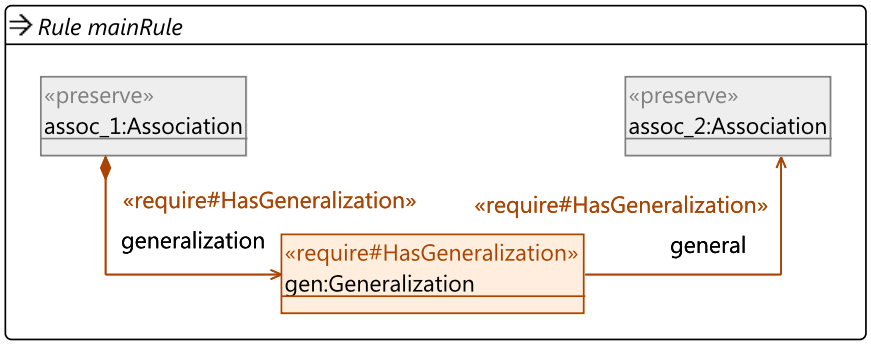


Figure 5.6: Pattern specification of model smell *Specialization Aggregation*

AFFECTED QUALITY CHARACTERISTICS AND GOALS Specialized associations are hard to understand and might represent redundant modeling since involved classes can be already specializations. Simplicity, Redundancy → Comprehensibility

5.2.3 *Affected quality aspects*

Considering a model smell, it is not always clear which quality aspects are affected by this smell. Nevertheless, this section presents a first assignment of selected UML class model smells described in Appendix D to quality aspects presented in Chapter 4. However, these assignments need further consideration in future.

Table 5.4 shows a first assignment of selected UML model smells being suitable in an early stage of a model-based software development process to 6C quality attributes presented in Section 4.2. An entry × in cell (i, j) indicates that UML smell i influences quality

attribute j to some extent. The entries in this table result from the corresponding discussions in the smell descriptions which can be found in Appendix D of this thesis but also in the case study presented in Section 6.1.

	6C Quality Attributes					
UML Model Smell	*Correctness*	*Completeness*	*Consistency*	*Comprehensibility*	*Confinement*	*Changeability*
01. Concrete Superclass	×			×		
02. Data Clumps					×	×
03. Diamond Inheritance			×	×	×	
04. Equally Named Classes	×		×	×		×
05. Large Class	×	×		×	×	
06. Long Parameter List	×			×		×
07. No Specification	×	×	×	×	×	
08. Primitive Obsession	×	×		×	×	
09. Redefined Attribute			×	×	×	×
10. Specialization Aggregation				×	×	
11. Speculative Generality		×		×	×	
12. Unnamed Element		×	×	×		
13. Unused Class	×	×			×	

Table 5.4: Possible impacts of class model smells on 6C quality attributes

The most affected quality attributes in this table are `Confinement` and `Comprehensibility`. This is not surprising since (1) modeling tries to raise the abstraction level in order to be more understandable, and since (2) the UML offers a variety of language features which are only of limited suitability for specific purposes like modeling the problem domain.

Moreover, it seems that an impact on quality attribute `Consistency` induces a potential impact on quality attribute `Comprehensibility`. This is also not surprising since most smells affecting quality attribute `Consistency` address contradictorily modeled facts which are consequently hard to understand.

One technique for improving the quality of a software artifact is *Refactoring*. Refactoring was introduced by Martin Fowler who gives a fit and proper definition in [64] probably being the most cited clause in this field of research:

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

Basically introduced to software code, refactoring has been successfully lifted to the level of software models, especially for (UML) class models being closely related to programmed class structures in object-oriented programming languages such as C++ and Java. For behavior models, the relation between models and code, and therefore the adoption of code refactorings to model refactorings, is less obvious.

It is hard to establish the preserving of the model's behavior since modeling languages such as the UML do not have a formal semantic in general. However, if the modeling languages are used for code generation purposes (e.g., by using the corresponding facilities provided by the Eclipse Modeling Framework (EMF) [44, 144] or the IBM Rational Software Architect [82]) the formal semantics of the target programming language such as Java can be considered instead.

In this section, we first present an overview on refactorings for UML class models discussed in literature. Then we give structured descriptions of selected refactorings. Finally, we discuss relationships between class model refactorings and class model smells presented in the previous section.

5.3.1 *An overview on UML class model refactorings*

We extracted altogether 23 UML class model refactorings from research literature. In Appendix C of this thesis, we present a catalog where each refactoring is presented by its name and a short description. Note that similar to the model smells catalog presented in Appendix B we do not claim that this catalog is complete.

The most refactorings for UML models are adopted from corresponding code refactorings presented by Fowler [64] but in the last decade research also concentrated on UML models in particular (for example, see [30] and [107] for a variety on refactorings for UML class models). Most of them focus on smaller model changes. However, 6 out of 23 refactorings are built up from those existing, so-called atomic refactorings. So, in the catalog presented in Appendix C, we distinguish between atomic and complex refactorings (with `Extract Superclass` being the mostly discussed and therefore the most prominent one). Chapter 7 presents an approach for the specification of

refactoring composition. Furthermore, most of the refactorings discussed in literature come with an inverse refactoring, taking back the original refactoring effect, for example `Pull Up Attribute` and `Push Down Attribute`. Finally, few refactorings are often discussed using slightly different descriptions, i.e., they come up with some variants.

Several researchers also discuss refactorings for behavioral UML models like state machines (18 refactorings) and activity diagrams (2 refactorings). Here, the main sources are [147, 127] (for state machines) and [20] (for activity diagrams and also for state machines).

5.3.2 *Selected UML class model refactorings*

In this section, we describe selected UML class model refactorings (one atomic and two complex ones). We describe these refactorings along a structured definition scheme. For each model refactoring a short description, the contextual meta model element type for applying the refactoring, and the input parameters of the refactoring are given. Furthermore, we present preconditions that have to be checked, either before or after parameter input by the refactoring user, as well as postconditions that specify the behavior preservation of the refactoring. We then specify the transformation that has to be performed after the precondition checks have passed. Finally, an example completes each model refactoring description. Further structured descriptions of UML class model refactorings can be found in Appendix E.

Rename Operation

DESCRIPTION The current name of an operation does not reflect its purpose. This refactoring changes this name [30, 107].

EXAMPLE Figure 5.7 shows a class *Book* owning an operation *gettitle*. Since *camel case* [17] makes it easier to read the operation's name is changed to *getTitle*.

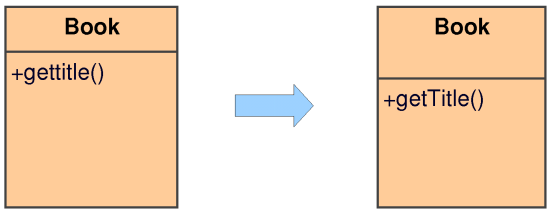


Figure 5.7: Example UML model refactoring *Rename Operation*

CONTEXTUAL ELEMENT `Operation`

INITIAL PRECONDITIONS CHECK There are no initial preconditions that have to be checked.

REFACTORING PARAMETERS `newName` - New name of the contextual operation.

FINAL PRECONDITIONS CHECK (1) There is no operation with name `newName` and with the same parameter list (equal parameter names and types) as the contextual operation in the class owning the contextual operation. (2) There is no operation named `newName` and with the same parameter list (equal parameter names and types) as the contextual operation in the inheritance hierarchy of the class owning the contextual operation.

MODEL TRANSFORMATION Change the name of the contextual operation to `newName`.

POSTCONDITIONS The name of the contextual operation is `newName`.

Extract Superclass

DESCRIPTION There are two or more classes with similar features. This refactoring creates a new superclass and moves the common features to the superclass. The refactoring helps to reduce redundancy by assembling common features spread throughout different classes [141, 106, 150, 107, 160].

EXAMPLE In Figure 5.8 classes *Bike* and *Car* have common attributes and operations. Extract these common features to a new superclass *Vehicle*.

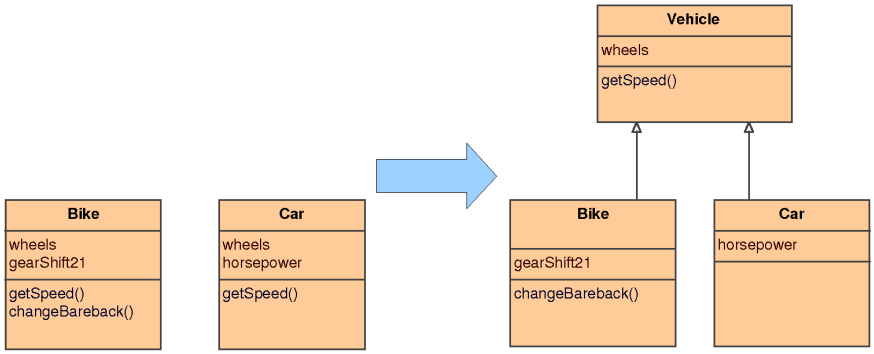


Figure 5.8: Example UML model refactoring *Extract Superlass*

CONTEXTUAL ELEMENTS Set of `Classes`

INITIAL PRECONDITIONS CHECK The contextual classes have similar features, i.e., attributes with the same name, type, visibility and multiplicity, or operations with the same name, visibility and parameter list. Additionally, the initial preconditions of the involved refactorings have to be checked properly.

REFACTORING PARAMETERS Each parameter of refactoring *Create Superclass*. Additionally, a list of attributes and operations which have to be pushed to the new subclass is taken from one contextual class.

FINAL PRECONDITIONS CHECK No final preconditions have to be checked. However, the final preconditions of the involved refactorings have to be checked properly.

MODEL TRANSFORMATION (1) Use refactoring *Create Superclass* on the contextual classes with the given parameters. (2) Use refactoring *Pull Up Property* on each attribute of the appropriate parameter list with the corresponding parameter. (3) Use refactoring *Pull Up Operation* on each operation of the appropriate parameter list with the corresponding parameter.

POSTCONDITIONS In each step the postconditions of the used refactorings have to be checked. No additional postconditions are required.

Introduce Parameter Object

DESCRIPTION There is a group of parameters that naturally go together. This refactoring replaces a list of parameters with one object. This parameter object is created for that purpose [64, 104, 161].

EXAMPLE In Figure 5.9 a date range is used in several operations. Use refactoring *Introduce Parameter Object* to build class *DateRange*.

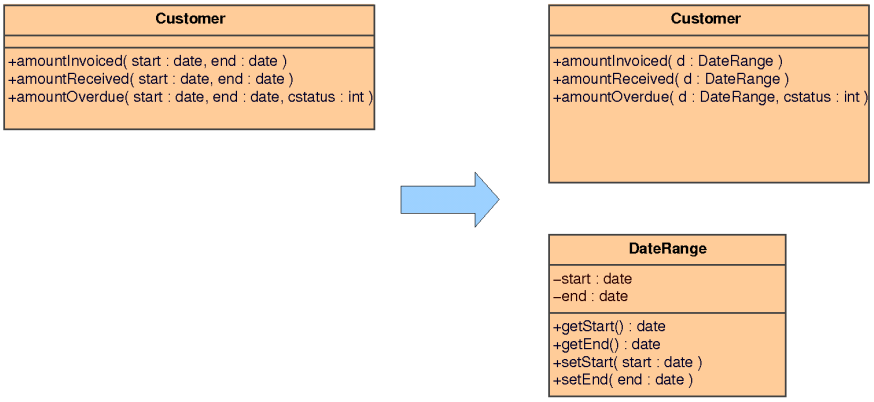


Figure 5.9: Example UML model refactoring *Introduce Parameter Object*

CONTEXTUAL ELEMENTS List of `Parameters`

INITIAL PRECONDITIONS CHECK All contextual parameters belong to the same operation.

REFACTORING PARAMETERS `className` - Name of the new parameter class.
`namespaceName` - Name space of the new parameter class given by a qualified name.

FINAL PRECONDITIONS CHECK There does not already exist a classifier named `className` in the name space named `namespaceName`.

MODEL TRANSFORMATION (1) Create a new class named `className` in the name space named `namespaceName` with default visibility. (2) Create for each contextual parameter a private attribute with getter and setter operations. (3) Replace the parameter list in all operations of the class owning the operation with the contextual parameters with a new parameter with type of the parameter class. Use refactorings *Add Parameter* and *Remove Parameter* for this purpose.

POSTCONDITIONS (1) There is a new class named `className` in the name space named `namespaceName` with default visibility. (2) For each contextual parameter there is a private attribute with getter and setter operations. (3) There is a new parameter with type of the parameter class in all operations of the class owning the operation with the contextual parameters.

5.3.3 *Smell-Refactoring relationships*

Since refactoring is the technique of choice for eliminating model smells, there is a strong coupling between concrete UML smells and refactorings. But the suitability of a certain model refactoring to fix a recognized model smell is not the only relation between these two model quality assurance techniques. Another potential relationship is motivated by the fact that the application of a certain refactoring may cause the occurrence of a specific model smell. This section addresses these relations between UML model refactorings and UML model smells.

Table 5.5 gives an overview on refactoring alternatives to eliminate the 13 UML class model smells presented in Table 5.4 on page 47. An entry $\times$ in cell (i, j) indicates that UML refactoring i can be used to eliminate smell j. The entries in this table are derived from the corresponding discussions in the refactoring descriptions which can be found in Appendix E of this thesis but also in the case study presented in Section 6.1.

	UML Model Smells									
UML Refactorings	*Concrete Superclass*	*Data Clumps*	*Diamond Inheritance*	*Equally Named Classes*	*Large Class*	*Long Parameter List*	*No Specification*	*Primitive Obsession*	*Speculative Generality*	*Unnamed Element*
Create Subclass							×		×	
Extract Associated Class		×			×			×		
Extract Subclass		×			×					
Extract Superclass		×			×					
Inline Class									×	
Introduce Parameter Object		×				×		×		
Move Attribute					×					
Move Operation					×					
Pull Up Attribute					×					
Pull Up Operation					×					
Push Down Attribute					×					
Push Down Operation					×					
Remove Empty Associated Class				×						
Remove Empty Subclass	×		×	×						
Remove Empty Superclass			×	×						
Remove Parameter						×			×	
Remove Superclass			×						×	
Rename Attribute										×
Rename Class				×						×
Rename Operation									×	×

Table 5.5: Positive impacts of UML refactorings on UML model smells

As last topic in this section we discuss 'negative' relations between UML refactoring and UML smells, in which the application of a refactoring may cause the occurrence of a specific smell. A first assignment is given in Table 5.6. An entry in cell (i, j) indicates that refactoring i can cause the occurrence of smell j. Completely new smell occurrences are marked with $\otimes$ whereas smells which already existed before the refactoring but in another context are marked with $\times$.

	UML Model Smells								
UML Refactorings	*Concrete Superclass*	*Data Clumps*	*Diamond Inheritance*	*Equally Named Classes*	*Large Class*	*Long Parameter List*	*No Specification*	*Primitive Obsession*	*Speculative Generality*
Add Parameter		$\otimes$				$\otimes$			
Create Associated Class				$\otimes$					
Create Subclass				$\otimes$					
Create Subclass	$\otimes$		$\otimes$	$\otimes$					
Extract Associated Class		$\times$		$\otimes$	$\times$	$\times$		$\otimes$	
Extract Subclass		$\times$		$\otimes$	$\times$	$\times$		$\otimes$	
Extract Superclass	$\otimes$	$\times$	$\otimes$	$\otimes$	$\times$	$\times$		$\otimes$	
Inline Class		$\otimes$			$\otimes$	$\times$		$\otimes$	
Introduce Parameter Object		$\otimes$		$\otimes$	$\otimes$	$\times$		$\otimes$	
Move Attribute		$\otimes$			$\otimes$				
Move Operation					$\otimes$	$\times$			
Pull Up Attribute		$\otimes$			$\otimes$				
Pull Up Operation					$\otimes$	$\times$			
Push Down Attribute		$\otimes$			$\otimes$				
Push Down Operation					$\otimes$	$\times$			
Remove Empty Subclass							$\otimes$		$\otimes$
Remove Superclass		$\otimes$			$\otimes$	$\times$		$\times$	
Rename Class				$\otimes$					

Table 5.6: Potential negative impacts of UML refactorings on UML smells

6

EXAMPLE APPLICATION CASES

In this chapter, we present three example cases performing the model quality assurance process presented in Chapter 3. The example cases serve as proof-of-concept implementations of this process and show its applicability, its flexibility, and hence its effectiveness. The first example case, a classical model-based software development scenario, uses the Unified Modeling Language (UML) [123] for modeling the problem domain in an early phase of a software development process. In contrast to the use of a General Purpose Language (GPL) in this example case, the subsequent example cases show the applicability of the quality assurance process on models of Domain Specific Languages (DSLs). The second example case uses a textual DSL whose models serve as main artifacts in a modern model-driven process for developing simple web applications. In the third example case, we specify quality assurance techniques for rule-based, in-place model transformation systems which are used for refactoring specification, for example.

6.1 QUALITY ASSURANCE OF UML CLASS MODELS

In this section, we discuss a simple example of the process presented in Chapter 3 concerning UML class modeling in early phases of a model-based software development process. After presenting the scenario, we describe the definition and application of a sample model quality assurance process in detail.

6.1.1 *Scenario description*

In our example, we consider a software project for the development of an accounting system for a vehicle rental company. This company has a headquarter and owns cars, trucks, and motorbikes which can be rented by customers via a vehicle rental service. These three kinds of vehicles have some common and some differing properties. A car has a manufacturer, a registration number, an engine power, and a number of seats. A truck has a manufacturer, a registration number, an engine power, and a weight. Finally, a motorbike has a manufacturer, a registration number, an engine power, and a cylinder capacity. Each

customer has a name and an email address and is related to a consultant being an employee of the company. Furthermore, the company has some subcontractors being specific employees and customers.

We assume that software models are used in the domain analysis phase in order to get an overview on real world entities in the problem domain. The modeling of the problem domain is done using UML class models.

Figure 6.1 shows a first UML example model that has been developed in an early stage of the problem analysis. Here, the example model is displayed in concrete syntax using the UML CASE tool IBM Rational Software Architect (RSA) [82].

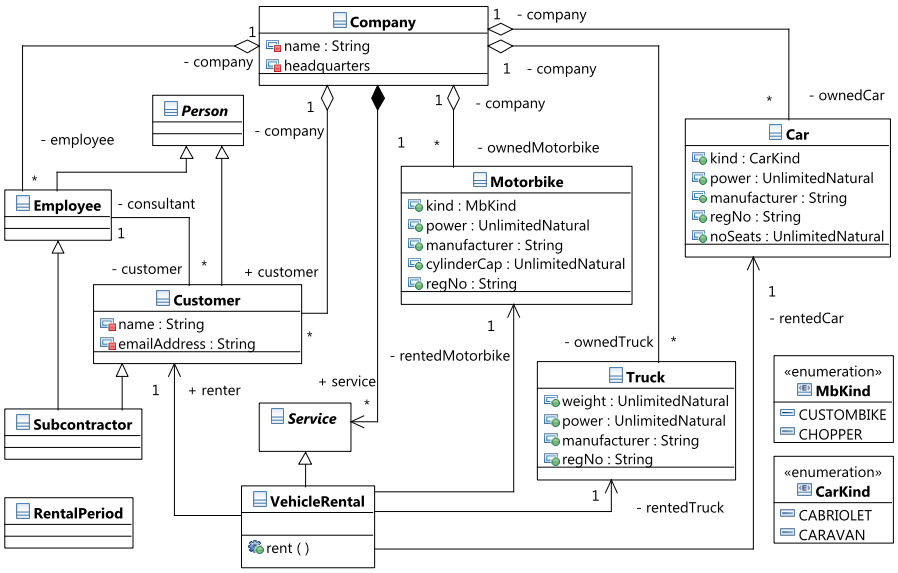


Figure 6.1: Example UML class model showing the first version of domain model *Vehicle Rental Company* (before model review)

Concerning quality issues, the model contains several suspicious parts. For example, information on the vehicles (cars, trucks, and motorbikes) is modeled redundantly (such as *power*). Furthermore, class `RentalPeriod` is not associated to any other class at all (which hints at some incompleteness). During a model review (see Figure 3.1 on page 17) this initial model is analyzed in terms of project-specific model metrics and model smells. Several refactorings in combination with additional model changes are applied subsequently.

Figure 6.2 shows the improved model after this review. The afore mentioned redundancies have been eliminated and incomplete model parts have been supplemented with further information. We discuss the concrete applied techniques, i.e., calculated metrics, detected model smells, and applied model refactorings, in the following sections.

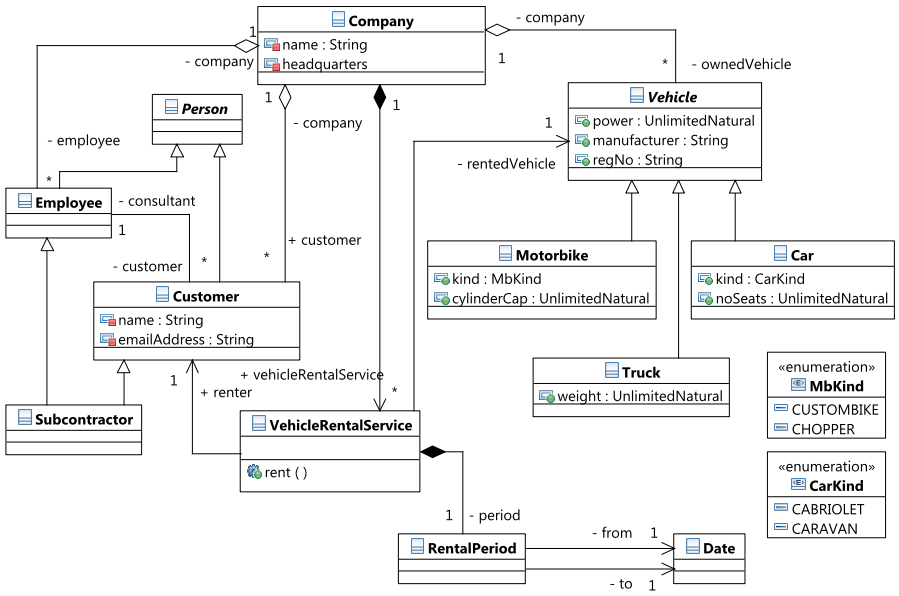


Figure 6.2: Improved sample UML class model after model review

6.1.2 *Specification of quality assurance techniques*

In this section, we demonstrate how the specification process for the used model quality assurance techniques (see Figure 3.2 on page 18) is applied along our example. Please note that this process must not be applied for each individual project in its full extent. Once these techniques are defined they can be reused in future projects as well.

Specification of relevant model quality aspects

In our example, we use the quality model described in Section 4.3 which is based on the 6C quality goals presented in Section 4.2.1 and determine those aspects which are most relevant as follows.

The most important property of a domain analysis model is that it models the problem domain in the right way, i.e., choosing the right elements and making correct statements on the domain. So, 6C goal *Correctness* is an essential quality aspect that has to be considered when applying a model quality assurance process. Since an analysis model is used for communicating with problem domain experts who are typically inexperienced in software modeling, it is also important that the model is easily understandable. This implies that the model must not contain any obvious ambiguity. Furthermore, the analysis model must not have unnecessary information that make it more complex as necessary. So, 6C goals *Comprehensibility*, *Consistency*, and *Confinement* can be seen as essential quality aspects.

Since the modeling purpose in our example is to get an overview on the problem domain, it is justifiable if less important information

is missing. So, 6C goal *Completeness* is a less important quality aspect in our example. Furthermore, since the domain to be modeled is very simple and manageable, model reviewers do not have to prioritize the quality goal *Changeability*.

Please note that we are arguing from a selective point of view only to keep the argumentation compact. However, the selection of the main quality aspects may vary depending on the intended modeling purpose. This demonstrates the complexities and challenges of this basic task.

Formulation of questions leading to static quality checks

After having specified relevant quality aspects we have to think about how to check the compliance with these aspects during the concrete modeling activity. This is done by formulating questions that lead to model smells. These questions have to be formulated in a way that they can be answered by static model analysis, i.e., they need answers which can be given by the model syntax only. In the following, we concentrate on one single quality aspect, namely *Confinement*, and present a selection of appropriate questions. Example questions are:

Q1: *Are there classes being not used by any other model element?* This is a typical case of unnecessarily modeled information.

Q2: *Are there classes inheriting from another class several times?* This would indicate that the modeler uses the inheritance concept in a too complex way, i.e., the model is more detailed than necessary.

Q3: *Are there abstract classes not doing much?* Again, this might be an indicator for unnecessary information within the model.

Q4: *Are there at least three similar attributes staying together in more than one class?* This might be a hint that the modeler does not use the inheritance concept of the UML which might be more suitable in this case.

Q5: *Are there attributes redefining other ones within the inheritance hierarchy?* Since the purpose of the model is to get an overview about the problem domain the use of this language construct might be too complex, i.e., it does not suit to the modeling purpose.

Further questions that consider other quality aspects are:

Q6: *Are there equally named classes owned by different packages?* Equally named classes could lead to misunderstandings of the modeled aspects (aspect *Comprehensibility*). If they are representing the same real world entity the modeler uses some kind of controlled redundancy that in turn can compromise quality aspect *Changeability*.

Q7: *Are there abstract classes that are subclasses of non-abstract classes?* This would be an example for incorrect modeling, i.e., it compromises quality aspect *Correctness*.

Q8: *Are there abstract classes that are not specialized by at least one concrete class?* This might indicate that there is something missing in the model, i.e., quality aspect *Completeness* is violated.

Specification of project-specific UML smells

The questions formulated in the previous section lead to model smells that hint at model parts possibly violating the quality aspect *Confinement*. A structured definition of each smell including a name, the corresponding question, an informal description, an example, affected 6C quality goals, and ways to detect the smell can be found in Appendices B and D of this thesis. The derived UML smells are:

UNUSED CLASS (DERIVED FROM QUESTION Q1): An unused class often stands alone in the model without any references to other classes. This smell is adapted from Riel who analyzed object-oriented design [133] and can be detected by two different mechanisms. First, we can define the absence of child classes, associated classes, and attributes with class type as anti-patterns based on the abstract syntax of UML and check whether they do not match on a concrete instance class. Second, we can define a constraint that uses three metrics (*Number of direct children*, *Number of associated classes*, and *Number of times the class is externally used as attribute type*) and that checks whether each metric is evaluated to zero. Nevertheless, the former alternative seems to be the most appropriate one.

DIAMOND INHERITANCE (QUESTION Q2): This smell is based on the multiple inheritance concept of UML. It occurs when the same predecessor is inherited by a class several times. It is known in literature as 'diamond' inheritance problem for object-oriented techniques using multiple inheritance and was first discussed by Sakkinen [136]. An adequate mechanism to detect this smell is to specify a corresponding pattern on the abstract syntax of UML and to find matches in concrete UML instances.

SPECULATIVE GENERALITY (QUESTION Q3): If there is an abstract class inherited by one single class only, this smell is found. It is based on the corresponding code smell introduced by Fowler [64] and refined by Zhang et al. [161]. To detect this smell we can check whether metric *Number of direct children* evaluates to 1 on an arbitrary UML class. Of course, the corresponding constraint must check whether this class is abstract. Furthermore, it is possible to specify this smell by a corresponding pattern based on

the abstract syntax of UML and try to match this pattern on classes of a concrete UML instance model.

DATA CLUMPS (QUESTION Q4): A UML model holds this smell if interrelated data items often occur as 'clump'. More precisely, this smell can be defined as follows:

- At least three attributes stay together in more than one class.
- These attributes should have the same signatures (same names, same types, and same visibility).
- The order of these attributes may vary.

Again, this smell is also based on the corresponding code smell introduced by Fowler [64] and refined by Zhang et al. [161]. To detect this smell there must be a mechanism to detect similarities in UML models. This is due to the fact that one can not predict how many attributes are involved in this smell. Furthermore, there might be variants wrt. similar attributes when using a more general definition of this smell than here (think of attribute names that need not to be equal but just similar or attributes with different visibilities). Another possibility to detect this smell is to define a metric for an UML class counting all equal attributes with other classes. Nevertheless, using a strict definition with exactly three attributes and equal signatures it is possible to define this smell as pattern based on the abstract syntax of UML.

REDEFINED ATTRIBUTE (QUESTION Q5): UML allows for redefining attributes owned by ancestor classes. However, using this language feature could lead to misunderstandings of the modeled aspect and might be confusing for model readers. It can be checked by matching a corresponding pattern or by evaluating metric *Number of redefined attributes* to zero.

Table 6.1 summarizes the impact of the UML smells described above on 6C quality aspects. An entry $\times$ in cell (i, j) indicates that UML smell i influences quality goal j to some extent. The entries in this table result from the corresponding discussions in the smell descriptions from above. Furthermore, the table contains three model smells that can be deduced from questions Q6 to Q8. Here, UML smells *Equally named Classes* and *Concrete Superclass* are based on the thesis from Christian Lange [97] who calls them defects, whereas smell *No Specification* is adapted from the corresponding object-oriented design smell described by Riel [133].

	Correctness	Completeness	Consistency	Comprehensibility	Confinement	Changeability
Unused Class	×	×			×	
Diamond Inheritance			×	×	×	
Speculative Generality		×		×	×	
Data Clumps					×	×
Redefined Attribute			×	×	×	×
Equally Named Classes	×		×	×		×
Concrete Superclass	×		×	×		
No Specification	×	×	×	×	×	

Table 6.1: Possible impacts of UML model smells on 6C quality attributes

Specification of project-specific UML refactorings

After having specified appropriate model smells as done in the previous section, suitable refactorings have to be defined in order to support the handling of 'smelly' models. Table 6.2 gives an overview on refactoring alternatives to eliminate the UML smells presented above. An entry × in cell (i, j) indicates that UML refactoring i can be used to eliminate smell j.

To eliminate the *Unused Class* smell no single suitable refactoring can be deduced since one can not determine automatically whether this class is either useless or if there are some missing relationships. So, this smell can either be eliminated by removing the class (i.e., by using the simple refactoring *Remove Unused Class*) or by adding further information to the model not indicated as refactorings.

Smell *Diamond Inheritance* can be eliminated by applying refactorings *Remove Superclass* or *Remove Intermediate Superclass*. Both refactorings can also be used to eliminate UML smell *Speculative Generality*. Here, the unnecessarily modeled abstract class has to be removed by one of those refactorings, depending on whether this class has a parent class or not. A further applicable refactoring addresses missing information, more precisely missing subclasses of the abstract class. This refactoring is called *Extract Subclass*. It creates a new subclass and applies refactoring *Push Down Attribute* to a set of attributes of the contextual class (which is empty in our case).

	Unused Class	Diamond Inheritance	Speculative Generality	Data Clumps	Redefined Attribute
Extract Class				×	
Extract Superclass				×	
Extract Intermediate Superclass				×	
Extract Subclass			×		
Remove Superclass		×	×		
Remove Intermediate Superclass		×	×		
Remove Redefined Attribute					×
Remove Unused Class	×				

Table 6.2: Suitable refactorings to erase specific UML model smells

The *Data Clumps* smell can be removed in two different ways: either by moving the corresponding attributes to a new associated class or by moving them to a new class that is a common superclass of the owning classes. The first option uses UML refactoring *Extract Class* that internally uses refactorings *Create Associated Class* and *Move Attribute*. The second alternative uses either refactoring *Extract Superclass* or *Extract Intermediate Superclass* if the owning classes have a common superclass already. Besides the creation of an empty (intermediate) superclass, both refactorings use refactoring *Pull Up Attribute* to move equal attributes to this newly created class.

Last but not least, UML smell *Redefined Attribute* can be eliminated using refactoring *Remove Redefined Attribute* that removes the redefinition relationship as well as the contextual attribute if and only if the redefined attribute is visible to the owning class of the redefining attribute.

In Appendix E, you find a structured definition of each UML refactoring including a name, a short description, an illustrating example, the contextual meta model element for applying the refactoring, and the input parameters. Furthermore, we use a three-part specification scheme reflecting a primary application check for a selected refactoring without input parameters, a second one with parameters, and the proper refactoring execution steps. Please note that some of the UML

refactorings are adapted from corresponding UML refactorings, for example discussed in [150], [160], and [107].

As last topic in this section we discuss relations between UML refactoring and UML model smells. Inter-relations are presented in Table 6.3. An entry in cell (i, j) indicates that UML refactoring i can cause the occurrence of UML smell j.

	Unused Class	Diamond Inheritance	Speculative Generality	Data Clumps	Redefined Attribute	Equally Named Classes	Concrete Superclass	No Specification
Extract Class				×		⊗		
Extract Superclass		⊗		×		⊗	⊗	
Extract Intermediate Supercl				×		⊗	⊗	
Extract Subclass	⊗		⊗	×		⊗		
Remove Superclass				⊗				
Remove Intermediate Supercl				⊗				
Remove Redefined Attribute	⊗							
Remove Unused Class								

Table 6.3: Possible impacts of UML refactorings on UML model smells

Each *Extract ... Class* refactoring may cause UML smell *Data Clumps* if appropriate attributes are moved to the newly created class. Please note that this smell already existed before the refactoring but in another context (without the newly inserted class). We mark this kind of smell with × whereas completely new smell occurrences are marked with ⊗. Furthermore, smell *Data Clumps* can also be introduced by refactorings *Remove Superclass* and *Remove Intermediate Superclass* when moved attributes complete an equivalent set of attributes in some subclasses.

The application on refactoring *Extract Superclass* can introduce smell *Diamond Inheritance* to the model if the contextual classes from which the new superclass shall be extracted have a common subclass already. Furthermore, it can introduce smell *Concrete Superclass* if it is applied on an abstract class. This could be avoided by restricting this refactoring on concrete classes only. For the same reason, smell *Concrete Superclass* can be introduced by refactoring *Extract Intermediate Superclass* as well.

Refactoring *Extract Subclass* can lead to an used class if no attribute is pushed down to the new class. Furthermore, if this refactoring is applied on an abstract class that is not inherited so far UML smell *Speculative Generality* is introduced.

Except for smell *Data Clumps* there is no UML smell (in the set of analyzed smells) that can be introduced by refactorings *Remove Superclass* and *Remove Intermediate Superclass*, respectively. Refactoring *Remove Redefined Attribute* can lead to an unused class if the type class of the removed attribute has been the only use of this class in the model. Finally, refactoring *Remove Unused Class* does not cause any smell from the analyzed list.

6.1.3 *Application of quality assurance techniques*

In this section, we discuss the application of our quality assurance process on the UML class model presented in Figure 6.1 in detail.

Metrics calculation and interpretation

For the first overview on a model, a report on project-specific model metrics might be helpful. In our example, we calculate model metrics for UML packages concerning abstractness and inheritance issues. Within the package depicted in Figure 6.1 there are altogether 11 classes (9 concrete and two abstract classes). The concrete classes own altogether 20 attributes from which 2 are inherited from parent classes (attributes *name* and *emailAddress* of class `Subcontractor`).

Three metrics are calculated using these 'basic' metrics. The abstractness (`A`) of the package is 0.18 (ratio between the number of abstract classes and the total number of classes in the package), the attribute inheritance factor (`AIF`) is 0.10 (ratio between the number of inherited attributes in all concrete classes in the package and the total number of attributes in all concrete classes in the package), and the average number of attributes in concrete classes within the package (`AvNAtP`) is 2.22. As a first evaluation of these metrics results, one can state that the model might not be complete since (1) there are only 11 classes modeled for the vehicle company domain, and (2) these classes have little more than two attributes on average. Furthermore, language concepts of abstractness and inheritance are not used too exhaustively. So the model is less complex and easier to understand. On the other hand, the low values of `A` and `AIF` can be interpreted as a hint that the modeling purpose is not yet achieved since the modelers use the provided language features insufficiently only.

Smell detection and interpretation

The discussion of metrics results shows that a manual interpretation of metric values seems to be unsatisfactory and error-prone. So, an-

other static model analysis technique is required, more precisely an automatic detection of specific smells for UML models.

Some smells can be found when looking for specific patterns defined on the abstract model syntax, other model smells are based on corresponding metrics. For a metric-based model smell, an appropriate threshold can be configured. In our example, we consider UML smell *Data Clumps* as metric-based smell. It relies on metric `NEAC` (number of equal attributes with further classes) and comparator $\geqslant$ (greater or equal). We set the limit for smell *Data Clumps* to `3`, i.e., this smell occurs if a class owns more than two attributes with same name, type, and visibility in at least one other class.

Analyzing the example UML model shown in Figure 6.1, the smell detection analysis discovers the existence of altogether six concrete smells which affect quality aspect *Confinement*. Smell *Data Clumps* occurs three times, more concretely in classes `Car`, `Truck`, and `Motorbike`. Smell *Diamond Inheritance* occurs once. Here, the involved elements are classes `Person`, `Employee`, `Customer`, and `Subcontractor`. Another detected smell is *Speculative Generality* since abstract class `Service` has one single child class only. Furthermore, there is the unused class `RentalPeriod`.

The next step during a model review is to interpret the results of the smell detection analysis. Potential reactions on detected smells are:

- Use refactoring *Extract Superclass* on classes `Car`, `Truck`, and `Motorbike` to insert a common parent class `Vehicle` and pull up attributes *manufacturer*, *power*, and *regNo* to it.
- The diamond inheritance smell detected on class `Subcontractor` should not be eliminated since this seems to be an important detail that has to be addressed in the domain model.
- Smell *Speculative Generality* should be removed by using refactoring *Remove Superclass* on class `Service` since the company does not offer further services.
- Class `RentalPeriod` is unused up to now. It should be associated to class `VehicleRental` and shall refer a new class `Date` twice (named *from* and *to*).

Refactoring application and manual model changes

Besides manual changes, model refactoring is the technique of choice to eliminate occurring smells. Figure 6.3 shows our example UML model after performing several model changes, being refactorings and manual changes, as described at the end of the last section. Now, classes `Car`, `Truck`, and `Motorbike` have a common superclass `Vehicle` owning the afore redundant attributes *manufacturer*, *power*, and *regNo*.

Class `Service` has been removed so that `VehicleRentalService` is the only offered service left. Finally, class `RentalPeriod` has been completed by additional information, i.e., class `Date` and associations *period*, *from*, and *to*.

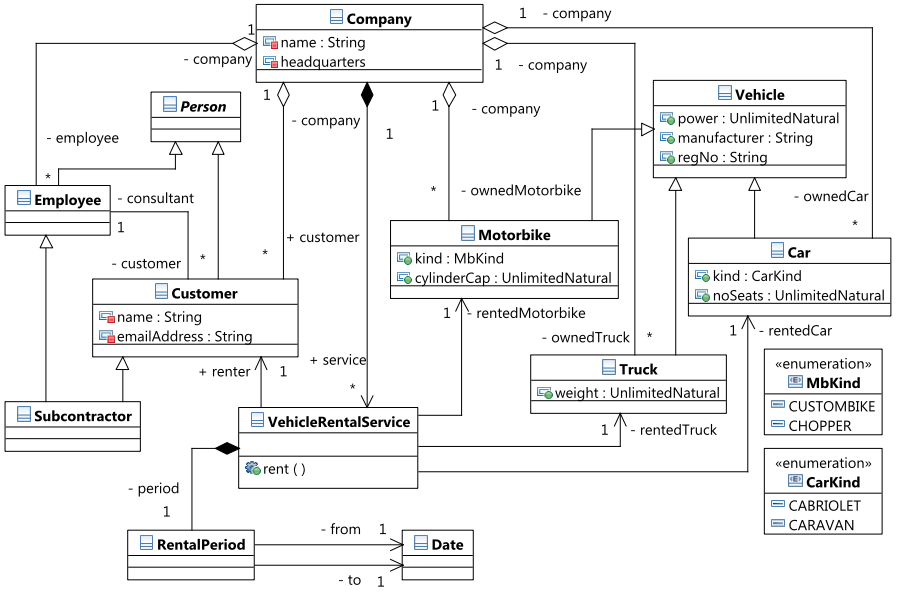


Figure 6.3: Example UML class model after several model changes during a first model review

From the detected smell occurrences only one is left (smell *Diamond Inheritance* in class hierarchy `Subcontractor` ⇒ `Person`). Nevertheless, there are model parts remaining suspicious with respect to several model quality aspects. For example, there are two elements indicating incorrect modeling. First, class `Vehicle` is concrete even though it should represent a generic term for concrete vehicle kinds, hence should be abstract. Moreover, the association between classes `Company` and `VehicleRentalService` has a too general name and should be named *vehicleRentalService* instead. Furthermore, there are associations from class `Company` to classes `Car`, `Truck` and `Motorbike` respectively from class `VehicleRentalService` to these classes hinting to some kind of redundant modeling.

The former discussion shows that project-specific model quality assurance techniques do not have to be completely defined before a project starts. In our example, the quality assurance process should be adapted during the model development phase in order to be steadily improved. UML smells *Concrete Superclass* and *Association Clumps* as well as UML refactorings *Rename Association* and *Pull Up Association* would extend the suite of project-specific model quality assurance techniques in a meaningful way.

6.2 QUALITY ASSURANCE OF TEXTUAL MODELS FOR THE DEVELOPMENT OF SIMPLE WEB APPLICATIONS

In this section, we present the design and implementation of an example case for quality assurance of textual models. As example language we take a domain-specific modeling language (DSML) called Simple Web Model (SWM) for defining a specific kind of web applications in a platform-independent way[1]. In this example, we concentrate on quality aspect *Completeness*. This means that we analyze SWM models whether they are ready for code generation and improve model parts using domain-specific refactorings. After presenting an example scenario, we describe definition and application of a sample model quality assurance process for SWM models in detail.

6.2.1 *Motivation and scenario description*

The use of (often textual) DSMLs is a promising trend in modern software development processes to overcome the drawbacks concerned with the universality and the broad scope of general-purpose languages like the Unified Modeling Language (UML) [123]. Such a DSML can help to bridge the gap between a domain experts view and the implementation. Often, a DSML comes along with a code generator and/or interpreter to provide functionality that should be hidden from the domain expert. In the generator case, high code quality can be reached only if the quality of input models is already high.

In this example case, we assume the following scenario (taken from [21]): A software development company is repeatedly building simple web applications being mostly used to populate and manage persistent data in a database. Here, a typical three-layered architecture following the Model-View-Controller (MVC) pattern [68] is used. As implementation technologies, a relational database for persisting the data as well as plain Java classes for retrieving and modifying the data are employed for building the model layer. Apache Tomcat is used as Web Server. The view layer, i.e., the user interface, is implemented based on JavaServer Pages and the controller layer is implemented based on Java Servlets. The company decided to develop its own textual DSML called Simple Web Modeling Language (SWM) for defining their specific kind of web applications in a platform-independent way. Furthermore, platform-specific models following the MVC pattern should be derived with model transformations from which the Java-based implementations are finally generated.

The SWM language is defined as follows. A `WebModel` consists of two parts: a `DataLayer` for modeling entities which should be persisted in the database, and a `HypertextLayer` presenting the web pages of the application. An `Entity` owns several `Attributes` (each

1 Several variations of SWM are used in literature, for example in [21].

having a SimpleType) and can be related to several other entities (see meta class Reference). A Page is either a StaticPage having a static content or a DynamicPage having a dynamic content depending on the referenced entity. An IndexPage lists objects of this entity whereas a DataPage shows concrete information on a specific entity like its name, attributes, and references. Pages are connected by Links.

```
1  WebModel := 'webmodel' Name '{'
2          DataLayer
3          HypertextLayer
4      '}' .
5  DataLayer :=  'data {'
6          Entity*
7      '}' .
8  Entity :=  'entity' Name '{'
9          Attribute*
10         Reference*
11     '}' .
12 Attribute := 'att' Name ':' SimpleType .
13 Reference := 'ref' Name ':' (Entity) .
14 HypertextLayer := 'hypertext {'
15         Page+
16         'start page is' (StaticPage)
17     '}' .
18 Page := StaticPage | DynamicPage .
19 StaticPage := 'static page' Name '{'
20         Link*
21     '}' .
22 DynamicPage := IndexPage | DataPage .
23 IndexPage :=  'index page' Name [ 'shows entity' (Entity) ]  '{'
24         Link*
25     '}' .
26 DataPage :=  'data page' Name [ 'shows entity' (Entity) ]  '{'
27         Link*
28     '}' .
29 Link :=  'link to page' (Page) .
30 Name := Letter+
31 Letter := 'A' | ... | 'Z' | 'a' | ... | 'z' .
32 SimpleType := 'Boolean' | 'Email' | 'Integer' | 'String' .
```

Listing 6.1: Grammar of the SWM language in EBNF

Listing 6.1 shows the grammar of the SWM language in Extended Backus-Naur Form (EBNF) [159]. The grammar owns altogether 15 production rules, e.g., for WebModel, Entity, and DynamicPage. Language terminals are defined using inverted commas (like *'index page'*). Optional parts are specified in squared brackets whereas round brackets represent cross-references to already existing language constructs (for example, '(Entity)' refers to an already existing instance of an entity in the model). The logical *or* is encoded by '|' whereas oper-

ations '+' and '*' mean repeat 1 or more times, or 0 or more times, respectively.

6.2.2 *Specification of quality assurance techniques*

In this section, we use the specification process for quality assurance techniques concerning textual models of the SWM language as presented in Section 3.2.2.

Since in our scenario platform-specific models should be derived from SWM models and should be used to generate the Java-based implementations, the major quality aspect to be fulfilled on SWM models is *Completeness*. A model is complete if it contains all relevant information, and if it is detailed enough to serve the modeling purpose [116]. This means for SWM models that (A) on the data layer each entity must contain all relevant attributes and references to other entities whereas (B) the hypertext layer must contain a complete set of (potentially linked) pages which should be part of the web application. Potential SWM model smells violating quality aspect *Completeness* are:

EMPTY ENTITY The entity does not have any attributes or references to other entities. (This violates completeness issues of type A.)

NO DYNAMIC PAGE The entity is not referenced by a dynamic page to be depicted in the web application. (type B)

MISSING DATA PAGE The entity is referenced by an index page but not by a data page. (type B)

MISSING INDEX PAGE The entity is referenced by a data page but not by an index page. (type B)

UNUSED ENTITY The entity is referenced neither by a dynamic page nor by another entity. (types A and B)

MISSING LINK The index page is not linked by the start page of the web application. (type B)

Further SWM model smells violating quality aspects *Correctness* and *Confinement* because of redundantly modeled parts are:

MULTIPLE LINK DEFINITIONS The page has multiple links to the same destination page.

EQUALLY NAMED PAGES There are pages within the hypertext model having the same name.

In addition to the model smells described above, several metrics can be used to analyze completeness of SWM models. For example,

metrics *Number of Entities in the Model (NEM)* and *Number of Dynamic Pages in the Model (NDPM)* can be used to get a first overview on the model structure. Here, a ratio between the values of these metrics less than 1 : 2 might be a hint for missing dynamic pages, i.e., one entity should be referenced by two dynamic pages - both an index page and a data page. Similarly, metrics *Average number of Attributes (resp. References) in Entities of the Model (AvNAE resp. AvNRE)* are useful to detect missing information in the data layer.

After having specified appropriate model smells, suitable refactorings have to be defined in order to support the handling of 'smelly' SWM models. Smells *No Dynamic Page* and *Unused Entity* can be eliminated by a refactoring which inserts both an index page and a data page referencing the corresponding entity to the hypertext layer (refactoring `Insert Dynamic Pages`). A missing index page referencing the same entity as an existing data page can be inserted by refactoring `Add Index Page to Data Page`. Similarly, refactoring `Add Data Page to Index Page` inserts a new data page to the hypertext layer that references the same entity as an existing index page.

To eliminate smell *Missing Link* an appropriate refactoring `Update Links to Index Pages` can be used. This ensures that the start page owns links to all index pages of the model. However, there is no adequate refactoring to eliminate smell *Empty Entity*. Here, manual model changes should be performed (such as deleting the entity or adding attributes or references, respectively).

The smells violating quality aspects *Correctness* and *Confinement* because of redundantly modeled parts can be simply eliminated using refactorings `Remove Multiple Links from Page` (smell *Multiple Link Definitions*) and `Rename Page` (smell *Equally Named Pages*).

6.2.3 *Application of quality assurance techniques*

In this section, we demonstrate how the techniques presented in the previous section are applied to a concrete SWM instance model. We now assume that the software company has to develop a web application for the rental system of a vehicle rental company. Listing 6.2 shows a first SWM model being developed in an early stage of the development process. This model is the object of interest in the model review described in the remainder of this section.

For a first overview, a report on project-specific model metrics might be helpful. In our example model, metrics NEM and NDPM (see previous section) evaluate to 4 and 3, respectively. This means that there are more entities in the web model than dynamic pages hinting at potentially missing dynamic pages. Moreover, the extremely low values of metrics AvNAE and AvNRE (1.25 and 0.25, respectively) are hinting at some missing information within the data layer of the model.

To make this problems more explicit (and thus more obvious), an analysis with respect to so-called model smells representing model parts to be improved can be performed. In our example model shown in Listing 6.2, there are altogether six concrete smell occurrences which should be investigated in detail.

```
webmodel VehicleRentalCompany {
    data {
        entity Customer {
            att name : String
            att email : Email
            ref address : Address
        }
        entity Address {
            att street : String
            att city : String
        }
        entity Car {
            att type : String
        }
        entity Agency {
        }
    }
    hypertext {
        index page carindex shows entity Car {
            link to page cardata
        }
        data page cardata shows entity Car {
        }
        index page agencyindex shows entity Agency {
        }
        static page indexpage {
            link to page carindex
            link to page agencyindex
            link to page carindex
        }
        start page is indexpage
    }
}
```

Listing 6.2: Example SWM instance showing the first version of model Vehicle Rental Company (before model review)

Two entities are not referenced by a dynamic page. This leads to two occurrences of model smell *No Dynamic Page* on entities `Customer` and `Address`. Moreover, entity `Customer` is not referenced by another one leading to smell *Unused Entity*. Entity `Agency` is also involved in two smell occurrences. On the one hand it is referenced by an index page only (smell *Missing Data Page*). This seems to be well since entity `Agency` does not have any attributes or references to be depicted within a data page. One the other hand, the absence of any attributes

and references lead to smell *Empty Entity*, of course. Finally, the start page `indexpage` owns two links to index page `carindex` resulting in smell *Multiple Link Definitions*.

Besides manually changing the model, refactoring is the technique of choice to eliminate occurring smells. In our example, we can use refactoring *Insert Dynamic Pages* to eliminate smell *No Dynamic Page* on entity *Customer*. Please note that we do not eliminate smell *No Dynamic Page* on entity *Address* since this entity is referenced by entity *Customer*, i.e., it is part of this entity. The result of refactoring *Insert Dynamic Pages* is shown in Listing 6.3. Two dynamic pages (an index page and a data page) referencing entity *Customer* are inserted into the hypertext layer of the model. Furthermore, the inserted data page is linked by the index page which is in turn linked by the static page named *indexpage* being the starting page of the hypertext layer.

```
1  webmodel VehicleRentalCompany {
2      ...
3      hypertext {
4          ...
5          static page indexpage {
6              link to page carindex
7              link to page agencyindex
8              link to page customerindex }
9          data page customerdata shows entity Customer { }
10         index page customerindex shows entity Customer {
11             link to page customerdata }
12         ...
13     }   }
```

Listing 6.3: Inserted and changed model elements after applying refactoring *Insert Dynamic Pages*

In order to eliminate smell *Empty Entity* we add a new attribute `address` to entity `Agency`. Afterwards, the smell *Missing Data Page* should also be eliminated using refactoring *Add Data Page to Index Page* on index page `agencyindex`. Here, a new data page referencing entity `Agency` is inserted into the hypertext layer and the inserted data page is linked to the contextual index page `agencyindex`.

Obviously missing information within the data model (detected by low AvNAE and AvNRE values) is added to the model: entity `BankAccount` with attributes `number`, `bankCode`, and `bankName`, attribute `account` to entity `Customer`, attribute `postalcode` to entity `Address`, and finally attributes `manufacturer` and `power` to entity `Car`.

Last but not least, we use refactoring *Remove Multiple Links from Page* to eliminate smell *Multiple Link Definitions* on the starting page which is renamed to `startingpage` since name `indexpage` could lead to misunderstandings because of the key words *index page* within the SWM grammar. Listing 6.4 shows the improved resulting model after the model review.

```
webmodel VehicleRentalCompany {
    data {
        entity Customer {
            att name : String
            att email : Email
            ref address : Address
            ref account : BankAccount
        }
        entity Address {
            att street : String
            att postalCode : Integer
            att city : String
        }
        entity BankAccount {
            att number : Integer
            att bankCode : String
            att bankName : String
        }
        entity Car {
            att manufacturer : String
            att type : String
            att power : Integer
        }
        entity Agency {
            ref address : Address
        }
    }
    hypertext {
        index page carindex shows entity Car {
            link to page cardata
        }
        data page cardata shows entity Car {   }
        index page agencyindex shows entity Agency {
            link to page agencydata
        }
        data page agencydata shows entity Agency {   }
        index page customerindex shows entity Customer {
            link to page customerdata
        }
        data page customerdata shows entity Customer {   }
        static page startingpage {
            link to page agencyindex
            link to page carindex
            link to page customerindex
        }
        start page is startingpage
    }
}
```

Listing 6.4: Example SWM instance of model Vehicle Rental Company (after model review)

6.3 Quality assurance of rule-based in-place model transformation systems

In many model transformation approaches, model transformations are software models themselves. Consequently, model transformation is a suitable scenario for adapting the structured process for specifying model quality assurance techniques presented in Section 3.2.2. Following this process, we identify quality aspects for model transformation systems and introduce suitable smells (potential indicators of low quality) and refactorings that make existing knowledge explicit about how to write model transformation systems. As a result, this section provides a first collection of useful quality assurance techniques, especially refactorings, for rule-based in-place model transformation systems. To integrate these refactorings into a systematic quality assurance process, we further discuss quality aspects for model transformation systems and define a first collection of smells based on metrics and patterns.

6.3.1 *Background and core transformation concepts*

Model transformations have been applied to solve various tasks in model-driven engineering (MDE) such as model refactoring and optimizations, translation into other modeling languages, simulation and analysis, model migration and code generation [156].

While model translations are typically *out-place*, i.e., constructing new result models, endogenous model transformations (sticking to one language) may also be *in-place*, i.e., modifying the input model directly [28]. Model simulation and refactoring as well as other kinds of model modifications such as further model optimizations and advanced editing operations are typically realized by in-place transformations. Note that we consider refactorings of in-place model transformations in this section, since our refactorings do not refer to either source or target model elements only. Refactorings of out-place model-to-model transformations are presented in detail in [158]. Nevertheless, we could also apply our techniques and tool to out-place (model-to-model) transformations, which can be emulated by considering an integrated domain model constructed from the source and target domain, and defining a rule set where only target domain model elements are generated [37].

The core concepts of rule-based in-place model transformation approaches form the basis for our catalogs of smells and refactorings. Of course, taking further concepts into account, the corresponding transformation language is widened and the catalogs shall be extended accordingly. Transformation languages offering (most of) these core concepts are, e.g., Henshin [4, 54], ViaTra [61], Groove [78], and ATL (in-place) [63].

An instance model consists of a set of *objects* having *attributes* and *references*. While attributes are typed over data types, references are typed over classes. All instance models have to conform to a domain or type model, also called meta model, supporting *class inheritance*, including *abstract classes* (without instances) and *containment relations*. As example, consider the domain model for phones in the upper left-most screen shot in Figure 6.4. Due to simplification purposes, we do not discuss multiplicities and further constraints here.

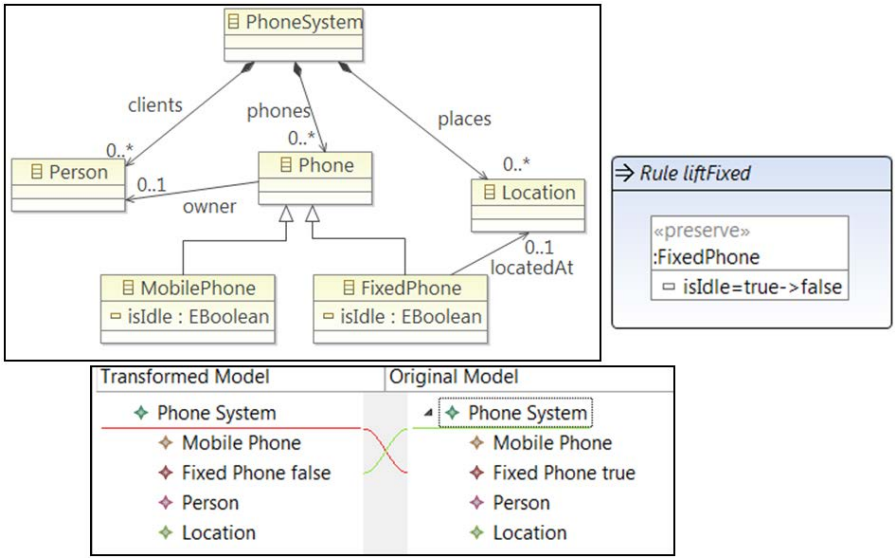


Figure 6.4: Domain model, rule, and a transformation step in Henshin

Transformation rules specify local changes on instance models. Usually, a *rule* r contains two model patterns, called left-hand side (LHS) specifying the precondition and right-hand side (RHS) formulating the postcondition of the rule. Either the differences between LHS and RHS show us the modifications induced by the rule (as in Henshin and ViaTra) or all modifications are defined in the RHS only (as in ATL). Alternatively, one pattern may be given being an integration of both rule sides where elements and references to be deleted or created are annotated accordingly. In addition, checks and computations of attribute values can be specified by expression languages such as JavaScript and OCL [121]. In Henshin, a rule is applicable to some model if the LHS pattern occurs in the model[2] or the guard pattern is satisfied, including the satisfaction of all attribute value checks. In that case, all specified rule actions are performed[3]. Rule elements may be typed over abstract classes, however, when applied, each rule element has to be mapped to some model element concretely typed. Rule elements specifying object creation have to be typed concretely

2 We restrict to injective matching of the LHS.

3 Formally, we follow the DPO graph transformation approach for rule application [35].

already in the rule. Furthermore, variables for attribute values may be defined in the scope of a rule to be used for checks and computations. When a rule is applied, its variables are bound to concrete data type values.

The application of a rule may be further restricted by conditions being any kind of propositional expression over the existence of model patterns. In the following, we restrict our considerations to the most simple ones being used by graph transformation-based approaches, i.e., *negative application conditions* (NACs) and *positive application conditions* (PACs) which forbid respectively require the existence of certain model patterns in instance models the rule is applied to.

Figure 6.4 shows an example using Henshin: A simple domain model for phone systems is shown together with rule *liftFixed* for lifting a fixed phone. The only effect of this rule is to unset attribute *isIdle*. This rule is applied to a simple instance model shown underneath using EMFCompare [45]. Note that the transformed instance model is shown on the left, while the original one is on the right.

6.3.2 *Quality aspects*

In this section, we motivate quality aims for model transformation systems. As for other software artifacts, the *correctness* of a model transformation system is defined w.r.t. the transformation language used and its interpretation in terms of the domain. While *language correctness* is considered syntactical, the interpretation forms the *model semantics*. Refactorings are supposed to preserve the model semantics.

Conciseness is concerned with the compactness of models which should present systems on the right abstraction level. It is open how to measure conciseness effectively. We can consider the size of transformation models, i.e., the size of domain models and the numbers of rules and rule elements. Sticking to a level of abstraction, we can say that the smaller these numbers are, the more concise is the model. A discussion on model transformation metrics can be found in [153].

A model transformation system is *changeable*, if it can be evolved rapidly and continuously. Conciseness and moreover, low redundancy and low coupling of modules, seem to be necessary prerequisites for the *changeability* of model transformation systems.

A model transformation system is *comprehensible* if it is understandable by the intended users. *Comprehensibility* is increased if a system is simple, concise, and structured enough to grasp its design. Moreover, comprehensibility is also influenced by the quality of the used concrete syntax (textual or graphical layouts), however, we do not consider this quality aspect throughout this section.

In summary, in the following catalogs we concentrate on *conciseness*, *changeability*, and *comprehensibility* of model transformation systems

when discussing potential relationships between quality aspects and smells respectively refactorings.

6.3.3 *Selected smells*

In this section, we present a small set of selected smells for rule-based in-place model transformation systems. Smells indicate suspicious system parts which should be inspected closer. Since we are mainly interested in the conciseness, comprehensibility, and changeability of model transformation systems, we investigate size and redundancy issues. Each smell is described in a structured way including affected quality aspects and refactorings that can eliminate them (the refactorings are described in detail in Section 6.3.4).

Large Rule

A rule specifies a model pattern and replaces it. It should handle a single aspect of the behavior. A large rule seems to care about too many different concerns.

DETECTION: This smell can be easily detected by counting the number of elements in a given rule. This smell depends very much on the modeling purpose: First, it has to be decided if objects, relations, preconditions, or actions are counted. Second, the threshold value has to be determined by experimental investigations.

AFFECTED QUALITY ASPECTS: Large rules do not represent a good modular design and can contain redundant information. Conciseness and comprehensibility might be affected.

USABLE REFACTORINGS: Loop Edges to Boolean Attributes, Extract Precondition

Redundant Attributes and References

Several model element types have equivalent attributes and references.

DETECTION: This smell can be detected by comparing the number of all attributes and references and the number of equivalent attributes and references.

AFFECTED QUALITY ASPECTS: Redundant information blows up the meta model and potentially also the rule set. It affects the conciseness, comprehensibility, and changeability of model transformation systems.

USABLE REFACTORINGS: Pull Up Attribute, Pull Up Reference

Redundant Rules

Several rules with equal pattern structures may differ in model element and attribute types used only.

DETECTION: This smell can be detected by comparing the number of all rule pairs differing in types only.

AFFECTED QUALITY ASPECTS: Redundant information blows up the meta model and the rule set. It affects the conciseness, comprehensibility, and changeability of model transformation systems.

USABLE REFACTORINGS: Pull Up Attribute, Pull Up Reference, Abstract Rule

Unused Object Type

There are object types that are not used in rules at all. Here, the purpose of the transformation rule set has to be considered when interpreting this smell (e.g., transformation of the entire model vs. local transformation).

DETECTION: This smell can be detected by counting the rules using a specific object type.

AFFECTED QUALITY ASPECTS: Unused object types may affect the correctness, the completeness and the conciseness of of transformation systems, dependent on the reason for this smell. Wrong types may be used, rules may be missing, or types may not be needed.

USABLE REFACTORINGS: Eliminate Object Type, Change Object Type

Delete and Create the Same Object

There are rules with objects being first deleted and then created again with the same attribute values but different contexts, or the same contexts but different attribute values.

DETECTION: This smell can be detected by applying clone detection to find corresponding patterns in rules.

AFFECTED QUALITY ASPECTS: If objects are deleted and immediately created again keeping their attribute values or their contexts, rules are not as concise and comprehensible as possible and can be improved.

USABLE REFACTORINGS: Move vs. Delete / Create

Rules With Common Subrule

The model transformation system has several rules containing the same subrule.

DETECTION: This smell has to apply some clone detection to find common subpatterns in rule parts.

AFFECTED QUALITY ASPECTS: If rules have common subrules, they contain redundant information that may affect the quality aspects conciseness, changeability, and comprehensibility.

USABLE REFACTORINGS: Unify Rules with Same Actions

Further smells are the well-known object-oriented smells that may be checked on the meta model having also effects on the rule set in general.

6.3.4 *Selected refactorings*

In this section, we present a collection of refactorings for rule-based in-place model transformation systems, each described in a systematic way. This collection mirrors our experiences in the application of model transformation to various purposes. It shows a range of refactorings serving several quality aims. For example, refactoring `Pull Up Attribute` reduces the amount of redundancy with respect to attribute definitions and potentially also reduces the number of rules. `Extract Precondition` reduces the number of rule elements and thus improves the conciseness. Each refactoring is systematically described including an example and change of identified smells before and after a refactoring, and an argumentation how semantics is preserved. For model transformation systems, semantics preservation may refer to the preservation of model transformation sequences, the preservation of transformed models, or the preservation of the amount of information in models.

Note that we do not present a refactoring which is probably most useful, i.e., the renaming of transformation systems, rules, types, etc., since its specification is obvious. Furthermore, the well-known refactorings of object-oriented models such as `Extract Superclass`, `Pull Up Attribute`, `Remove Middle Man`, etc. are applicable to domain models. Changes in domain models can imply changes in rules [36]. It may happen that rules differing in types only can be merged by using a superclass as type. Furthermore, most of the refactorings presented below come with an inverse, taking back the original refactoring effect. For instance, the inverse of `Pull Up Attribute` is `Push Down Attribute` which might be useful to prepare a variation of attribute definitions in subtypes. The inverse of refactoring `Extract Precondition`, called `Inline Precondition`, may be helpful for rule modifications. Inverse refactorings are not presented in detail.

Merge Rules Differing in Types Only

If there are rules which differ in object types only and these types are subclasses of the same superclass, they can be merged to one rule. This refactoring is often combined with a `Pull Up Attribute` refactoring of the domain model.

Input parameter: Names of the rules to be merged.

Example: Phones are refined to fixed and mobile phones. Both subtypes are attributed by a Boolean attribute *isIdle*. Two rules describe the lifting of fixed resp. mobile phones (see Figure 6.5 with domain model in Figure 6.4 on page 75). A refactoring `Pull Up Attribute` is performed on the domain model first to pull attribute *isIdle* up to class *Phone* (if class *Phone* does not have the *isIdle* attribute already). Figure 6.6 shows the desired domain model and contains a lift rule for phones in general being abstracted from the two original lift rules. This is possible, since the rules in Figure 6.5 differ in types only and thus, can be merged to the rule in Figure 6.6.

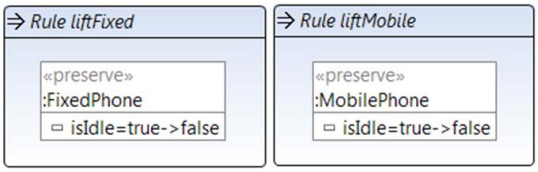


Figure 6.5: Before refactoring *Merge Rules Differing in Types Only*

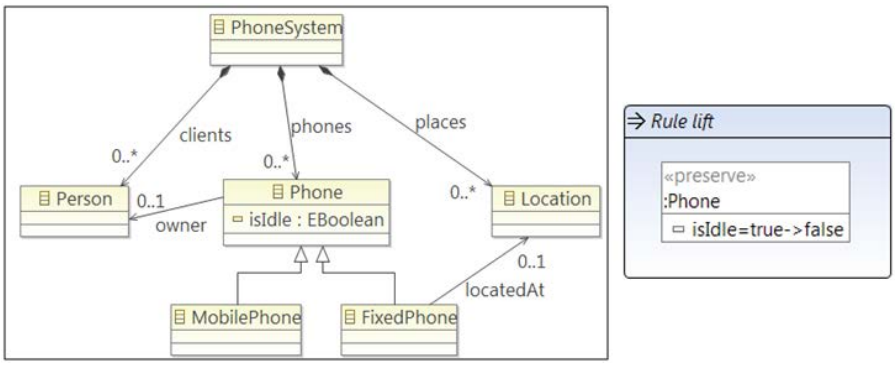


Figure 6.6: After refactoring *Merge Rules Differing in Types Only*

Precondition: Indicated rules differ in one object type only. The set of varying object types found contains all subclasses of a common superclass.

Strategy:

1. Identify all varying object types being classes with a common superclass.

2. Construct a new rule by taking one original rule and replacing identified subclasses by identified superclass. Rename the modified rule, if necessary.
3. Delete all remaining original rules.

Postcondition: All original rules are replaced by one new rule using the identified superclass as object type.

Affected smells: Redundant rules

Quality improvement: The number of rules becomes smaller. The model becomes more concise.

Semantics: The semantics is preserved, since the same transformation sequences are induced.

`Extract Precondition`

This refactoring makes preconditions explicit by extracting preserved parts as positive application conditions.

Input parameter: name of the rule

Example: A new fixed phone is installed. The rule mainly consists of context, i.e., preserved model elements that are not transformed. We extract the context that is not needed for inserting new edges into a positive application condition to make it more explicit (see Figure 6.7). Note that this reduces the size of the internal rule representation, though this effect is not visible in our compact notation.

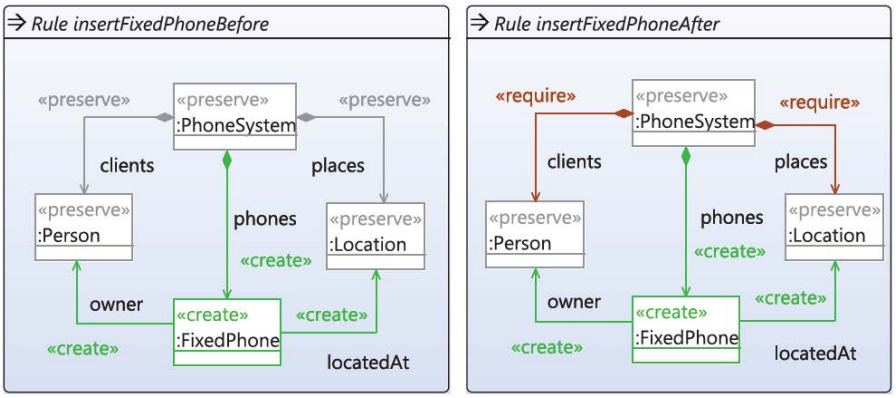


Figure 6.7: Before and after refactoring *Extract Precondition*

Precondition: none

Strategy:

1. Determine the preserved part of the input rule.
2. Create a new PAC and put those preserved objects into it that are not needed as targets for newly created references.

3. Reduce the rule's preserved part to the boundary objects needed for creating new references.

Postcondition: The preserved part of the rule is minimal.

Affected smells: Large Rule, Implicit Precondition

Quality improvement: The rule is more comprehensible, since the precondition is expressed more explicitly.

Semantics: The semantics is preserved, since the same transformation sequences are induced.

`Move Vs. Delete / Create`

Rule elements being deleted and created in the original rule, are moved afterwards.

Input parameter: Name of the rule

Example: Taking up the *Phone* example again, we consider a rule that replaces a fixed phone at one location by another one at another location, i.e., the fixed phone at the original location is deleted and a new one is created at the new location. After the refactoring, the rule specifies the movement of a fixed phone from one location to another one (see Figure 6.8).

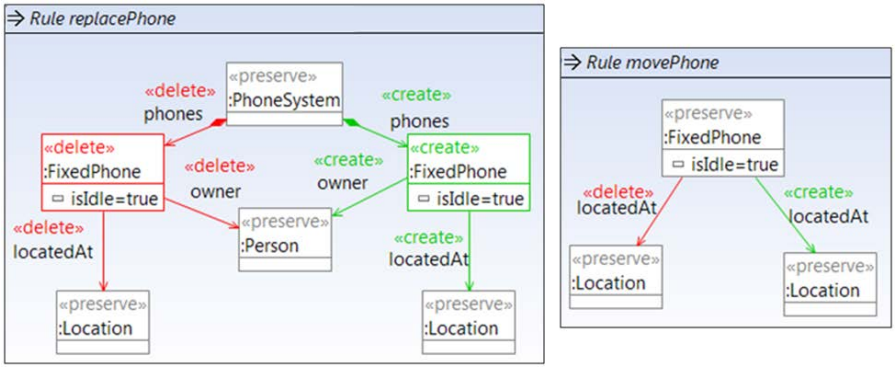


Figure 6.8: Refactoring of deletion and creation of a fixed phone

Precondition: There are model objects being first deleted and then created again with the same attribute values but different contexts or same contexts but different attribute values.

Strategy:

1. Identify objects and references being deleted and created afterwards. If these elements are attributed, they are either identified if the attribute values of created elements are the same as of deleted ones or if their adjacent references are created in the same way as they existed before.

2. Preserve identified elements instead of deleting and creating them.

Postcondition: The rule does not contain any object that is deleted and created with the same attribute values or the same context.

Affected smells: Delete and Create the Same Object

Quality improvement: The resulting rule is more concise, since unnecessary actions are avoided.

Semantics: The semantics is preserved in the sense that the same models are created, when both rules are applicable; however, the number of transformation effects when applying the refactored rule is reduced. Note that the original rule is not applicable if the *FixedPhone* node has more incident edges than specified by the rule (in the DPO approach), whereas the refactored rule is applicable also to fixed phones with more connections.

`Unify Rules with Same Actions`

Given a set of rules which share a subset of actions. This subset is encapsulated in a new rule to be applied first. The original rules are reduced to their remaining actions each.

Input parameter: Set of rule names

Example: For registering a new phone, it suffices for mobile phones to set the person who will own it. For fixed phones, their location has to be registered in addition. These two cases are specified in rules *registerMobilePhone* and *registerFixedPhone* in the upper part of Figure 6.9. However, the owner registration is common to both rules. Thus, we can form a kernel rule for phone registration handling the owner registration only. While this is all what has to be done for mobile phone there is a remainder rule for fixed phones. It specifies the location registration only. We have to make sure that to fixed phones both rules in the lower part of Figure 6.9 are applied.

Precondition: None

Strategy:

1. Identify the set of actions being shared by the set of input rules.
2. Besides the common actions identify also the common preserved model part.
3. Create a new rule, called *kernel rule* containing all identified actions and the identified preserved part. If common actions and preserved parts differ only in all subclasses of

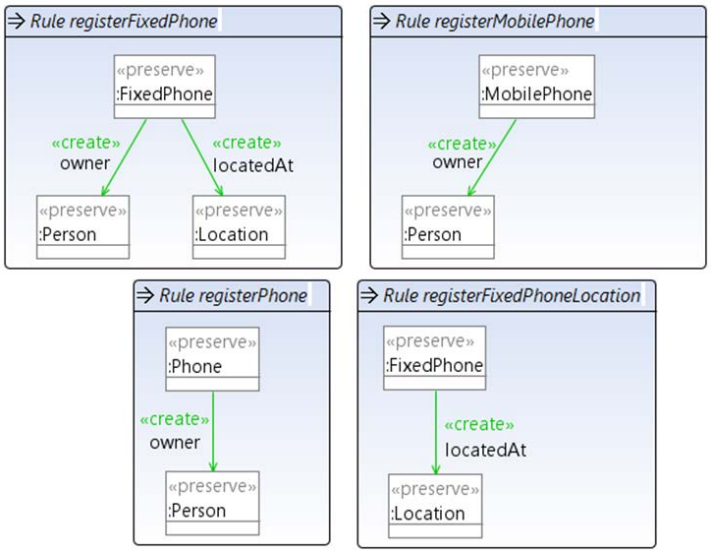


Figure 6.9: Before refactoring *Unify Rules with Same Actions* (top) and afterwards (bottom)

a common super class, this common super class is used as object type instead.

4. Reduce each of the original rules, called *remainder rule*, by the identified set of actions. Reduce the preserved part if it is common and not needed for the remaining actions, i.e., if it forms a precondition.
5. Make sure that the kernel rule is applied before remainder rules. This can be done e.g. by an additional control structure putting both rules into a sequence.

Postcondition: There is a new rule, the kernel rule, that contains all common actions and the common preserved part. All remainder rules do not contain common actions or preconditions anymore. A remainder rule is not applicable without applying the kernel rule beforehand.

Affected smells: Rules With Common Subrule

Quality improvement: The number of elements in the considered rule set is reduced, i.e., its conciseness is increased.

Semantics: Each original rule can be constructed by the composition of the kernel rule and optionally, a remainder rule. There may be more transformation sequences than before, since the resulting transformations allow for more interleaving of rule applications than before.

7

COMPOSITE MODEL REFACTORING

In Section 5.1, we discuss several complex metrics which rely on more basic metrics. Consequently, the question comes up whether the concept of composition can also be applied to other model quality assurance techniques considered in this thesis. In this chapter, we present an approach for composite model refactorings that concentrates on the specification of refactoring composition. The main idea of the approach is to specify composite model refactorings by a hierarchy of so-called refactoring units with parameter passing between different units by ports and port mappings.

The chapter is organized as follows: after motivating this work using selected refactoring examples for the UML, we reflect requirements and design decision for our approach. Then, we present the concepts of the approach and present an example specification in detail. Discussions on the automatic deduction of composite preconditions and related work conclude this chapter.

7.1 MOTIVATION AND EXAMPLES

This section motivates our work on composite model refactorings and gives some selected refactoring examples for UML models.

7.1.1 *Motivation and state-of-the-art*

In the literature, a variety of model refactorings, especially UML refactorings, are presented. See e.g. [147, 107, 130] for class model and statechart refactorings. While mainly focusing on smaller model changes, larger model refactorings are rarely considered though. Looking at code refactoring, however, it was soon clear that refactorings should be distinguished in atomic ones performing primitive changes and composite refactorings that are built up from existing ones [125, 134].

There is a number of approaches for specifying model refactorings, for example [129, 107, 114]. They differ heavily in the way refactorings are specified. It is common to all these approaches to use a preferred model transformation approach for specifying model refactorings and to concentrate on the specification of atomic ones. But atomic refactorings are rarely applied in isolation. Instead, they are

part of a group of refactorings that are all needed to perform a larger change. Despite the multitude of model refactoring approaches, the specification of composite model refactorings is not yet sufficiently supported by existing approaches in the sense that composite refactorings are consequently built up from existing ones being developed independently.

7.1.2 *Example refactorings*

The development of a specification language for composite model refactorings requires an analysis of model refactorings found in literature. In this section, we analyze composite statechart refactoring *Merge States* [130, 20] in detail that is used as running example throughout this chapter. Furthermore, we discuss a selection of further composite class model refactorings wrt. their reuse of more basic ones. The purpose of this analysis is to extract the important information of model refactoring composition that need to be specified, whereas we do not consider the specification of atomic checks and changes.

Running example: Merge States

State diagrams are used in software development to describe the behavior of systems [80]. They mainly consist of states, transitions between states, events, conditions, and actions. Refactoring *Merge States* is used to form a set of states into a single one [20][1]. Figure 7.1 (a) shows a simple statechart diagram dealing with the verification of the delivery address of a customer (i.e., it describes the life cycle of an object of class *DeliveryAddress*). It is started in state `not verified` and after performing the verification process (states `requested` and `retrieved`) it either returns to `not verified` or moves to `verified`, depending on the result of the verification. To further simplify the model, states `requested` and `retrieved` can be merged. This is possible since (1) the two states are arranged in a simple sequence, (2) there are only one entry and one exit action and nothing happens in state `retrieved`, and (3) the transition from `requested` to `retrieved` does not have a specified effect.

Refactoring *Merge States* is triggered from a contextual state (State `requested` in our example) and has one further parameter specifying the state that should be merged into the contextual state (State `retrieved` in our example). In contrast to [20], we treat this refactoring as composition of altogether three simpler ones. First, refactoring *Merge State Features* moves all actions from the parameter state to the contextual state, redirects all external transitions of the parameter state to the contextual state, and finally removes all inner transitions in between both states. Then, refactoring *Remove Isolated State* is ap-

1 For simplicity reasons we restrict this set to consist of only two states here.

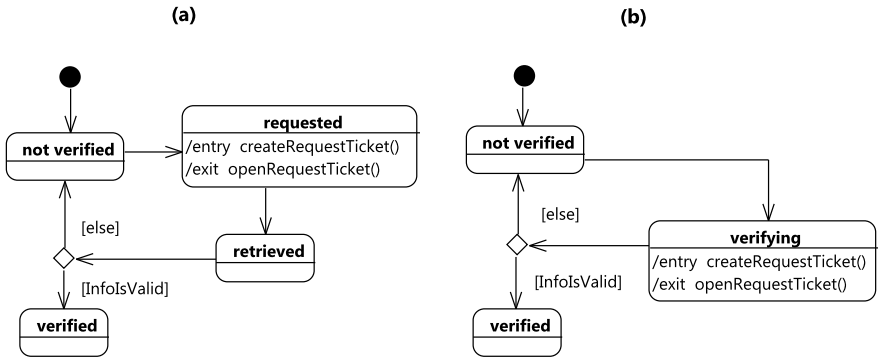


Figure 7.1: Example UML statechart (a) before and (b) after refactoring *Merge States*

plied (on state `retrieved`). Finally, refactoring *Remove Redundant Transition* is applied on each incoming transition of the contextual state `requested`. Whereas the application of the former two refactorings is mandatory, the application of the latter one is optional in order to execute refactoring *Merge States* successfully (in our example, it is not applied).The refactored statechart diagram is depicted in Figure 7.1 (b). Here, states `requested` and `retrieved` are merged. In addition, state `requested` is renamed to `verifying` by refactoring *Rename State* afterwards.

Extract Superclass

One of the most prominent refactorings for UML class models is *Extract Superclass* [64, 107]. It generates a new class as parent of a set of existing classes and pulls up their common features (attributes and operations) to the newly created class. Each of these actions can be considered as atomic refactorings. Refactoring *Extract Superclass* is triggered from a set of classes and has one further parameter, the name of the new class to be inserted. After checking several preconditions, refactorings *Create Superclass* can be applied on each contextual class, *Pull Up Attribute* on each attribute of the first contextual class, and *Pull Up Operation* on each operation of the first contextual class.

Extract Composite

This refactoring is motivated by the objective to improve the quality of a class diagram when introducing well-approved design patterns as presented in [85]. Figure 7.2 (a) shows a class diagram modeling the formula concept of the propositional calculus. Here, the information that exactly two formulae are combined to another one (either disjunction or conjunction) is modeled redundantly. Refactoring *Extract Composite* removes this redundancy as shown in Figure 7.2 (b).

The specification of this composite refactoring is similar to *Extract Superclass*. First, refactoring *Create Abstract Intermediate Class* is applied on classes `Conjunction` and `Disjunction`. It has one further parameter: the name of the new class (`CompositeFormula` in our example). Then, refactoring *Pull Up Composite Aggregation* is applied on attribute `formulae` of both selected classes. Finally, further common features (attributes and operations) are moved to the new class by (composite) refactoring *Pull Up Features*.

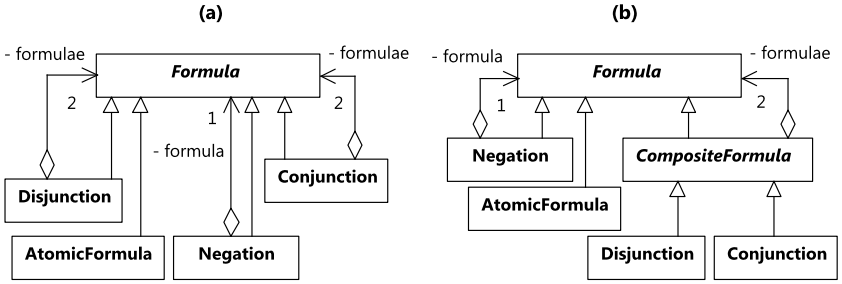


Figure 7.2: Example UML class model (a) before and (b) after refactoring *Extract Composite*

Further examples for composite model refactorings are *Extract Class*, *Inline Class*, *Extract Subclass*, *Inline Subclass*, *Remove Superclass*, and *Introduce Parameter Object* (compare Appendix E of this thesis).

7.2 Requirements and Design Decisions

This section discusses requirements and design principles for our approach on composite model refactoring based on the examples presented in the previous section.

7.2.1 *Requirements*

The main motivation to develop an approach for composite model refactoring is to raise the abstraction level as well as the grade of flexibility for specifying composite model refactorings. Doing so, the development of high-quality refactorings shall become easier and faster. This general motivation results in the following list of requirements:

- `Standard refactoring structure:` Each refactoring consists of an initial precondition check without taking parameters into account, a final precondition check using parameters, and a model change. A composite refactoring has to be a refactoring again.
- `Declarative composition:` The refactoring designer should be able to concentrate on the composition of refactorings, while neglecting technical details.

- `Simple and clear specification of composite refactorings:` A refactoring-specific and powerful set of specification language features is needed to support a simple and clear specification of composite model refactorings.
- `Black box composition:` The component refactorings are considered as black boxes, i.e., it does not matter how component refactorings are specified. The refactoring designer considers refactoring signatures only and can easily reuse existing refactorings (potentially coming from a pre-defined library).
- `Composite pattern:` Composite refactorings can be used as components of other composite refactorings. Their usage does not differ from that of atomic refactorings.

7.2.2 *Design decisions*

The requirements lead to the following basic design decisions our refactoring specification language is based on:

- Refactorings are model transformations with input parameters only. They are not supposed to return values. This principle is underlying most refactoring approaches in the literature.
- Model refactorings consist of three parts: an initial precondition check, a final precondition check, and the actual model change being a special kind of model transformation.
- Composite model refactoring units are composed from already existing refactorings using the composite design pattern [68]. The leaves of the applied composite design pattern are given by so-called *atomic units*. Each atomic unit represents a call of an existing *model refactoring*. Following this design, atomic as well as composite refactorings can be reused inside a composite refactoring.
- For parameter passing we use the concept of typed *ports* and *port mappings*.
- To adapt ports to special needs of component refactorings we use so-called *helper units*. Each helper unit represents a call of an existing *helper*, i.e., helpers can be reused in different contexts.

7.3 CONCEPTS, EXAMPLE SPECIFICATION, AND EVALUATION

In this section, we present the main concepts of our approach on composite model refactoring. The concepts are illustrated by a detailed example specification of the example refactoring *Merge States* as presented in Section 7.1.2. Finally, we give a short evaluation.

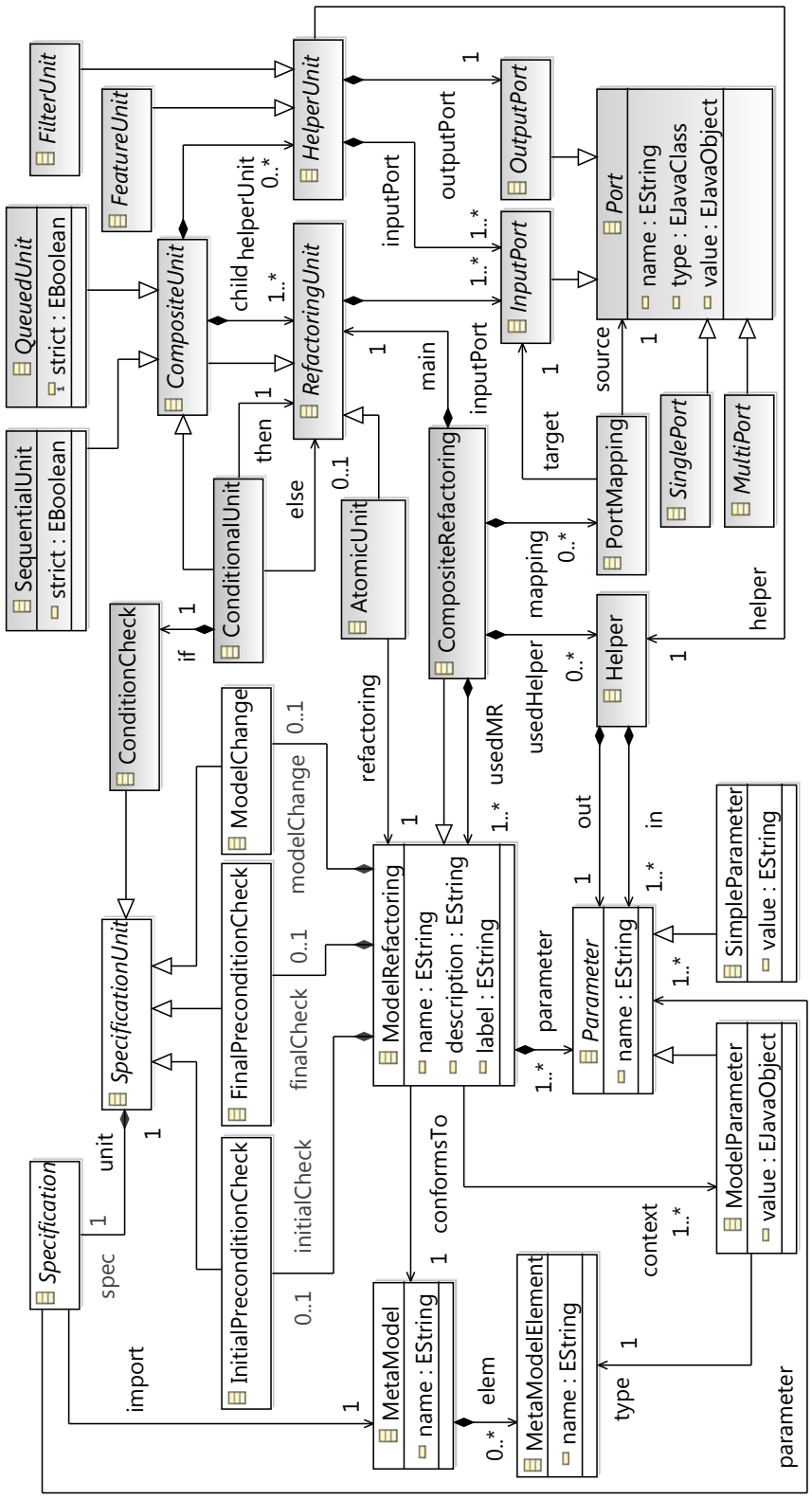


Figure 7.3: Meta model of the CoMReL language

7.3.1 *Main concepts*

We developed a domain-specific language for composite model refactoring, called **CoMReL** (**Co**mposite **M**odel **Re**factoring **L**anguage), based on the requirements and design decisions presented in the previous section. The approach consists of the following concepts (compare right-hand-side of the corresponding meta model in Figure 7.3):

Refactoring units. Composite model refactorings are specified by so-called `RefactoringUnits` defining their internal control structures of `CompositeRefactorings` building up on existing refactorings. Each composite refactoring is represented by exactly one `RefactoringUnit` which is normally composed by further units using the composite design pattern [68]. So-called `AtomicUnits` form the leaves of corresponding refactoring unit trees. They represent calls to already defined `ModelRefactorings`. A composite refactoring is considered again as `ModelRefactoring`, thus atomic units need not contain atomic refactorings only but also composite refactorings to be reused.

Ports and port mappings. To specify unit parameters, our approach uses `Ports`. A port is specified by a *name* and a *type*. Furthermore, it has a *value* representing an instance object that conforms to the *type* attribute. To pass port values from parent units to child units and from helper units to other units, so-called `PortMappings` are used. Each `PortMapping` connects a *source* port with a *target* port having identical types. Refactoring and helper units use different kinds of ports with respect to their direction: input and output ports. Furthermore, a port differentiation with respect to the multiplicity of values is useful. To address these two differentiations, we introduce altogether four abstract port kinds, `InputPort`, `OutputPort`, `SinglePort`, and `MultiPort` as well as four concrete port kinds, i.e., `SingleInputPort`, `SingleOutputPort`, `MultiInputPort`, and `MultiOutputPort`. Note that each kind of port mapping has an input port as target. This is due to the fact that output ports are used by helper units only.

Helper units. It can happen that ports of existing model refactoring units have types different from expected ones for composite refactorings. To solve this problem, we introduce the concept of `HelperUnits` intended to prepare the application of existing refactorings. Similarly to atomic units and existing model refactorings, a helper unit calls a corresponding `Helper` providing the intended functionality. Using this technique, it is possible to reuse existing helpers as often as necessary. Our approach provides two main kinds of helper units: `FeatureUnits` and `FilterUnits`. A feature unit extracts a specific feature of a model element, while filter units are used to extract elements from collections, for example.

Control constructs. For defining the execution flow of composite refactoring units, our approach uses three core constructs of iterative

programming: sequences, conditions, and loops. A `SequentialUnit` consists of at least one child unit. In general, sequential units are distinguished with respect to the semantics of the execution. If a sequential unit is defined to be *strict*, it is successfully executed if each child unit is executed successfully. In other words, if at least one child unit is not executed successfully, the entire sequence does not lead to any model changes. Vice versa, a non-strict sequential unit is always executed successfully, even if some of its child units are not executed successfully. A `ConditionalUnit` consists of a `ConditionCheck` being a specification unit and one or two refactoring units. If the condition check is executed successfully, the *then* unit is called, otherwise the *else* unit is called, if existing. A looping execution of refactoring units is defined by a `QueuedUnit` that consists of exactly one child unit.

7.3.2 *Example specification*

Figure 7.4 shows a visual representation of the specification model of refactoring *Merge States*. As described in Section 7.1.2, refactoring *Merge States* relies on three atomic model refactorings. The main refactoring unit *Merge States* is a strict `Sequential Unit` consisting of two `AtomicUnit` and one `SingleQueuedUnit`. The first `AtomicUnit` moves all actions from the parameter state to the contextual state, redirects all external transitions from the parameter state to the contextual state, and finally removes all inner transitions in between both states. The second `AtomicUnit` removes the (now empty) parameter state and finally the `SingleQueuedUnit` is applied on each incoming transition of the contextual state. This execution order of sub-units is defined by the usual direction of reading (left-to-right). Each refactoring unit is depicted with two compartments for maintaining included helper units respectively refactoring units.

Each `AtomicUnit` calls an already existing model refactoring, in our example *Merge State Features*, *Remove Isolated State*, and *Remove Redundant Transition*. The latter refactoring is applied to each incoming transition of the contextual state but need not be executed successfully on each transition. Hence, this atomic unit is put into a non-strict `SingleQueuedUnit` to address the looping execution for each redundant transition. We specify two parameters for refactoring *Merge States*, thus the main unit (`SequentialUnit` *Merge States*) must have two input ports. In Figure 7.4, input ports are visualized as rectangles. To distinguish single-valued and multi-valued ports, the latter ones are shown in red. The ports of each atomic unit are deduced from its referenced model refactoring. For example, the atomic unit that references *Merge State Features* gets two single input ports. Both ports have type *State* each whereas the first one represents the selected state and the second port represents the state whose features should be merged.

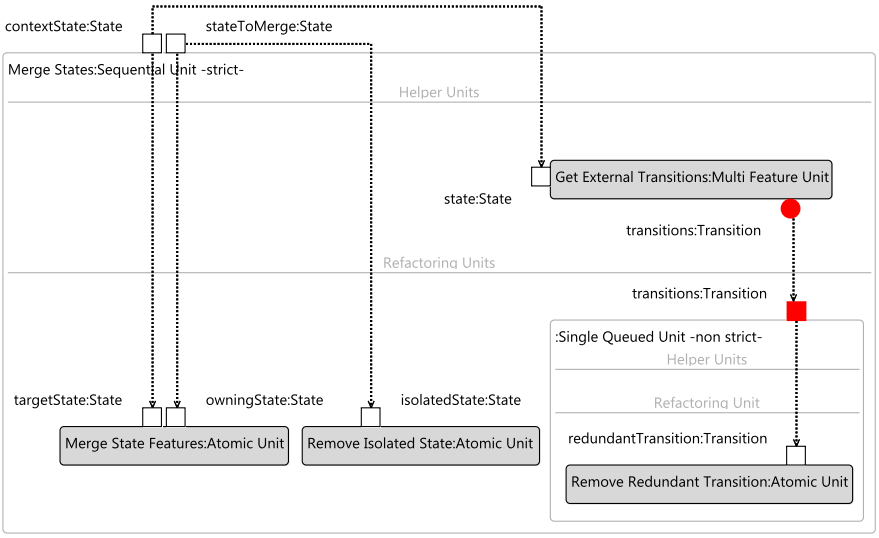


Figure 7.4: Unit specification of composite model refactoring *Merge States*

To model parameter passing, appropriate port mappings have to be created between corresponding ports, shown as dotted arrows in Figure 7.4. To ensure conformity with respect to typing and multiplicity of included ports, the main refactoring unit *Merge States* requires one helper unit: `MultiFeatureUnit` *Get External Transitions* takes the contextual state as input and yields all incoming and outgoing transitions except for potential reflexive transitions (transitions to itself). The corresponding output ports are visualized as circles.

7.3.3 *Evaluation*

As a proof of concept evaluation, our approach has been used to specify over 15 composite model refactorings such as extract and remove superclass, extract associated class, and introduce parameter object. While most of them combine atomic model refactorings only, we also considered the composition of composite model refactorings, e.g., for specifying refactoring *Extract Associated Class*. Our concepts have proved to be well suited for the specification of these composite model refactorings taking existing refactorings as black boxes. All needed helpers are simple and can be reused often. Concrete implementation details including appropriate CoMReL models can be found in Appendix F of this thesis.

7.4 TOWARDS AUTOMATIC DEDUCTION OF PRECONDITIONS

Composite refactorings are specified by a hierarchy of composite and atomic refactorings. Although each child refactoring behaves well in

the sense that its model changes can be performed once its preconditions are satisfied, the application of the composite refactoring may be performed partially only. This can happen since preconditions of child refactorings are not checked as early as possible but immediately when applying them. This situation can be improved by shifting preconditions of subunits to the precondition part of their surrounding composite refactoring. Such an automatic deduction of composite preconditions can be performed if the specification language supports conflict and dependency analyses. Preconditions may be dependent on model changes of preceding refactorings. If a precondition is purely dependent on preceding model changes, i.e., does not contain independent parts, it can even be erased since it is ensured by those corresponding model changes. Moreover, the automatic precondition deduction can be optimized by erasing equal or included preconditions.

In our approach for the automatic deduction of composite preconditions, we define a refactoring as a quadruple $R = (P_R, I_R, F_R, C_R)$ with P_R being a set of parameters, I_R and F_R being sets of initial and final preconditions and C_R being the actual model change performed by the refactoring. In case that all refactoring specifications I_R, F_R and C_R are defined using a model transformation language like Henshin [4], an atomic refactoring can be formalized by algebraic graph transformation [35]. Assuming the application of composite refactoring R as a simple sequence $R = R1; R2$, we have to check whether transformation rules in C_{R1} are in conflict with rules in I_{R2} respectively F_{R2}. To deduce composite preconditions, the main idea is to compute I_R and F_R by exploiting produce/use- and delete/forbid-dependencies of rules. The presence of such a dependency means that the corresponding precondition does not become a composite precondition, since it is automatically true. However, all further preconditions become preconditions of the composite refactoring R.

7.5 RELATED WORK

In this section, we compare our approach with related work on core concepts of composite refactorings.

In [134], Roberts considers dependencies between refactorings in a systematic way. Especially the computation of preconditions for composite refactorings from the preconditions of their components is useful and has been taken up by subsequent approaches such as the ones in [118, 89, 135]. O'Cinneide and Nixon [118] present composition concepts for Java refactorings, but do not mention language design and tooling for their concepts. Kniesel and Koch [89] present their language ConTraCT being based on conditional transformations. It has some similarities to our approach but seems to be simpler, since it relies on refactoring chains only. Concepts like conditional

and queued units as well as helper units are not mentioned. Their main contribution is a concept for automatic deduction of composite preconditions from component preconditions. Saadeh [135] introduces fine-grain transformations to specify UML model refactorings and to analyze their dependencies and conflicts. This work is purely conceptual, a domain-specific language for refactoring composition is not considered.

Our formalization of refactorings by graph transformation and the automatic deduction of composite preconditions is closely related to that in [89] and [135]. We use graph transformation, since it can be used as formal foundation of the model transformation language Henshin coming with a conflict and dependency analysis for model transformation rules. However, since specification languages can be chosen flexibly in our approach, other specification approaches and analyses may be easily integrated.

8

CONCLUSION AND FUTURE WORK

Since models are the primary artifacts in model-based software development, model quality assurance is of increasing importance for the development of high quality software. In this part, we inspected several model quality assurance techniques and integrated the techniques model metrics, model smells, and model refactorings into a syntax-oriented model quality assurance process that can be easily adapted to specific needs in model-based projects.

The quality assurance process consists of two sub-processes: First, dependent on the modeling language and the modeling purpose, specific quality goals, and hence project- and domain-specific quality checks and refactorings have to be defined. Quality checks are formulated using model smells which can be specified in terms of model metrics and anti-patterns. Then, the specified quality assurance process is applied to concrete software models. Static model analysis uses the pre-defined model metrics and smells. Based on the outcome of the model analysis, appropriate model refactoring steps can be performed. However, it has to be considered that new model smells can be introduced by refactorings. This check-improve cycle should be performed as long as needed to get a reasonable model quality.

In future work, further model quality assurance techniques such as a structured use of design patterns may be considered. In this context, the use of modeling conventions that have to be proven to be effective with respect to prevention of defects might be integrated into the quality assurance process. Here, adequate modeling conventions have to be developed being usable to prevent for specific model smells. Moreover, there are model smells which are difficult to describe by metrics or patterns. For example, *shotgun surgery* is a code smell which occurs when an application-oriented change requires changes in many different classes. This smell can be formulated also for models, but it is difficult to detect it by analyzing models. It is up to future work to develop an adequate technique for this kind of model smells.

In our approach, we concentrate on quality aspects to be checked on the model syntax. They include not only the consistency with the language syntax definition, but also the conceptual integrity in using patterns and principles in similar situations, and the conformity with modeling conventions often defined and adapted to specific software

projects. As a conceptual basis for a Goal-Question-Metrics approach to our quality assurance process we take six classes of quality goals for software models identified in a systematic literature review [116]. Here, it is up to future work to identify potential dependencies between these so-called 6C goals in order to support an appropriate selection on helpful quality assurance techniques.

Since UML is a widely accepted standard in software modeling and subject of a number of research activities, we provide an overview on metrics, smells, and refactorings for UML models discussed in literature including structured descriptions and relations to the 6C goals. Here, we concentrate on class models since class diagrams are the mostly used UML diagram type [29]. However, due to the actually pragmatic search strategy this overview is rather incomplete (however quite comprehensive). Conducting systematic literature reviews may be an adequate mean to overcome these limitations in the future.

Finally, this part also presents an approach for composing refactorings to more complex ones. In this context, it is up to future work to analyze the preconditions of component refactorings with respect to their execution order and to deduce a composite precondition therefrom. Here, we think of using concepts from algebraic graph transformations like critical pair analysis [35]. In a similar way, specifications of model smells and model refactorings could be analyzed in order to decide whether the refactoring (1) is usable to erase the smell, or (2) its application would insert a new one.

To conclude, the author is convinced that performing quality assurance processes is an essential task to obtain software of high quality. It has been shown in three example applications that using the structured model quality assurance process presented in this part, model-based and model-driven development can be made more mature yielding software of higher quality.

Part II

A FLEXIBLE TOOL ENVIRONMENT FOR QUALITY ASSURANCE IN THE ECLIPSE MODELING PROJECT

9

INTRODUCTION TO PART II

The increasing use of model-based and model-driven software development processes induces the need for high-quality software models. As presented in Part I of this thesis, the model quality assurance process might be structured into two sub-processes: a process for the specification of project-specific model quality assurance techniques, and a process for applying them on concrete software models during a MBSD process. Both parts are based on model analysis techniques, more specifically on reports on model metrics and on checks against the existence (respectively absence) of model smells. Finally, refactoring is the technique of choice for fixing a recognized model smell.

Since manually reviewing models is time consuming and error prone, several tasks of the proposed project-specific model quality assurance process should consequently be automated. In particular, the following major functionalities should be provided:

- User-friendly support for project-specific configurations of model metrics, smells, and refactorings.
- Calculation of model metrics, detection of model smells, and application of model refactorings.
- Generation of model metrics reports.
- Suggestion of suitable refactorings in case of specific smell occurrences.
- Provision of suitable information in cases where new model smells come in by applying a certain refactoring.
- Support for the implementation of new model metrics, smells, and refactorings.

This part presents a flexible tool environment that has been development to fulfill these requirements and that represents a major contribution of this thesis. The tool environment is part of the Eclipse Modeling Project (EMP) [49] and can be found and downloaded as official Eclipse incubation project named **`EMF Refactor`** under the following URL: `http://www.eclipse.org/emf-refactor/`. It is open source and available under the Eclipse public license (EPL).

Besides supporting the afore mentioned functionalities EMF Refactor addresses two main concepts. First, it is *highly integrated* in two senses. On the one hand, all model quality assurance tasks can be performed directly within the Eclipse IDE. This means, that the users do not have to export the model (using its XMI format) and use third-party tools, for example for analyzing it. On the other hand, the tool environment integrates the model quality assurance techniques in various ways. For example, smells are based on specific metrics and refactorings are proposed as quick fixes for occurring smells. Second, EMF Refactor is *flexible* with respect to the specification mechanisms of new model quality assurance techniques. New techniques can be either defined using existing ones (e.g., metrics can be composed to more complex metrics) or they are specified by one of the supported languages Java, OCL, and Henshin. Here, further languages can be integrated by new adapters.

The chapters of Part II contain the following:

Chapter 10 introduces basic technologies related to the development of EMF Refactor like the Eclipse Modeling Framework (EMF) and the Language Toolkit (LTK). Furthermore, it presents an overview on the state-of-the-art of model quality assurance tooling for UML and EMF as well as a comparison study on refactoring tools in Eclipse.

Chapter 11 discusses several topics concerning the development process of EMF Refactor. It gives an overview on the specific requirements and presents details on its design and the architecture.

Chapter 12 presents the application of model quality assurance techniques supported by the tools of the EMF Refactor framework. It covers the main functionalities metrics calculation and reporting, smell detection, and refactoring along an example UML class model.

Chapter 13 illustrates concrete specification mechanisms for model quality assurance techniques with respect to a domain-specific modeling language. It shows how to define new metrics, new model smells, and new refactorings as well as how to manually specify relations between model smells and model refactorings.

Chapter 14 evaluates EMF Refactor. By performing and analyzing several studies, the suitability of the tools for supporting the techniques of the model quality assurance process presented in Part I as well as scalability respectively performance issues are considered.

Finally, ***Chapter 15*** concludes and discusses directions for future work on EMF Refactor.

10

BASIC TECHNOLOGIES AND STATE-OF-THE-ART

This chapter addresses some basic technological topics related to the development of an integrated tool environment for model quality assurance. The chapter starts with a brief introduction to the Eclipse Modeling Framework (EMF). We decided to use EMF as underlying technology (1) since it represents a widely-used open source technology in model-based software development, (2) since it comes with a very active community providing a variety of helpful tools, and (3) due to our comprehensive knowledge in this domain. Then, we give an overview on the state-of-the-art of model quality assurance tooling for both, EMF and UML, the mostly used MOF-based modeling language [140]. Finally, we present a comparison study on refactoring tools in Eclipse as a preliminary step for extracting adequate requirements for developing the EMF model refactoring component of the tool set which is called EMF Refactor in the following.

10.1 THE ECLIPSE MODELING FRAMEWORK (EMF)

The Eclipse Modeling Framework Project (EMF) [44, 144] extends Eclipse by modeling facilities including the generation, editing and view of models. It allows defining models and modeling languages by means of so-called structured data models.

The core of EMF contains **`Ecore`**, a meta model similar to the superstructure of UML class diagrams, and a runtime support for models including change notifications, XMI serialization, and an API for manipulating EMF objects. Ecore represents an implementation of the Essential MOF (EMOF) part of the Meta-Object Facility (MOF) standard [120] defined by the Object Management Group (OMG) [122]. It serves as general meta model and is typed over itself, i.e., Ecore is an EMF model itself.

Figure 10.1 shows a subset of Ecore with its most important meta classes and relations. The classes essentially correspond to common entities in UML class diagrams, i.e., `EPackage`, `EClass`, and `EAttribute` correspond to packages, classes, and attributes. Classes may be declared as abstract classes or interfaces by corresponding attributes of `EClass`. Meta class `EReference` corresponds to associations. However, these kind of references are always directed. Moreover, refer-

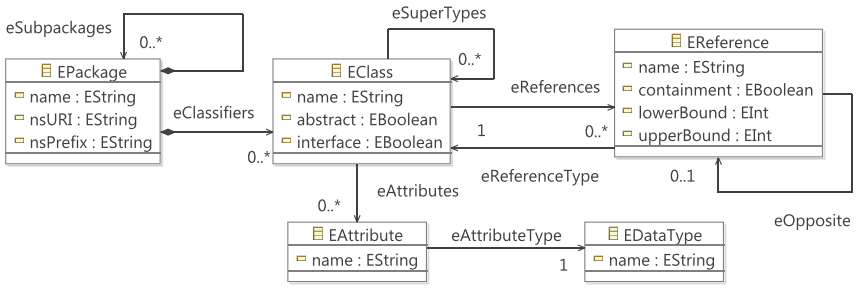


Figure 10.1: Subset of the Ecore meta model

ences may be explicitly equipped with lower and upper bound and be declared as derived references which shall rather be calculated in a certain way. Additional attributes further support the use of these Ecore elements in different ways, e.g., the attribute *nsURI* of `EPackage` is very important as it assigns a globally unique namespace to a package to allow for its unambiguous identification.

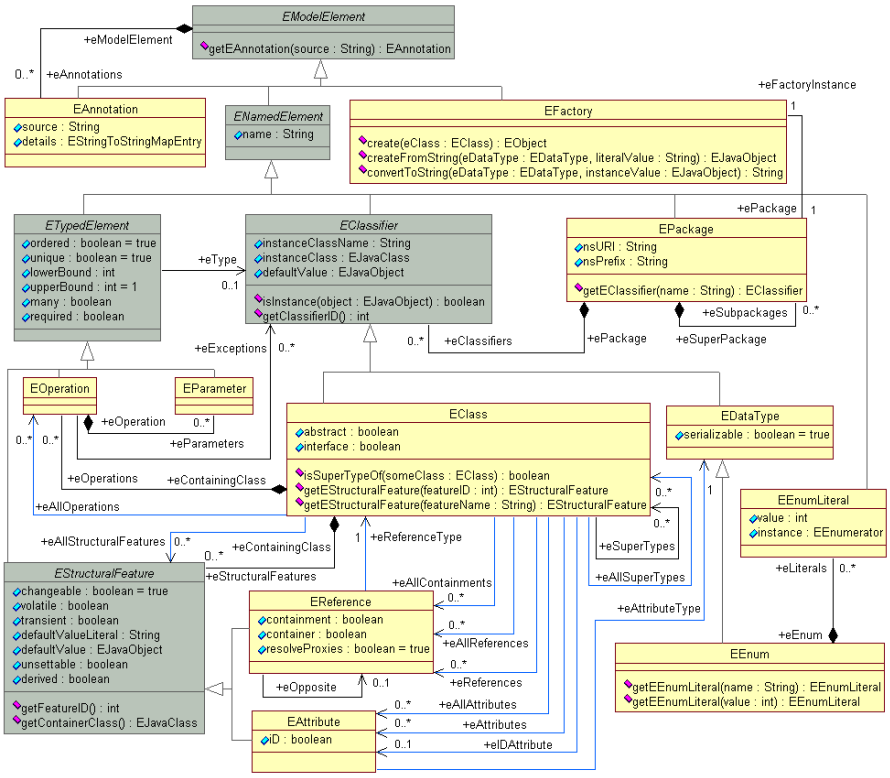


Figure 10.2: The Ecore meta model

Figure 10.2 gives a more detailed overview of the Ecore components and their relations, attributes, and operations (taken from the javadoc API part of the EMF web site [44]). Abstract meta classes are indicated by italic letters and are colored by a slightly darker background.

The basic editors in the Eclipse Modeling Framework are the tree-based editors providing basic CRUD operations [1]. EMF also ships a generic tree-based editor, called *Sample Reflective Editor*, that allows to edit arbitrary EMF models, i.e., Ecore models and instance models. Furthermore, EMF comes with a GMF based diagram editor [53] for Ecore models as well. EMF instances are by default edited using a generated tree-based editor or the Sample Reflective Editor. However, dedicated diagram as well as textual editors can be generated with some effort using frameworks like GMF, EuGENia [52], Sirius [60], and Xtext [62].

10.2 TOOL SUPPORT FOR MODEL QUALITY ASSURANCE

The existing tool support for model quality assurance is mainly aiming at UML and EMF modeling.

10.2.1 *UML modeling*

Considering UML modeling, quality assurance tools are integrated in standard UML CASE tools to a certain extent. In the following, we give a rough overview on existing UML model quality assurance tools: In UML CASE tools such as the IBM Rational Software Architect (RSA) [82] and MagicDraw (MD) [103], a number of metrics and validation rules are predefined and can be configured in metrics and validation suites. MD supports class model metrics (e.g., measuring the number of classes, inheritance tree depth, and coupling), so-called system metrics such as Halstead [79] and McCabe [111], and requirements metrics based on function points and use cases. Validation rules comprise completeness and correctness constraints such as all essential information fields are filled, properties have types specified, etc. Further validation rules can be specified using Java or a restricted form of OCL. RSA also supports predefined metrics. In addition, models can be checked against validation rules being based on metrics. A tool dedicated to the calculation of UML metrics is SDMetrics [1]. SDMetrics analyzes the structural properties of UML models and uses object-oriented measures as well as design rule checking to automatically detect design problems in UML models such as circular dependencies and violation of naming conventions. Measurement data is displayed in different views (e.g., tables, histograms, and kiviat diagrams) and can be exported in various formats like HTML and XML. Furthermore, SDMetrics supports custom definitions of UML metrics and design rules using XML-based configuration files.

As far as we know, no popular commercial UML CASE tool (such as Sparx Enterprise Architect [75], IBM Rational Software Architect (RSA) [82], and MagicDraw [103]) supports model refactoring facili-

1 Create, Read, Update, Delete

ties exept for renaming model elements. However, some research prototypes for model refactoring are discussed in the literature, e.g., in [129, 20, 107]. Most of them are no longer maintained. For example, Porres [129] describes the execution of UML model refactorings as sequence of transformation rules and guarded actions. He presents an execution algorithm for these transformation rules and constructed an experimental, meta model driven refactoring tool that uses SMW, a scripting language based on Python, for specifying the UML model refactorings.

To summarize, UML CASE tools and further model analysis tools for UML provide model analysis by predefined metrics and validation rules and support the custom configuration of metrics and validation suites as well as the definition of further custom techniques but do not offer an integrated, custom configured quality assurance environment for UML models based on metrics, smells (validations), and refactorings.

10.2.2 *EMF modeling*

Since EMF has evolved to a well-known and widely used modeling technology, it is worthwhile to provide model quality assurance tools for this technology. To the best of our knowledge, explicit tool support for metrics calculation on EMF-based models is not yet available. However, there is the EMF Model Query Framework [46] to construct and execute query statements that can be used to compute metrics and to check constraints. These queries have the form of select statements similar to SQL and can also be formulated based on OCL. Specified queries are triggered from the context menu. The configuration of queries in suites as well as reports on query results in various forms are not provided. The EMF Validation Framework [48] supports the construction and assurance of well-formedness constraints for EMF models. Two modes are distinguished: batch and live. While batch validations are explicitly triggered by the client, live validations listen to change notifications to model objects to immediately check that the change does not violate any well-formedness constraint.

The Epsilon language family [51] provides the Epsilon Validation Language (EVL) to validate EMF-based models with respect to constraints that are, in their simplest form, quite similar to OCL constraints. Moreover, EVL supports dependencies between constraints, customizable error messages to be displayed to the user and the specification of fixes to be invoked by the user to repair inconsistencies. For reporting purposes, EVL supports a specific validation view reporting the identified inconsistencies in a textual way. Suitable quick fixes are formulated in the Epsilon Object Language (EOL – the core language of Epsilon) and therefore not specifically dedicated to model refactoring. Here, Epsilon provides the Epsilon Wizard Lan-

guage (EWL) [91], a textual domain-specific language for in-place transformations of EMF. We compare our first refactoring prototype with EWL in detail in the next section.

Another approach for EMF model refactoring is presented in [132, 32]. Here, the authors propose the definition of EMF-based refactoring in a generic way, however they do not consider the comprehensive specification of preconditions. Our experiences in refactoring specification show that it is mainly the preconditions that cannot be defined generically.[2] Furthermore, there are no attempts to analyze EMF models wrt. model smell detection.

Finally, the MoDisco framework [7] provides a model-driven reverse engineering process for legacy systems in order to document, maintain, improve, or migrate them. Here, several specific models are deduced (for example, Java models are deduced from Java code) which can be analyzed in order to detect anti-patterns and then be manually improved, for example by refactorings. Similar to the UML and EMF tooling discussed so far, MoDisco supports the specification and computation of custom metrics and queries on models as well as metrics visualization. The main difference between MoDisco and the tool environment presented in this thesis is the intended purpose (reverse engineering vs. modeling).

In summary, there are various tools to support EMF model analysis and to improve EMF models by refactoring. However, there is not yet a comprehensive tool environment for specifying and applying predefined and custom metrics, smells, and refactorings to EMF models in an integrated way where metrics, smells, and refactorings are tightly inter-related. This thesis is heading towards such a tool environment in the following.

10.3 AN EXPLORATION STUDY ON EMF REFACTORING TOOLS

A variety of tools for quality assurance of code exist, in particular for the refactoring of Java code as provided, e.g., by the Eclipse Java Development Tools (JDT) [55]. As demonstrated in the previous section, tool support for model refactoring is limited, particularly for models based on the Eclipse Modeling Framework (EMF). In this section, we present the results of a comparison study that examines three approaches for EMF model refactoring, namely the Language Toolkit (LTK), the Epsilon Wizard Language (EWL) and a preliminary prototype of EMF Refactor. The aim of this study is to extract adequate requirements for developing the EMF model refactoring component of EMF Refactor.

The section is organized as follows. First, we describe the design of the comparison study and the criteria for analyzing the approaches.

2 For example, see [4] for a more complex refactoring with elaborated precondition checks.

Then, we present basic implementation details for each refactoring solution. Finally, the differences as well as the benefits and drawbacks of the solutions are discussed.

10.3.1 *Study description*

Since UML class models are closely related to class structures in object-oriented programming languages such as C++ and Java, many existing code refactorings can be directly adopted to UML. However, few model refactorings are specific to the model level only. The sample refactoring used in this comparison study is of the latter category.

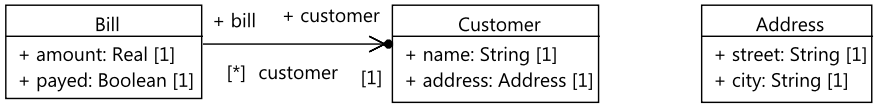


Figure 10.3: Example class diagram before refactoring (excerpt)

Figure 10.3 shows an excerpt of an example class diagram. At a first glance, class *Address* seems to be isolated from all further model elements. However, taking a closer look to the model, we identify attribute *address* in class *Customer* being of type *Address*. For a better understanding of class structures, it would be worthwhile to represent this relationship more explicitly. This can be achieved by applying model refactoring *Change Attribute to Association End*. After refactoring application, attribute *address* of class *Customer* will be depicted as an association end in the same manner as attribute *customer* of class *Bill*. Please note that the result of the refactoring might lead to misunderstandings, e.g., if there are too many associations such that the diagram is harder to comprehend [3]. Figure 10.4 shows the corresponding part of the UML superstructure specification [124].

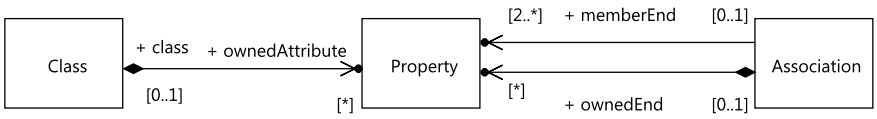


Figure 10.4: UML specification for attributes and association ends (excerpt)

We implement this model refactoring by means of the Language Toolkit (LTK) [67] and the Epsilon Wizard Language (EWL) [91], two existing solutions to handle refactorings in Eclipse. Furthermore, the comparison study investigates a preliminary prototype of EMF Refactor [47] that has been presented in the thesis of Lars Schneider [138].

3 This means, that applying refactoring *Change Attribute to Association End* is not suitable to improve the model's quality in general.

We analyze the implemented solution of each approach with respect to seven defined criteria. For each criterion questions are defined which are observed during the specification and execution of the refactoring solutions.

Concerning the **specification** step of the example UML refactoring we considered the following four criteria:

- **Complexity** – What is the amount of work to implement the example refactoring? Are there any ways to reduce this effort? Here, LoC and the number of specified rules are discussed and compared according to the personal perception of the author.
- **Correctness** – Is it possible to specify a refactoring resulting in an inconsistent model when applied? Are there any precautions to avoid this?
- **Testability** – Which effort is needed to test the specified refactoring in detail? Are there ways to automate these tests?
- **Modularity** – Is it possible to combine already implemented refactorings? This might be an important aspect when defining more complex refactorings by reusing existing ones.

With respect to the **application** of the example UML refactoring we analyzed the following three criteria:

- **Interaction** – How easy is it to apply the refactoring? Are there any facilities to simplify user inputs? Here, differences considering UI features have to be discussed from a (indeed subjective) user's point of view.
- **Features** – Does the refactoring provide a preview of its effect? Does it provide undo and redo functionality?
- **Malfunction** – What happens if the appropriate refactoring cannot be executed in the given situation? Are there reasonable error messages?

10.3.2 *Study implementations*

Before presenting the interesting parts of the three implementations using LTK, EWL, and the EMF Refactor prototype [4], these technologies are introduced first.

Considered tools

The **Language Toolkit (LTK)** [67] is a language neutral API to specify and execute refactorings in an Eclipse-based IDE. Therefore, it

4 The prototype is called *ProRef* in the following.

can also be used to handle EMF model refactorings. The API can be found in the plug-ins `org.eclipse.ltk.core.refactoring` and `org.eclipse.ltk.ui.refactoring`. Their classes provide an exact, predefined procedure for refactorings in Eclipse. Example refactorings that use LTK are those for Java provided by the JDT [55].

The **Epsilon Wizard Language (EWL)** [91] is an integral part of Epsilon [51], a platform for building consistent and interoperable task-specific languages for model management tasks. For this purpose, Epsilon consolidates common facilities in a base language, the Epsilon Object Language (EOL) [90], to be extended by new task-specific languages. EWL is a tool-supported language for specifying and executing automated model refactorings [5]. These model refactorings are applied on model elements that have been explicitly selected by the user. Here, Epsilon provides an Eclipse-based interpreter for executing EWL programs.

The third tool in this comparison study, **ProRef**, has been developed by Lars Schneider in the context of his diploma thesis [138]. It serves as a preliminary prototype of the refactoring component of EMF Refactor [47]. In ProRef, the development of new refactorings is based on EMF Tiger [15, 13], an Eclipse plug-in that performs in-place EMF model transformations [14, 114]. The model transformation concepts of EMF Tiger are based on algebraic graph transformation concepts. It provides a graphical editor for the design of transformation rules and a Java code generator which has been extended by ProRef.

The LTK solution

The specification of the example UML refactoring required the implementation of seven Java classes. During the implementation, it became obvious that only four of them are refactoring specific whereas the remaining three classes can be seen as common for all kind of EMF model refactorings. The refactoring specific classes are:

- `RefactoringInfo` - This class manages all required informations like the selected attribute (object of meta type `Property`), the name of the new association and the name of the association's *ownedEnd* property.
- `RefactoringInputWizardPage` - This class is responsible for UI tasks like displaying and handling the required input (name of the new association and name of the association's *ownedEnd* property).
- `RefactoringAction` - This class is responsible for refactoring initiation. It sets the selected attribute and initializes instances of several LTK classes. Moreover, this class serves the extension point `org.eclipse.ui.popupMenus`.

5 Kolovos et al. [91] call them *update transformations in the small.*

- `RefactoringProcessor` - This is the main class for executing the sample refactoring. Method `checkInitialConditions()` checks whether the type of the selected attribute is an instance of `Class` and whether it is not already part of an `Association` (see Listing 10.1). Method `createChange()` creates an instance of class `EMFChange` by generating a `ChangeDescription`[6] that describes all required model changes and is also used for undo and redo functionality. Listing 10.2 shows an excerpt of this method. Here, feature *name* of the newly created `Association` is set to the appropriate String managed by the `RefactoringInfo` object.

```
1  RefactoringStatus result = new RefactoringStatus();
2  Property property = this.refactoringInfo.getProperty();
3  if (property.getType() != null
4    && property.getType() instanceof Class) {
5      if (property.getAssociation() != null) {
6         result.addFatalError("The selected Property is " +
7             "already an association end!");
8      }
9  } else {
10   result.addFatalError("The type of the selected " +
11         "Property is not a Class!");
12 }
13 return result;
```

Listing 10.1: Method body *RefactoringProcessor::checkInitialConditions()*

```
1  Association as = UMLFactory.eINSTANCE.createAssociation();
2  Map.Entry<EObject, EList<FeatureChange>> entryAsName =
3    createEObjectToChangesMapEntry(as);
4  FeatureChange fCAsName = createFeatureChange();
5  fCAsName.setFeatureName("name");
6  fCAsName.setDataValue(this.refInfo.getAssociationName());
7  entryAsName.getValue().add(fCAsName);
8  changeDescription.getObjectChanges().add(entryAsName);
```

Listing 10.2: Excerpt of method body *RefactoringProcessor::createChange()*

The EWL solution

In EWL, the sample refactoring has been implemented as follows: First, the type of the selected model element has to be checked to be a `Property` of type `Class`. Furthermore, this property does not already have to be part of an `Association`. These preconditions are checked in the `guard` section of the EWL program (see Listing 10.3). Here, variable *self* refers to the model object which is used to invoke the refactoring (the attribute in our example). If the guard conditions fails, the refactoring is canceled automatically.

6 `org.eclipse.emf.ecore.change.ChangeDescription`

```
guard {
  if (self.isKindOf(Property)) {
    if (self.type.isDefined()) {
      if (self.type.isKindOf(Class)) {
        return self.association.isUndefined();
      } else { return false; }
    } else { return false; }
  } else { return false; }
}
```

Listing 10.3: Guard section of the EWL solution

The most important part of the EWL solution is the `do` section that specifies the effects of the refactoring (see Listing 10.4). After obtaining the user input (the name of the new association and the name of the association's *ownedEnd* property; not shown in Listing 10.4) all necessary new objects are created (instances of `Association` and `Property` as owned end) and the proper features are set. Finally, the new association is added to the owning package.

```
var upperVal : new LiteralInteger;
upperVal.value = 1;
var lowerVal : new LiteralInteger;
lowerVal.value = 1;
var ownedEndP : new Property;
ownedEndP.name = srcProperty;
ownedEndP.type = self.class;
ownedEndP.upperValue = upperVal;
ownedEndP.lowerValue = lowerVal;
var assoc = new Association;
assoc.name = associationName;
assoc.ownedEnd.add(ownedEndP);
assoc.memberEnd.add(self);
self.class.package.packagedElement.add(assoc);
```

Listing 10.4: Guard section of the EWL solution

The ProRef solution

In ProRef (respectively EMF Tiger) model refactorings are designed by ordered sets of rules. Each rule describes an if-then statement on model changes. If the pattern specified in the left-hand side (LHS) exists, it is transformed into another pattern defined in the right-hand side (RHS). Additionally, so-called negative application conditions (NACs) can be specified which represent patterns that prevent the rule from being applied. Mappings between objects in LHS and RHS and/or between objects in LHS and NACs are used to express preservation, deletion, and creation of objects.

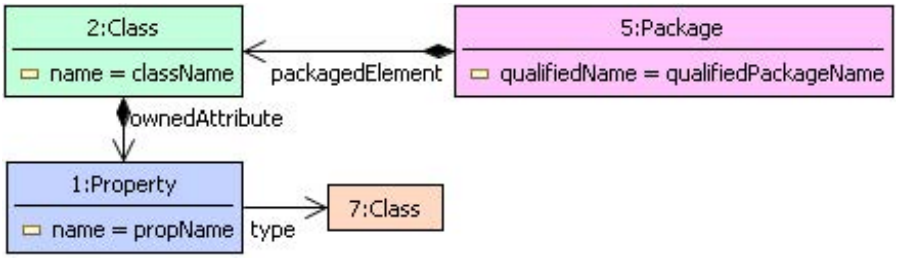


Figure 10.5: Left-hand-side (LHS) of the ProRef / EMF Tiger solution

The LHS of the rule that specifies the sample refactoring is shown in Figure 10.5. This pattern represents the abstract syntax which has to be found when starting the refactoring from within the context menu of a `Property` named *propName* whose type is a `Class`. To ensure that the selected `Property` is not already part of an `Association` an appropriate NAC is defined, that is similar to the LHS but with an additional `Association` instance that references the selected `Property` as *memberEnd* (not shown here).

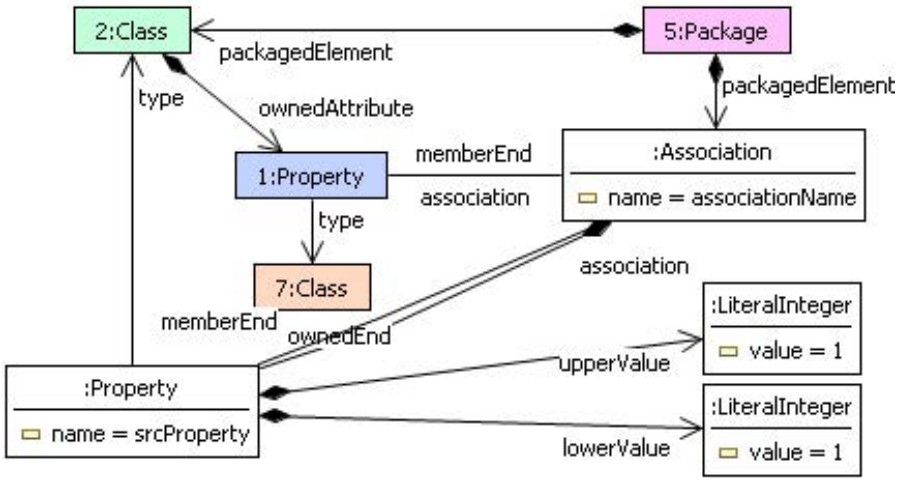


Figure 10.6: Right-hand-side (RHS) of the ProRef / EMF Tiger solution

Figure 10.6 shows the RHS of the sample refactoring rule. It contains a new `Association` object with a new opposite association end (`Property`). This end is equipped with multiplicity 1 as lower and upper bound. The newly created objects are named by additional input variables *associationName* and *srcProperty*.

During rule specification it is possible to check whether the specified transformation rule is consistent. This means that the EMF model transformation always leads to models that are consistent with typing and containment constraints. To do so, you have to check whether the rules perform restricted changes of containments only. Consistent EMF model transformations behave like algebraic graph transformations. Hence, the rich theory of algebraic graph transformation can be applied to show functional behavior and correctness [16]. The

sample refactoring rule is consistent because all new object nodes (`Association`, `Property`, and two `LiteralIntegers`) are connected immediately to their respective container (see Figure 10.6).

10.3.3 *Study observations*

In this section we discuss the results of the comparison study. The solutions presented in the previous section are compared along the criteria introduced in Section 10.3.1.

Complexity – All three techniques require a good understanding of relevant parts of the UML meta model [124]. In *LTK*, seven Java classes consisting of 711 LoC were implemented. 416 LoC can be generated and 195 are refactoring specific, in particular methods *createChange()* and *checkInitialConditions()* of class `RefactoringProcessor`. Here, the most challenging task is to properly implement the appropriate object of the LTK class `ChangeDescription` due to its complex API. In *EWL*, one single file with 47 LoC was implemented. Automatically generating generic parts would not lead to a significant reduction. Finally, in *ProRef* the entire refactoring code was generated from one rule specification only that contains 32 objects (`EClasses` and `EReferences`). Individual parameter settings for code generation are supported by a convenient wizard.

Correctness – In *LTK*, an incorrectly specified `ChangeDescription` object would lead to an inconsistent model after executing the refactoring. There are no known precautions available to avoid this. Since all model changes in *EWL* are directly implemented, there is also no special support to specify refactorings which yield consistent models only. *ProRef* however uses EMF Tiger that provides consistency checks regarding containment and multiplicity issues. This is done using the underpinning graph transformation concepts. Hence, it is almost impossible to specify transformations, especially refactorings leading to inconsistent models.

Testability – A specified refactoring has to be tested by applying it to various models that represent possible situations. Since every refactoring in *LTK* is a single Eclipse plug-in, it is very time-consuming to start a new Eclipse instance after each code change. These tasks could be facilitated by generating test code or using PDEUnit [58], a test framework for Eclipse plug-ins. Because *EWL* is an interpreted language, testing is not that time-consuming and a straightforward task. Nevertheless, there is no known way to automate this. For *ProRef* the same comments as for *LTK* hold.

Modularity – Since all model changes in *LTK* are directly implemented in Java, it seems to be possible to combine several existing refactorings to more complex ones by passing required parameters, and adapting conditions, and `Change Descriptions`. Here, it is necessary to develop an advanced approach to support this features. In

EWL, there is no known way to combine refactorings so far, except for copying and adapting code of existing ones. For *ProRef* the same comments as for *LTK* hold.

Interaction – All approaches provide the selection of refactorings via the context menu of a `Property` element in the standard EMF instance editor. *EWL* additionally supports graphical GMF-based editors [53] which can be done by the other approaches as well if they serve a further extension point. The refactoring wizard page of *LTK* provides one input line for each required parameter. Each parameter has a specified default value. In *EWL*, the context menu has an entry specific to the name of the selected `Property`. All parameters are entered in separate dialogs including specified default values. For *ProRef* the same comments as for *LTK* hold.

Features – In *LTK*, after parameter editing the wizard provides an optional preview of the model changes made by the refactoring. The preview is provided by EMF Compare [45]. Undo/Redo functionality is supported. In *EWL*, there is no preview available, but Undo/Redo functionality is supported. After parameter editing in *ProRef* the wizard always shows a preview of possible model changes when executing the refactoring. Again, this is provided by EMF Compare. Undo/Redo functionality is not supported.

Malfunction – If a certain precondition in *LTK* fails, a message box including a reasonable error message is shown as specified in method *checkInitialConditions()* of class `RefactoringProcessor`. *EWL* provides the refactoring only, if all preconditions specified in the `guard` section hold. After parameter input in *ProRef*, the user is informed when the refactoring can not be executed because of violated conditions. This is merely done by the generic message *The refactoring changed nothing at all*. Each solution requires non-empty parameters, more precisely names for the new model elements `Association` and `Property`.

Criteria	**LTK**	**EWL**	**ProRef**
Complexity	o	+	+
Correctness	-	-	+
Testability	o	o	o
Modularity	o	-	o
Interaction	+	+	+
Features	+	o	o
Malfunction	+	+	o

Table 10.1: Results of the comparison

Table 10.1 summarizes the results of the comparison study. Each approach has been evaluated and marked as follows:

- The approach meets the evaluation criterion: **+**
- The approach does not meet the evaluation criterion but is still moderate: **o**
- The approach does not meet the evaluation criterion at all: **-**

Each approach has its individual strengths and weaknesses. *LTK* provides permanent positive results when executing the model refactoring. This is not surprising because *LTK* was developed to unify refactoring processes in Eclipse. However, *ProRef* seems to be more suitable for specifying EMF model refactorings. This is because of its graphical nature of defining model transformations and its underlying graph transformation concepts. Last but not least, *EWL* shows advantages in both, refactoring specification and refactoring application. However, in both categories there is another approach that seems to be more suitable than *EWL*.

In summary, *LTK* is the leading approach during model refactoring application, whereas *ProRef* seems to be the most promising one in specifying EMF model refactorings. As a conclusion of the presented comparison study, it looks worthwhile to check whether *LTK* and *ProRef* can be combined in a way that merges the benefits of both approaches. Such a combination of *LTK* with *ProRef* seems to be a promising way to go.

The results of this comparison study help with extracting adequate requirements for developing the model refactoring component of EMF Refactor [47]. The requirements, design and architecture of this integrated tool environment for model quality assurance in Eclipse are presented in the following chapter of this thesis.

11

REQUIREMENTS, DESIGN AND ARCHITECTURE

This chapter presents basic topics related to early phases during the development of an integrated tool environment for model quality assurance in Eclipse, called EMF Refactor [47]. The chapter starts with a high-level overview on the requirements on the tool environment, subdivided into common requirements and those which are specific to the application of existing techniques and the specification of new techniques, respectively. Afterwards we present details on the design and the architecture of EMF Refactor. Finally, we summarize how the requirements are met by the design.

11.1 REQUIREMENTS

The definition of the proposed model quality assurance process presented in Part I of this thesis lead to a set of requirements on our supporting tool set concerning model metrics, model smells, and model refactorings. In this section, we summarize these requirements as high-level abstractions. We elaborated the requirements in an iterative process and completed them according to the results of the explanation on existing model quality assurance tools presented in the previous chapter.

11.1.1 *Common requirements*

In this section, we summarize those requirements which are common to all model quality assurance tools of EMF Refactor.

GENERALITY Each tool should be based on the Eclipse Modeling Framework (EMF) [44, 144], i.e., the corresponding functionality should be provided on any EMF-based model since. We decided to use EMF as underlying technology (1) since it represents a widely-used open source technology in model-based software development, (2) since it comes with a very active community providing a variety of helpful tools, and (3) due to our comprehensive knowledge in this domain.

REUSE The tool environment should reuse existing Eclipse and EMF components as far as possible. Moreover, already implemented

quality assurance techniques should be reusable since many of them recur most likely in several projects even if modeling purposes may differ.

11.1.2 *Application requirements*

The following requirements are specific for the application of the model quality assurance tools within EMF Refactor (metrics calculation, smell detection, and refactoring execution).

CONFIGURABILITY The modeler (respectively the model reviewer) should be provided with a project-specific configuration of predefined model metrics, smells, and refactorings suites. For smells which are based on specific metrics it should be possible to configure project-specific thresholds.

INTEGRATED APPLICATION The corresponding functionality should be triggered from within several editors in Eclipse like the standard tree-based EMF instance editor, but also graphical and textual model editors should be supported.

REPORTING Calculated metric values and detected model smell occurrences should be reported in specific integrated views. Model elements being involved in a specific smell occurrence should be highlighted in the standard tree-based EMF instance editor. Furthermore, it should be possible to export the results of both, a metric calculation and a smell search, in various formats such as HTML, PDF, and XML.

REFACTORING FEATURES The application of refactorings should follow the homogeneous refactoring execution structure in Eclipse including a preview of the resulting model. This includes a transactional execution of refactorings. Furthermore, the refactoring tool should provide undo and redo functionality as well as an optional analysis of smell occurrences before and after refactoring application. Moreover, smells should be related to refactorings being suitable to erase the smell, and refactorings should be related to smells potentially occurring after applying the refactoring.

QUICK-FIX MECHANISM It should be possible to invoke a suitable refactoring from within the context menu of a concrete smell occurrence in the smell results view.

11.1.3 *Specification requirements*

This section summarizes the requirements on the components for the specification of new model metrics, smells, and refactoring.

FLEXIBLE SPECIFICATION APPROACHES It should be possible to define custom metrics, smells, and refactorings for arbitrary EMF-based models. Here, the tools should support various concrete specification approaches. As default specification language, Java should be supported since Eclipse, especially EMF, is based on the Java technology.

COMPOSITION A designer should be provided with tool support for composing existing techniques. In particular, the tools should support compositional metrics, metric-based model smells, and composite model refactorings.

CODE GENERATION The tools should provide a comfortable input mechanism for specification-related information like the meta model, the name, and a description of an arbitrary metric, smell, or refactoring. Afterwards, each tool should generate Java code that can be used by the application component in order to provide the corresponding functionality (metrics calculation, smell detection, and refactoring execution).

11.2 DESIGN AND ARCHITECTURE

This section discusses the architecture of our tool environment for EMF model quality assurance and summarizes the components used by it. Each tool is based on the Eclipse Modeling Framework [144, 44], i.e., each tool can be used for arbitrary models whose meta models are instances of EMF Ecore, for example domain-specific languages, common languages like UML2[1] used by Eclipse Papyrus [59] and the Java EMF model used by JaMoPP [31] and MoDisco [7, 57], or even Ecore instance models themselves.

EMF Refactor mainly consists of six components out of two dimensions: With respect to the main functionalities (calculating model metrics, detecting smells, and executing refactorings) there is an ***application module*** for each. Similarly there are three ***specification modules*** for generating metrics, smell, and refactoring plugins containing Java code that can be used by the corresponding application module. For simplicity reasons, we refer to these plugins as ***custom QA plugins*** in the remainder of this section. We start with a description of the specification dimension.

11.2.1 *The specification modules*

Figure 11.1 shows the architecture of a specification module using a UML component model. The specification module provides the

1 In this thesis, we refer to UML2 being the standard EMF-based representation of UML2, i.e., `org.eclipse.emf.uml2.uml`.

generation of custom QA plugins containing the metric-, smell-, or refactoring-specific Java code. Using the Eclipse plugin technology, libraries consisting of model quality assurance techniques can be provided. So, already implemented techniques can be reused.

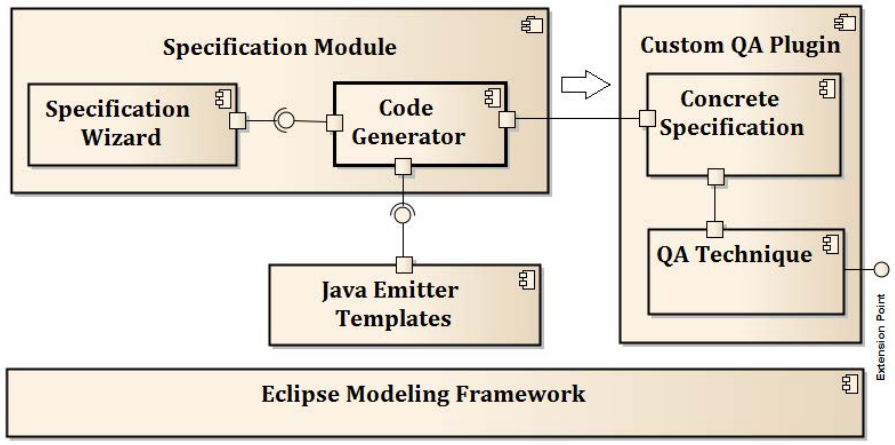


Figure 11.1: Composite structure of a specification module

Actually, the following specification technologies are supported:

- Java [126]; version 6.
- OCL [121] provided by the Eclipse Modeling Project [49].
- Henshin [4, 54], a model transformation engine for the Eclipse Modeling Framework based on graph transformation concepts. Henshin uses pattern-based rules that can be structured into nested transformation units with well-defined operational semantics.
- CoMReL, a model-based language for the combination of EMF model refactorings (see Chapter 7).

More concretely, the following techniques can be used in a concrete specification of a new EMF model metric, smell, or refactoring:

1. Model metrics can be concretely specified in Java, as OCL expressions, by Henshin pattern rules, or as a combination of existing metrics using a binary operator.
2. Model smells can be concretely specified in Java, as OCL invariants, by Henshin pattern rules, or as a combination of an existing metric and a comparator like *greater than* (>).
3. The three parts of a model refactoring can be concretely specified in Java, as OCL invariants (only precondition checks), in Henshin (pattern rules for precondition checks; transformations for the proper model change), or as a combination of existing refactorings using the CoMReL language.

The specification module provides wizard-based specification processes (component *Specification Wizard* in Figure 11.1). After inserting specific information (like the name of the metric, smell, or refactoring, and the corresponding meta model) the *Code Generator* component uses the *Java Emitter Templates* framework [56] to generate the specific Java code required by the corresponding extension point (see arrow in Figure 11.1). Table 11.1 shows the extension point descriptions for EMF model metrics, smells, and refactorings.

`org.eclipse.emf.refactor.metrics`	
Field name	**Description**
name	Name of the EMF model metric.
id	Unique identifier of the EMF model metric.
description	Description of the EMF model metric (optional).
metamodel	Namespace URI of the corresponding meta model.
context	Name of the context element type .
calculateclass	Java class that implements `IMetricCalculateClass`.

`org.eclipse.emf.refactor.smells`	
Field name	**Description**
name	Name of the EMF model smell.
id	Unique identifier of the EMF model smell.
description	Description of the EMF model smell (optional).
metamodel	Namespace URI of the corresponding meta model.
finderclass	Java class that implements `IModelSmellFinderClass`.

`org.eclipse.emf.refactor.refactorings`	
Field name	**Description**
name	Name of the EMF model refactoring.
id	Unique identifier of the EMF model refactoring.
description	Description of the EMF model refactoring (optional).
metamodel	Namespace URI of the corresponding meta model.
controller	Java class that implements `IController`.
gui	Java class that implements `IGuiHandler`.

Table 11.1: Extension point descriptions for metrics, smells, and refactorings

Besides basic information like the name, id, or the corresponding meta model of a concrete model quality assurance technique the following interfaces have to be implemented:

IMETRICCALCULATECLASS This interface provides the calculation of the corresponding EMF model metric on a given model element. Here, two methods have to be implemented: method `void setContext(List<EObject> context)` for maintaining the model element on which the metric should be calculated on, and method `double calculate()` for the proper calculation of the metric value on this element.

IMODELSMELLFINDERCLASS This interface provides the detection of the corresponding model smell in a given EMF model. It has one method which must be implemented by the corresponding Java class: `findSmell(EObject root)`. Here, the model is specified by parameter `root`. The method returns a list of detected smell occurrences where such an occurrence is given by a list of model elements which are involved in the detected smell.

ICONTROLLER This interface is responsible for executing the corresponding model refactoring. Here, the main method which has to be implemented is `getLtkRefactoringProcessor()` that returns an instance of class `RefactoringProcessor` from the Language Toolkit (LTK) API [67]. Within this class, the refactoring specific preconditions are checked by the two boolean methods `checkInitialConditions ()` and `checkFinalConditions()` whereas the refactoring is executed by method `createChange()`.

IGUIHANDLER This interface checks whether the refactoring can be executed on the given context elements (method `showInMenu (List<EObject> selection)`); the process is started by method `RefactoringWizard show()`. As above, `RefactoringWizard` is a class of the LTK API.

11.2.2 *The application modules*

Figure 11.2 shows the architecture of an application module. It uses the Java code of the custom QA plugins generated by the corresponding specification module (compare right-hand side of Figure 11.1 and left-hand side of Figure 11.2) and consists of two components. The *Configuration Component* maintains project-specific configurations of metrics, smells, and refactorings. The *Runtime Component* is responsible for metrics calculation, smell detection, and refactoring execution. Depending on the concrete specification approach, the runtime component uses the appropriate components Java, OCL, Henshin, or the internal CoMReL interpreter. Further languages, especially model transformation languages like EWL [91], may be integrated by suitable adapters [68]. For exporting calculated model metrics, the reporting engine BIRT [43] is used. Finally, the Language Toolkit (LTK) [67] is used for homogeneous refactoring execution and EMF Compare [45],

a tool that provides comparison and merge facility for any kind of EMF models, for refactoring preview.

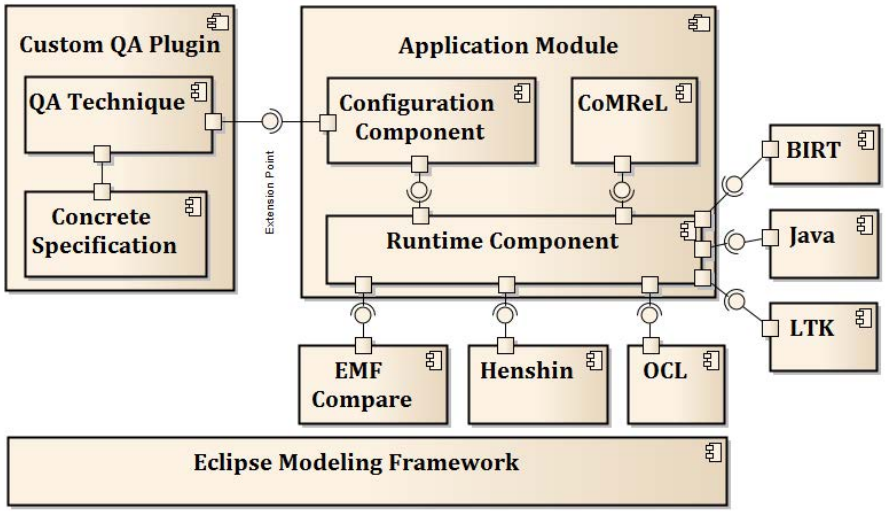


Figure 11.2: Composite structure of an application module

For manually defining the relationships between model smells and model refactorings, our tool environment uses the Eclipse extension point technology again to provide information about these relationships globally. Therefore, two extension points for the manual definition of relations between model smells and model refactorings are provided. Since our tools identify smells and refactorings by distinct identifiers (see Table 11.1), these extension points require relations from *smell IDs* to a list of *refactoring IDs* (in case of providing suitable refactorings for a given smell) and relations from *refactoring IDs* to a list of *smell IDs* (in case of possible new smells when applying a given refactoring). To serve these extension points in a user-friendly way, we extend the property page of a certain Eclipse plugin project in the workspace by providing graphical user interfaces for (de-)activating appropriate relations.

11.3 SUMMARY

In the preceding sections, we presented the requirements on the tool environment for model quality assurance as well as the design and the architecture of EMF Refactor. Table 11.2 summarizes how the requirements are met by the design.

The following chapters present how to work with both kinds of modules. For simplicity reasons and to relate the application of our tools to the process presented in Part I of this thesis, we first present how to use the application module and its implemented quality assurance techniques. Thereafter, Chapter 13 presents how to specify new metrics, smells, and refactorings.

Common requirements	
Requirement	**Implementation**
`Generality`	The entire tool set is based on EMF.
`Reuse`	EMF Refactor uses JET, BIRT, and EMF Compare. Already implemented techniques can be installed using the plugin- and extension point-technology of Eclipse.
Application requirements	
Requirement	**Implementation**
`Configurability`	Provided by the *Configuration Component* (see Figure 11.2).
`Integrated application`	The functionality is integrated into the standard tree-based EMF instance editors, graphical GMF-based editors as used by Papyrus UML, and textual editors provided by Xtext. Moreover, we integrated our tool environment into the widely used EMF-based UML CASE tool IBM Rational Software Architect.
`Reporting`	EMF Refactor provides specific metrics and smell analysis views, a highlighting mechanism of model elements, and a result reporting based on BIRT.
`Refactoring features`	EMF Refactor provides a homogeneous refactoring workflow by using LTK, a preview on refactoring changes, smell analysis facilities during refactoring, and extension points for specifying smell-refactoring relations.
`Quick-fix mechanism`	Provided by the smell-refactoring relations and dynamic analysis.
Specification requirements	
Requirement	**Implementation**
`Flexible Specification Approaches`	EMF Refactor supports Java, OCL, and the model transformation language Henshin as possible specification approaches.
`Composition`	Metrics can be composed to complex ones, smells can be based on a metric, and refactorings can be combined by using CoMReL.
`Code generation`	Provided by the *Specification Module* (see Figure 11.1).

Table 11.2: Requirements and corresponding implementation

12

EXAMPLE APPLICATIONS

This chapter presents the application of several model quality assurance techniques supported by the tools within the EMF Refactor framework [47]. We demonstrate this application on a UML class model representing the domain of a vehicle rental company which is presented in Section 12.1. The subsequent sections show

- how to select and calculate model metrics and how to report the corresponding results (Section 12.2),
- how to customize and search for model smells and how to report the corresponding findings (Section 12.3),
- and finally how to perform model refactorings in order to improve the structure of the model (Section 12.4).

12.1 EXAMPLE UML CLASS MODEL

Figure 12.1 shows a first UML example model that has been developed in an early stage during the development of an accounting and customer management system for a vehicle rental company. This first version of the domain model of the company is shown as UML class diagram modeled using the EMF-based UML CASE tool Papyrus [59]. The model consists of altogether four packages:

- Package `Commonalities` contains general concepts (enumerations, interfaces, and common classes like `Person` and `Date`).
- Package `RentalCompany` contains the main entities of the company (represented by class `VehicleRental`). The company has a number of employees and customers. Each customer is associated with a concrete employee (see association end *consultant*). Special persons are subcontractors of the company which represent both, a customer and an employee. The right-hand-side of this package shows that the company owns several cars, trucks, and motorbikes which can be rented by some customer.
- Packages `Services` and `Invoicing` contain classes for renting a vehicle by some customer as well as for billing purposes.

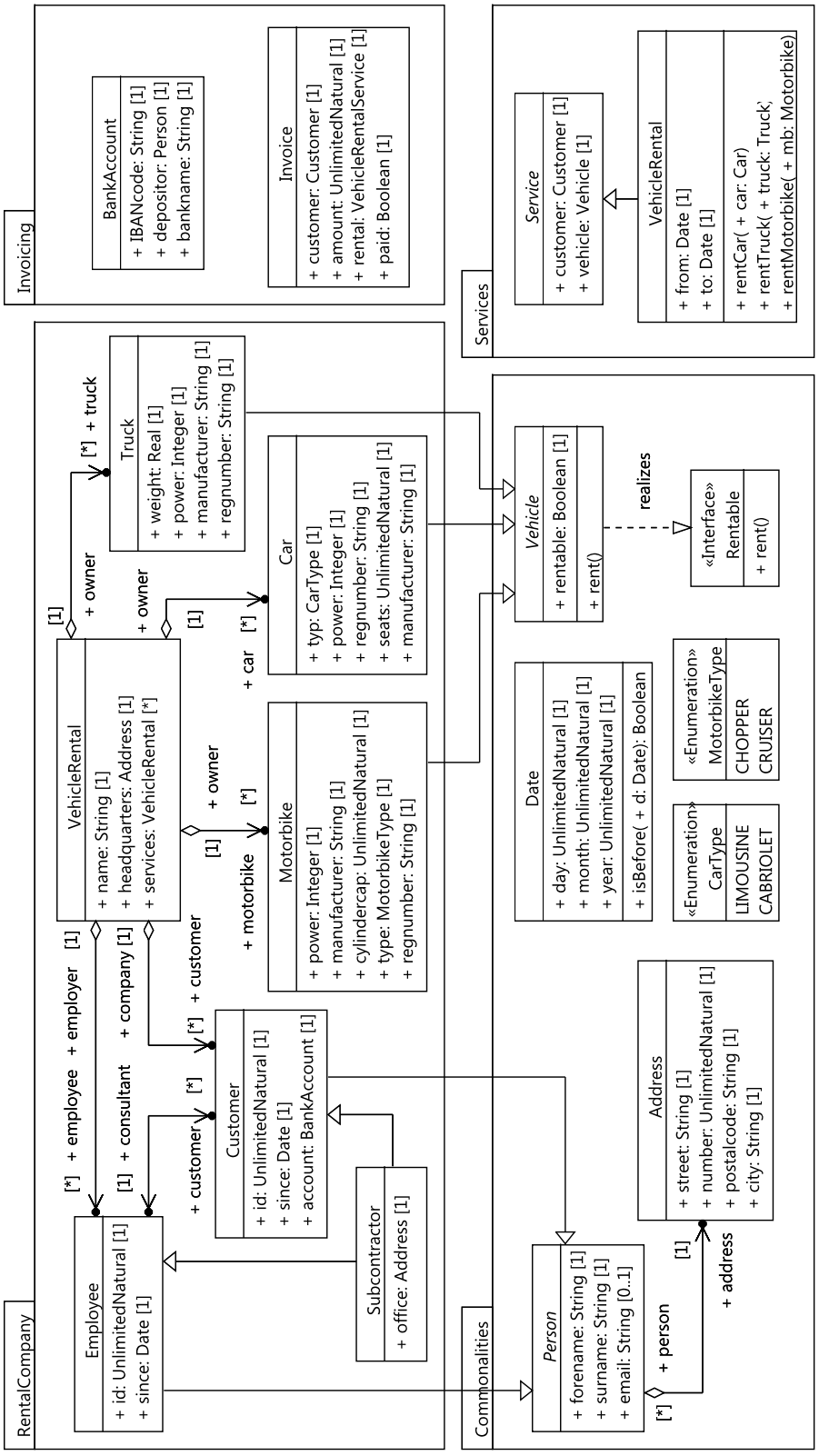


Figure 12.1: Example UML class model

For the first overview on a model, a report on project-specific model metrics might be helpful. In Sections 5.1 and 6.1 of this thesis, several metrics for UML models being useful for detecting corresponding smells have been discussed. In the following, we do not calculate this kind of smell-related metrics only but also other common metrics to get an overview on interesting model properties.

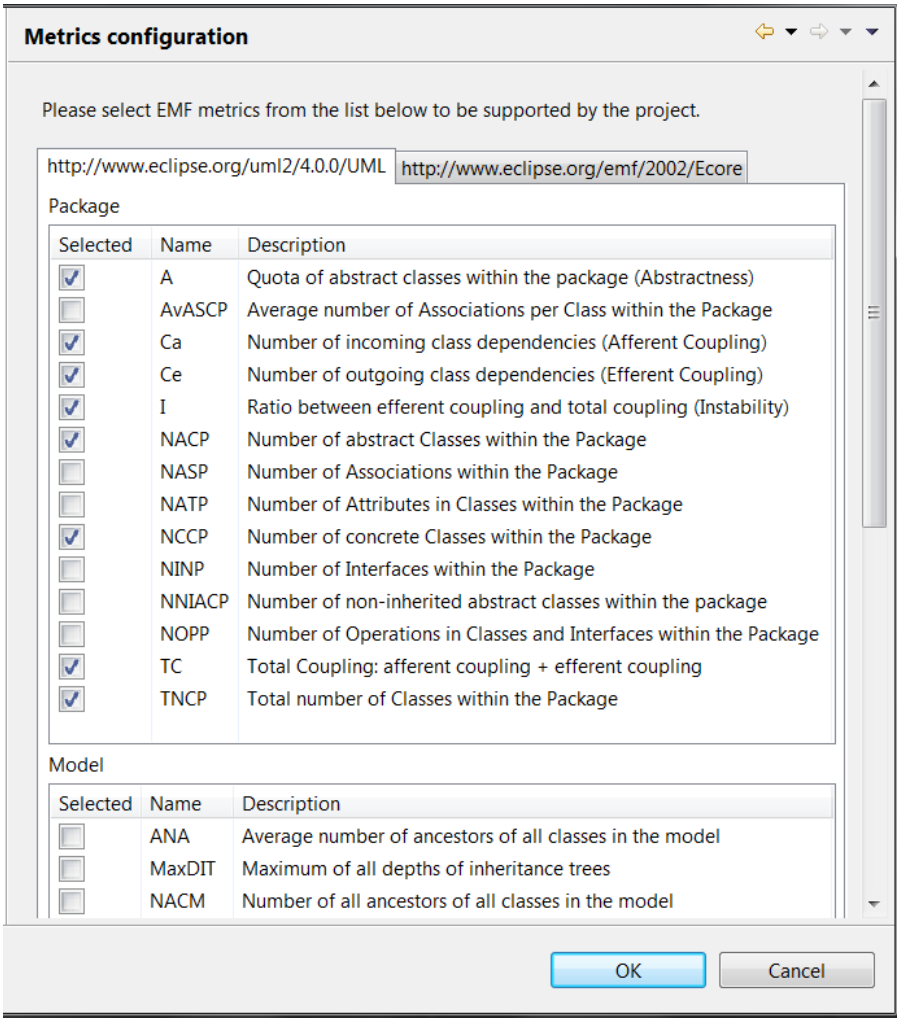


Figure 12.2: Configuration dialog for model metrics

To calculate relevant metrics only, our tool environment supports a project-specific configuration for the metrics suite[1]. Figure 12.2 shows the project-specific configuration page for our example project. On this page, all existing model metrics for EMF-based models are listed. They are structured with respect to the corresponding meta model (e.g., UML and Ecore) and to the corresponding element type the

1 Please note that the configuration task is done by the project respectively quality manager according to the project-specific needs (see Section 3.2.2).

metrics are calculated on (the *context*). In Figure 12.2 for example, we activate model metrics for UML packages concerning abstractness (`A`, `NACP`, `NCCP`, and `TNCP`) and coupling issues (`Ca`, `Ce`, `I`, and `TC`).

The calculation of metrics on a specific model element is started from its context menu. In our UML show case, this element is selected from within the graphical GMF-based [53] Papyrus editor. However, EMF Refactor also supports further editors like the tree-based EMF instance editor and textual editors generated by Xtext [62]. Figure 12.3 shows the calculated results of the configured UML metrics on packages `Commonalities` and `RentalCompany` (see Figure 12.1).

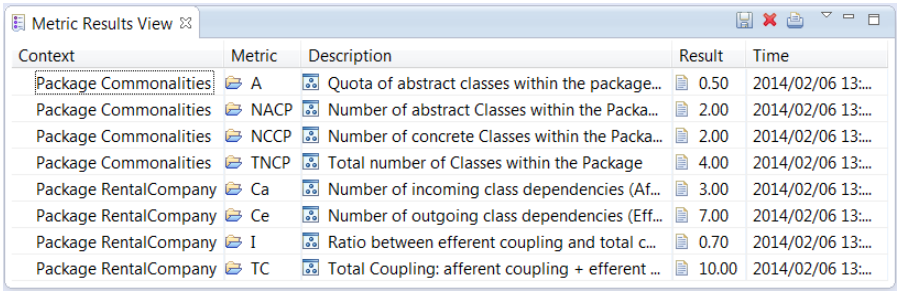


Figure 12.3: Results view displaying calculated metrics

The results view shows that package `Commonalities` contains altogether four classes (metric `TNCP`): two concrete and two abstract classes (metrics `NCCP` and `NACP`). Furthermore, metric `Ca` (afferent coupling: number of classes in other packages depending on classes of the package) of package `RentalCompany` is evaluated to 3 whereas its efferent coupling metric (number of classes within the package depending on classes in other packages - `Ce`) is evaluated to 7.

Metrics `A`, `TC`, and `I` are calculated using these 'basic' metrics. The abstractness (`A`) of package `Commonalities` is 0.5 (ratio between the number of abstract classes in the package and the total number of classes in the package). The total coupling (`TC`: afferent + efferent coupling) of package `RentalCompany` is 10 and its instability (`I`) is 0.7 (ratio between efferent coupling and total coupling).

An evaluation of both packages based on these metric values is a slightly difficult task (and is not an issue of this section rather of this thesis). However, according to [108] the following statements hold:

- The less abstract a package is the more likely it is to change and therefore to have an effect on the packages that use it.
- Instable packages are easier to change because few other packages in the application use them.

EMF Refactor's metrics tool provides the export of calculated results for reporting purposes. The following output formats are supported: XML (default), HTML, PDF, Postscript, MS DOC, MS PPT, MS

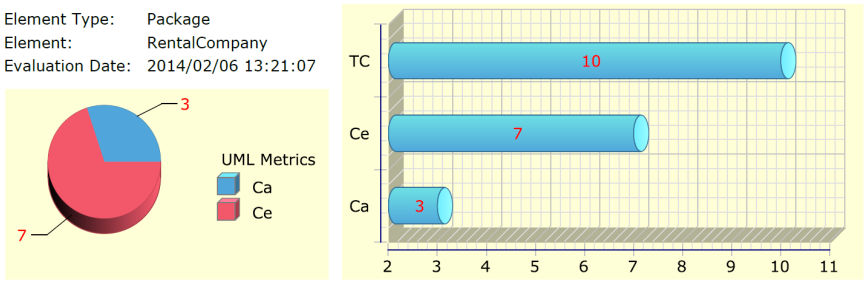


Figure 12.4: Excerpt of a generated PDF report concerning calculated metrics results using a pie diagram (left) and a tube diagram (right)

XLS, ODP, ODS, and ODT. Furthermore, several output designs are provided but also custom designs can be imported. Figure 12.4 shows two PDF exports of our example metrics calculation. On the left-hand side, metric values for `Ca` (Afferent Coupling) and `Ce` (Efferent Coupling) of package `RentalCompany` are compared using a pie diagram. The right-hand side of Figure 12.4 shows an exported tube diagram containing the metric values for `Ca`, `Ce`, and `TC` (Total Coupling).

12.3 MODEL SMELL DETECTION

The discussion of metrics results shows that a manual interpretation of metric values seems to be unsatisfactory and error-prone. So, another static model analysis technique is required, more precisely an automatic detection of model smells for UML models like specified in Sections 5.2 and 6.1, for example. As for model metrics, our tool environment provides a configuration of specific model smells that are relevant for the current project [2]. Figure 12.5 shows the configuration dialog listing all system-known model smells with respect to their meta model. For a metric-based model smell, a corresponding threshold can be configured.

In Figure 12.5, two metric-based smells are activated. Smell *Abstract Package* occurs if the value of metric `A` (Abstractness: ratio between the number of abstract classes in the package and the total number of classes in the package; see previous section) is higher than 0.7. The second metric-based smell, *Large Class*, relies on metric `NFEAC` (number of owned features of the class) and comparator $>$ (greater than). We set the limit for smell *Large Class* to `7.0`, i.e., this smell occurs if a class owns more than seven attributes attributes or operations.

Similar to the calculation process for model metrics, a smell analysis can be triggered either for the entire model or for a concrete model element. In the latter case, all smells are reported occurring within the

2 See footnote [1] on page 127.

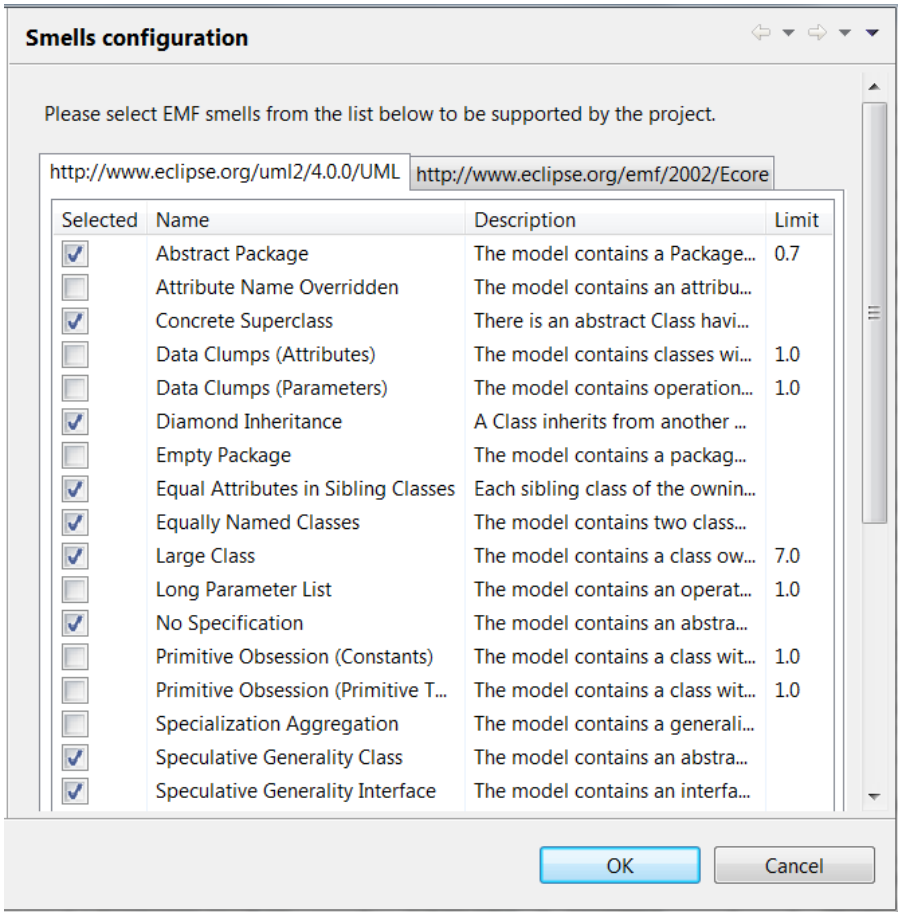


Figure 12.5: Configuration dialog for model smells

containment hierarchy of the selected model element. Nevertheless, it has to be considered that there are model smells which might be distributed along several subtrees (like *Multiple Definition of Classes with equal Names*, looking for equally named classes in different packages). However, EMF Refactor provides smell analysis on subtrees only in order to narrow the scope of the analysis, for example on large-scale models.

Analyzing the example UML class model shown in Figure 12.1, the smell detection analysis discovers the existence of altogether 19 concrete smells according to the configuration made in Figure 12.5. The left-hand side of Figure 12.6 shows the results of this analysis in a dedicated results view. The report shows that smell *Equal Attributes in Sibling Classes* occurs 13 times. Example occurrences are attributes `Motorbike::power`, `Car::manufacturer`, and `Customer::id`. Six kinds of model smells occur once each, for example smell *Speculative Generality Interface* looking for an interface that is realized by one single class only. Here, the involved elements are interface `Rentable` and class `Vehicle`.

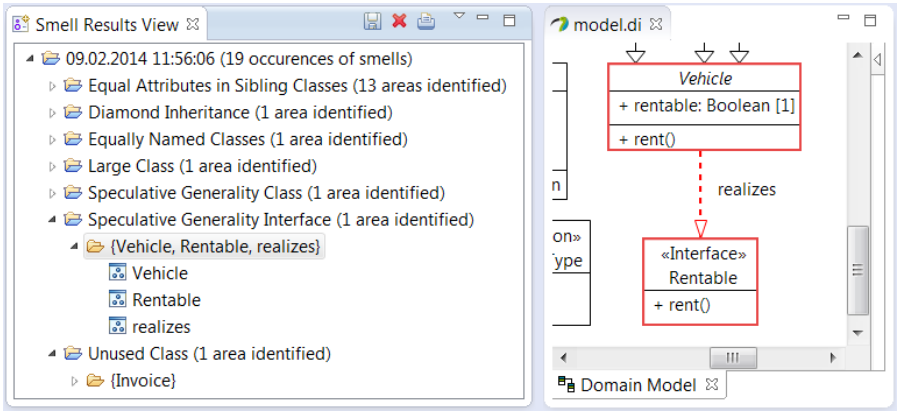


Figure 12.6: Results view displaying detected model smells (left) and highlighting of involved elements in smell *Speculative Generality* within the graphical Papyrus editor (right)

Concerning concrete smell occurrences, the smell detection tool in EMF Refactor provides a highlighting mechanism for involved model elements within the standard tree-based EMF instance editor, graphical GMF editors, and textual Xtext editors. For example, selecting the occurrence of smell *Speculative Generality Interface* in the smell view (see left-hand side of Figure 12.6) highlights interface `Rentable`, class `Vehicle`, and the realization relation between them within the graphical Papyrus editor as shown in the right-hand side of Figure 12.6.

The next step during a model review is to interpret the results of the smell detection analysis. Potential reactions on detected smells are (note that not each smell should be eliminated):

- Use refactoring *Pull Up Attribute* on attributes `manufacturer`, `power`, and `regnumber` from classes `Car`, `Motorbike`, and `Truck` to the common parent class `Vehicle`.
- Smell *Speculative Generality Class* should be removed by using refactoring *Remove Superclass* on class `Service` since the company does not offer further services.
- Rename classes `RentalCompany::VehicleRental` to *VehicleRentalCompany* and `Services::VehicleRental` to *VehicleRentalService*.
- Class `Invoice` is unused up to now. There should be an attribute named *invoices* in class `VehicleRentalCompany` with type `Invoice` and multiplicity `0..*`.

12.4 REFACTORING APPLICATION

Besides manual changes, model refactoring is the technique of choice to eliminate occurring smells. In our tool environment for model quality assurance, this task is provided by the primary functionality of

EMF Refactor. Again, this component provides a configuration mechanism to select refactorings being relevant for the given modeling project. The configuration user interface is similar to that of the metrics component (see Figure 12.2) and is not shown here. Please note that the configuration of model smells (see Section 12.3) combined with the specification of smell-refactoring relations (see Section 13.5) might influence the selection of model refactorings (and vice versa). We discuss this topic in Chapter 15.

There are two alternative ways to trigger a model refactoring in EMF Refactor: First, a refactoring can be invoked from within the context menu of at least one model element in the standard tree-based EMF instance editor, the graphical GMF-based editor, or the textual Xtext editor. Depending on the selected element(s), only those refactorings are provided in the menu being defined for the corresponding model element type(s). For example, UML refactoring *Extract Superclass* is provided only after selecting at least two classes.

The second way to trigger a model refactoring is to use the quick fix mechanism of the smell results view as shown on the left-hand side of Figure 12.6. Starting from this view, our tool environment provides a suggestion for potential refactorings according to pre-defined smell-refactoring relations (see Section 13.3) and a dynamic analysis of applicable model refactorings.

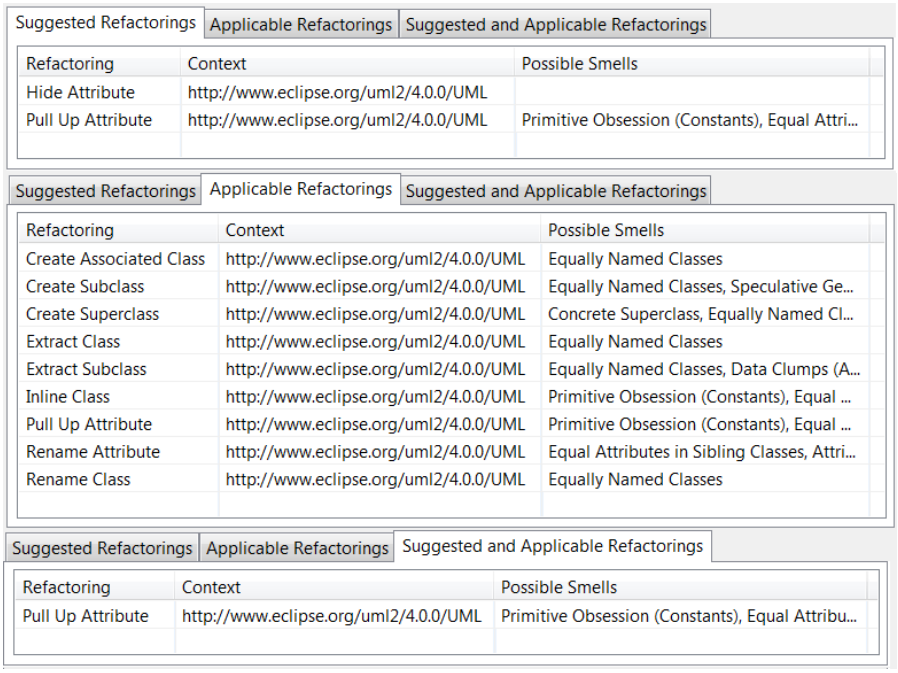


Figure 12.7: Quick fix mechanism: manually defined refactorings (top), actually applicable refactorings (middle), and manually defined applicable refactorings (bottom)

The suggestion dialog is started from within the context menu of a smell occurrence (e.g., occurrence *{Motorbike, power}* of smell `Equal`

`Attributes in Sibling Classes`) and consists of three tabs. The first tab (see top of Figure 12.7) suggests all model refactorings that have been manually defined as being suitable to erase the corresponding model smell. The second tab (see middle of Figure 12.7) lists all those model refactorings which have been proven to be applicable on at least one model element in the selected smell occurrence. Please note that this does not necessarily mean that each presented refactoring would improve the model quality by erasing a model smell. It simply means that the target model structure allows the application of that refactoring. For example, refactoring `Rename Class` obviously does not influence the afore mentioned smell. The third tab (see bottom of Figure 12.7) combines the manually defined solution and the actually applicable solutions. Finally, each tab informs about possible new smells potentially inserted when applying the refactoring (according to the manual configuration (see Section 13.4).

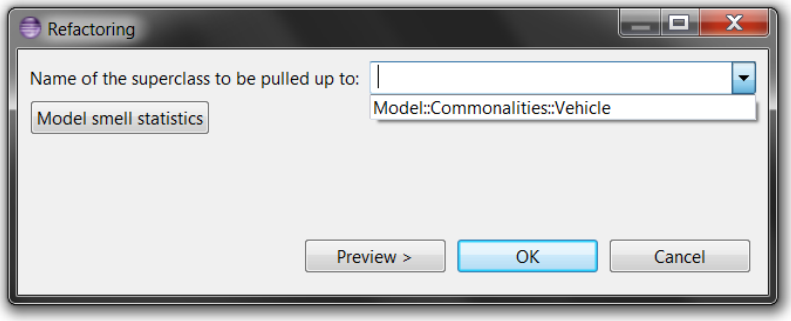


Figure 12.8: Parameter input dialog of UML refactoring *Pull Up Attribute*

After invoking a refactoring, either from within an editor or by the provided quick fix mechanism, refactoring-specific basic conditions are checked (initial precondition check). Then, the user has to set all needed parameters. Figure 12.8 shows the parameter input dialog for refactoring *Pull Up Attribute* that is invoked on attribute `power` of class `Motorbike`. Due to multiple inheritance in UML, the superclass to which the attribute should be moved must be set.

Then, EMF Refactor checks whether the user input does not violate further conditions (final precondition check). In case of erroneous parameter input a detailed error message is shown. If the final check has passed, a preview of model changes to be performed by the refactoring is provided using EMF Compare [45].

Besides the model change preview, EMF Refactor provides the opportunity to get a quantitative analysis on changes of smell occurrences. In contrast to the manual configuration of potential refactoring-smell-relations (see Section 13.4), this analysis provides the modeler with the total number of occurrences of model smells before and after a potential application of a given model refactoring. It thereby helps

with the decision whether or not a refactoring application would improve the overall model quality or would it even make worse.

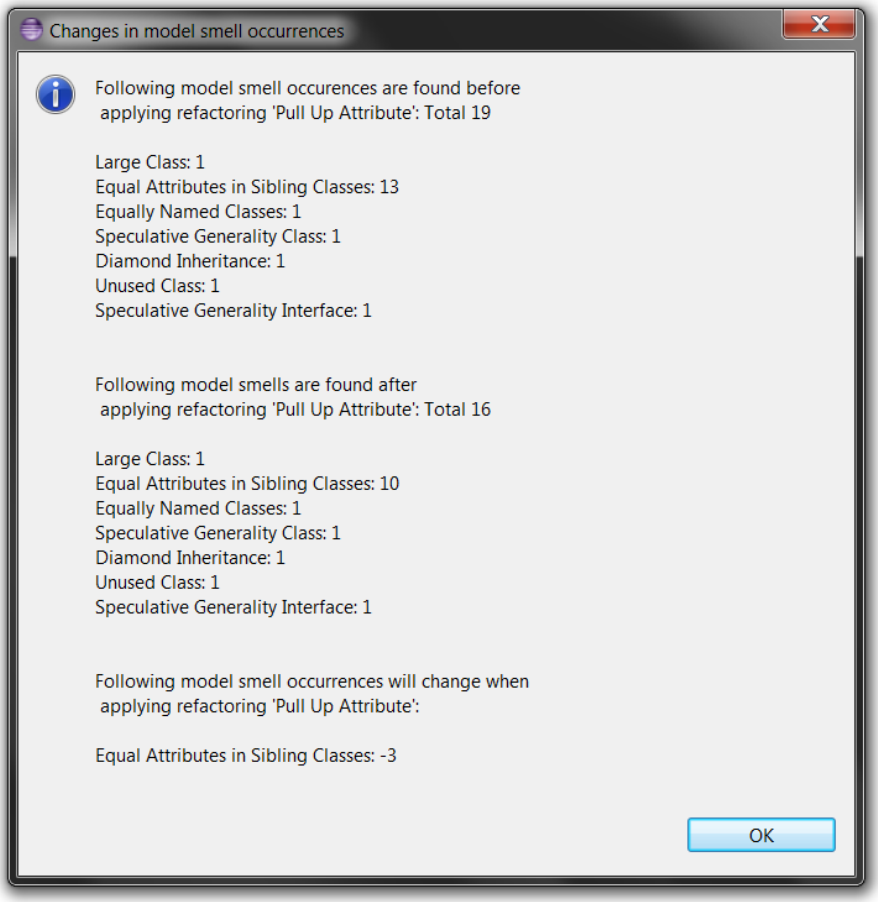


Figure 12.9: Smell analysis during the application of UML refactoring *Pull Up Attribute* on attribute *Motorbike::power*

Figure 12.9 shows the information dialog when applying UML refactoring *Pull Up Attribute* on attribute `Motorbike::power`. Before the refactoring, UML model smell *Equal Attributes in Sibling Classes* occurs 13 times; after the refactoring three occurrences would be eliminated (since attribute *power* is pulled up from each subclass of `Vehicle`, i.e., classes `Motorbike`, `Car`, and `Truck`). Moreover, no further smell would be inserted. However, 10 occurences of smell *Equal Attributes in Sibling Classes* as well as the single occurrences of smells *Large Class*, *Equally Named Classes*, *Speculative Generality Class*, *Diamond Inheritance*, *Unused Class*, and *Speculative Generality Interface* would remain. Finally, all model changes can be committed and the refactoring is performed.

Figure 12.10 shows the example UML class model after performing several model changes, being refactorings and manual changes, as described at the end of Section 12.3. Now, class `Vehicle` owns the afore redundant attributes *manufacturer*, *power*, and *regnumber*.

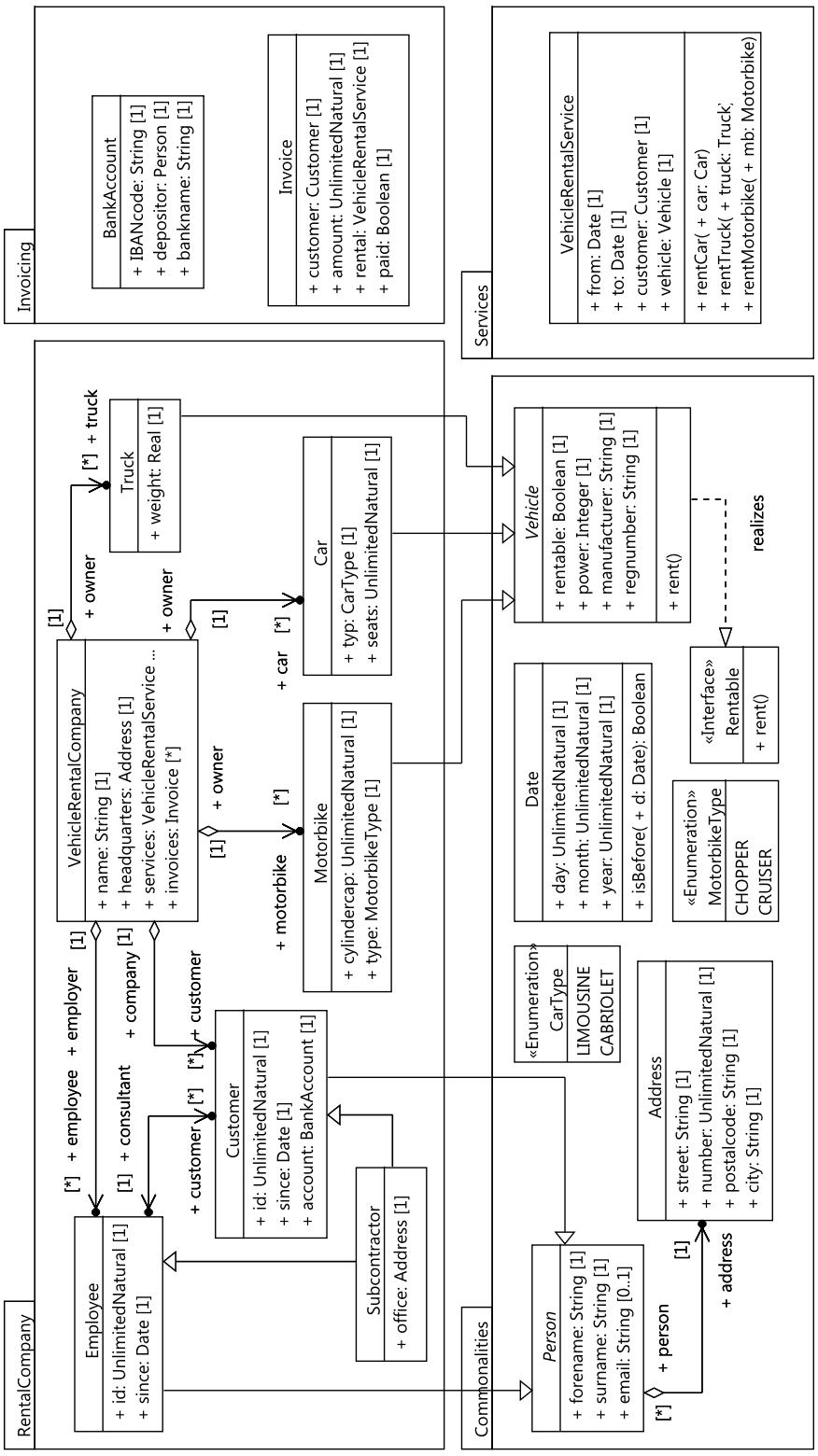


Figure 12.10: Example UML class model after several model changes as result of a first model review

Class `Service` has been removed so that `VehicleRentalService` (formerly named *VehicleRental*) is the only offered service left. Finally, the main class `VehicleRentalCompany` (also formerly named *VehicleRental*) has a new attribute *invoices* with type `Invoice` and multiplicity `0..*`.

From the detected smells seven occurrences are left. However, there are model parts remaining suspicious with respect to several model quality aspects. For example, there are associations from class `Company` to classes `Car`, `Truck` and `Motorbike` hinting to some kind of redundant modeling. This shows that project-specific model quality assurance techniques need not be completely defined (and implemented) before a project starts. The quality assurance process should be refined during the model development phase in order to be steadily improved. For example UML model smell *Association Clumps* as well as refactoring *Pull Up Association* would extend the suite of project-specific model quality assurance techniques in a meaningful way. How the specification of new model assurance techniques is supported by EMF Refactor is shown in the next chapter of this thesis.

13

EXAMPLE SPECIFICATIONS

EMF Refactor, the tool environment for EMF model quality assurance, provides a wizard-based specification process for each supported quality assurance technique. This chapter presents several concrete specification mechanisms for model quality assurance techniques. The techniques and mechanisms are discussed along a domain-specific modeling language for defining web applications. After introducing the DSL in Section 13.1, the subsequent sections discuss how to define new model metrics (Section 13.2), new model smells (Section 13.3), and new model refactorings (Section 13.4). Finally, Section 13.5 shows how to manually define relations between model smells and model refactorings. [1]

13.1 EXAMPLE DSL SIMPLE WEB MODEL (SWM)

To demonstrate the specification facilities provided by EMF Refactor, we use a domain-specific modeling language (DSML) called Simple Web Model (SWM) for defining a specific kind of web applications. This language has been already used in Section 6.2 to demonstrate a proof-of-concept implementation of the quality assurance process defined in Part I of this thesis.

However, for a better understanding we repeat the example scenario (taken from [21]): A software development company is repeatedly building simple web applications being mostly used to populate and manage persistent data in a database. Here, a typical three-layered architecture following the Model-View-Controller (MVC) pattern [68] is used. As implementation technologies, a relational database for persisting the data as well as plain Java classes for retrieving and modifying the data are employed for building the model layer. Apache Tomcat is used as the Web Server, and the view layer, i.e., the user interface, is implemented as Java Server Pages and the controller layer is realized as Java Servlets. The company decides to develop its own DSML called Simple Web Modeling Language (SWM) for defining their specific kind of web applications in a platform-independent way. Furthermore, platform-specific models following the MVC pat-

1 Details on implemented smells and refactorings for UML class models can be found in Appendices D and F of this thesis.

tern should be derived with model transformations from which the Java-based implementations are finally generated.

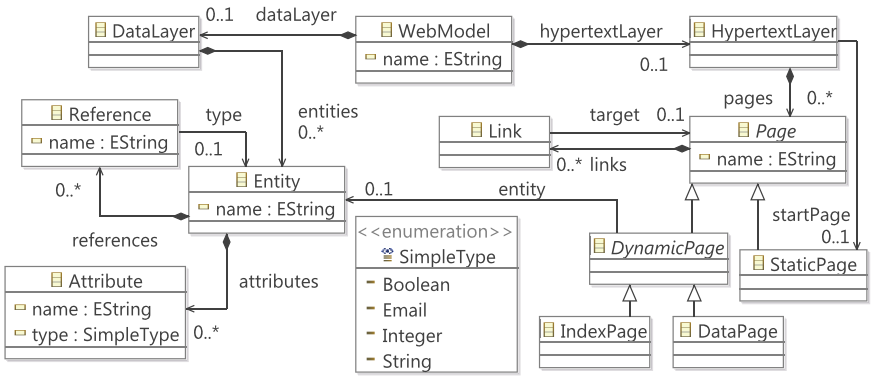


Figure 13.1: SWM meta model defined in Ecore

Figure 13.1 shows the language description of SWM as meta model modeled in EMF Ecore. A `WebModel` consists of two parts: a `DataLayer` for modeling entities which should be persisted in the database (see left-hand-side of Figure 13.1), and a `HypertextLayer` presenting the web pages of the application (see right-hand-side of Figure 13.1). An `Entity` owns several `Attributes` (each having a `SimpleType`) and can be related to several other entities (see meta class `Reference`). A `Page` is either a `StaticPage` having a static content or a `DynamicPage` having a dynamic content depending on the referenced entity. An `IndexPage` lists objects of this entity whereas a `DataPage` shows concrete information on a specific entity like its name, attributes, and references. Pages are connected by `Links`.

The following sections show how to implement quality assurance techniques for SWM which have been already introduced in Section 6.2. We start with the specification of metrics for the SWM language.

13.2 Specification of New Model Metrics

For the specification of model metrics, EMF Refactor supports four concrete technologies. As basic approaches, pure Java code using the modeling language API generated by EMF and OCL expressions can be used. Another approach is to define a pattern using the abstract model syntax first and to count its occurrences in a concrete model thereafter. These patterns are formulated as rules in a language included in the EMF model transformation tool Henshin [4, 54]. To define compositional metrics, the tool environment supports a com-

bination of existing ones. Here, the involved metrics as well as appropriate arithmetic operations have to be specified.

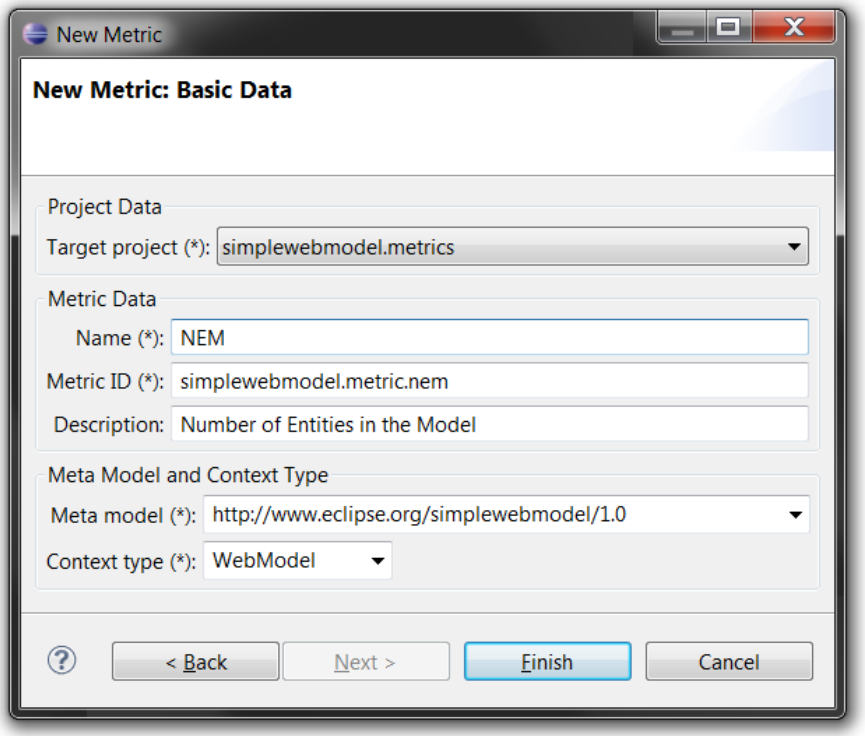


Figure 13.2: Wizard dialog for the specification of new model metrics

Figure 13.2 shows an example wizard dialog concerning the specification of SWM metric `NEM` (Number of Entities in the Model). After inserting metric-specific information like the name or the corresponding meta model and context type information, EMF Refactor generates metric-specific Java code and extends the list of supported model metrics using the extension point technology of Eclipse. Now, the metrics designer has to complete this code by the actual metrics calculation algorithm. As a result, we obtain a module with all metrics features as described in Section 12.2.

```
1 WebModel in = (WebModel) context.get(0);
2 double ret = 0.0;
3 // begin custom code
4 ret = in.getDataLayer().getEntities().size();
5 // end custom code
6 return ret;
```

Listing 13.1: Completed Java specification for SWM metric `NEM`

Listing 13.1 shows the completed Java code snippet specifying SWM metric `NEM`. Starting from the contextual `WebModel` element, the Java API of SWM generated by EMF is used. According to the SWM meta

model in Figure 13.1, a web model owns a `DataLayer` for modeling entities which should be persisted in the database. The custom code in Listing 13.1 simply navigates to the set of entities within the data layer and returns its size (see line 4 in Listing 13.1).

Since EMF models can be queried well using the Object Constraints Language [121], EMF Refactor supports model metrics specifications formulated as OCL queries. Listing 13.2 shows two alternative OCL expression being suited to calculate SWM metric `NDPM` (Number of Dynamic Pages in the Model). The first expression (lines 1 and 2) navigates from the contextual element (represented by the OCL variable *self* of type `WebModel`) to the set of pages within the hypertext layer of the model, selects those being of type `DynamicPage` (since there might be also static pages), and returns their number. The second alternative (line 4) uses the *allInstances()* operation of the OCL standard library to get a set consisting of all dynamic pages in the model and also returns their number.

```
1 String oclExpression_v1 = "self.hypertextLayer.pages " +
2         "-> select(oclIsKindOf(DynamicPage)) -> size()";
3
4 String oclExpression_v2 = "DynamicPage.allInstances() -> size()";
```

Listing 13.2: Two alternative OCL specifications for SWM metric `NDPM` (Number of Dynamic Pages in the Model)

To insert the OCL query during the specification process in addition to the basic data similar to Figure 13.2, the specification wizard provides a dedicated input page after selecting the *OCL* specification mode. Finally, EMF Refactor generates the complete metric-specific Java code. Here, the contextual element (an instance of `WebModel`) as well as the specified expression (represented as *String*) are passed to the OCL adapter as presented in Section 11.2.

As discussed in Section 6.2.2, a metric calculating the ratio between the values of both metrics presented before might be helpful to detect missing dynamic pages. For defining these kind of metrics, the specification wizard provides a dedicated page after selecting specification mode *Composite*. Here, the metric designer simply selects the involved existing metrics as well as the appropriate arithmetic operation. Here, the binary arithmetic operations sum, subtraction, multiplication, and division are supported.

In our example, for specifying SWM metric `DPpE` (Dynamic Pages per Entity), metrics `NEM` (Number of Entities in the Model) and `NDPM` (Number of Dynamic Pages in the Model) are combined using the binary arithmetic operation division (see Figure 13.3). To be consistent, the dialog page presents only those metrics whose contextual elements correspond to the contextual element of the new composi-

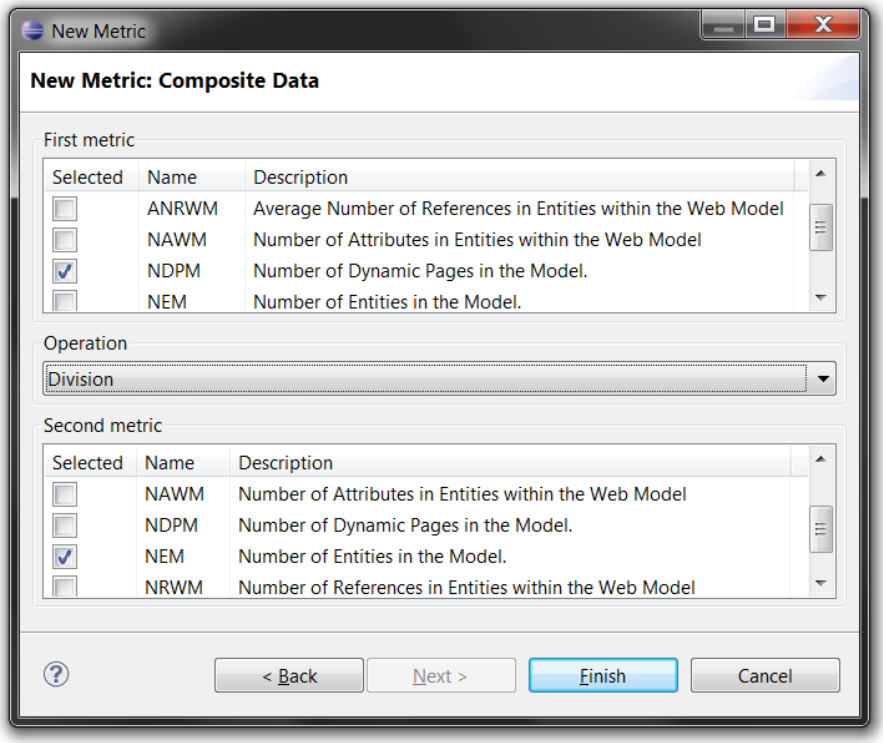


Figure 13.3: Compositional specification for SWM model metric *DPpE* (Dynamic Pages per Entity)

tional metric (`WebModel` in our example). Again, EMF Refactor finally generates the complete metric-specific Java code and extends the list of supported model metrics for SWM models.

As a last supported specification mechanism for EMF model metrics we present the use of Henshin pattern rules formulated on the abstract syntax on SWM. Figure 13.4 shows a Henshin pattern rule specifying SWM metric `NDPE` (Number of Dynamic Pages referencing the Entity) using the graphical syntax of Henshin. The left node *context* of type `Entity` represents the contextual model element for calculating metric `NDPE` whereas the remaining rule elements represent the pattern that has to be found in the model. The pattern defines a node *referencingPage* type `DynamicPage` that references the contextual entity by reference `entity`. It is formulated as positive application condition (PAC) (see rule elements annotated with ⟨⟨*require#reference*⟩⟩).

To calculate metric `NCCP`, the Henshin adapter of the metrics tool uses the Henshin interpreter to find and count matches of this pattern rule on concrete SWM instance models. Please note that this adapter requires the following guidelines for Henshin pattern rule specifications to work properly:

- The pattern rule must be named *mainRule*.

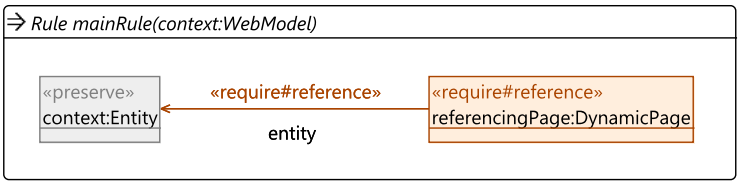


Figure 13.4: Henshin pattern rule specifying SWM model metric *NDPE*

- The rule must have a parameter named *index*.
- The contextual node must be named *index*.

For defining these metrics specified in Henshin, the specification wizard provides a dedicated import page for the appropriate Henshin file. As in the cases described before, EMF Refactor finally generates the complete metric-specific Java code and extends the list of supported model metrics for SWM models.

The metrics defined in this section may help to analyze the completeness of SWM models . However, to make suspicious model parts more explicit, the next section shows how to specify model smells for SWM models.

13.3 SPECIFICATION OF NEW MODEL SMELLS

EMF Refactor supports four concrete mechanisms for model smell specification. Again, pure Java code and OCL expressions can be used as basic approaches. Some smells can be detected well by metric benchmarks. Here, appropriate model metrics are used together with suitable benchmarks being set by project-specific configurations. Pattern-based smells (i.e., smells that are detectable by the existence of specific anti-patterns) can be specified by Henshin rules.

The specification process for model smells is similar to the specification process for model metrics. For each specification mode, the dialog page for inserting basic model smell data looks similar to the page shown in Figure 13.2. The only difference is the missing context element type definition since a smell does not have such a context.

After inserting smell-specific information like the name or the corresponding meta model (given by its *nsURI*, see Section 10.1), EMF Refactor generates Java code and extends the list of supported model smells using the extension point technology of Eclipse. In the case of selecting the *Java* specification mode, the corresponding generation module generates a skeleton implementation that has to be completed by the model smell designer.

Listing 13.3 shows the core Java specification of SWM smell `No Dynamic Page`. According to Section 6.2.2, this smell occurs if the

model contains an entity which is not referenced by a dynamic page, i.e., the entity would not be depicted in the web application. The condition in the if-clause checks whether the given entity is referenced by a dynamic page (line 9). In this case, the boolean flag is set (line 10). Finally, if no page references the entity, i.e., the boolean flag is not set, a new `SmellOccurrence` object is created, the entity is added to it, and the object is added to the list of found model smells (lines 14 to 16). [2]

```
1  LinkedList<SmellOccurrence> results =
2                  new LinkedList<SmellOccurrence>();
3  // begin custom code
4  List<Entity> entities = getAllEntities(root);
5  List<DynamicPage> dynamicPages = getAllDynamicPages(root);
6  for (Entity entity : entities) {
7          boolean isReferenced = false;
8          for (DynamicPage dynamicPage : dynamicPages) {
9                  if (dynamicPage.getEntity() == entity) {
10                         isReferenced = true;
11                 }
12         }
13         if (! isReferenced) {
14                 SmellOccurrence result = new SmellOccurrence();
15                 result.addEObject(entity);
16                 results.add(result);
17         }
18 }
19 // end custom code
20 return results;
```

Listing 13.3: Java specification of SWM model smell *No Dynamic Page*

The use of OCL is also an adequate approach to specify model smells. Listing 13.4 shows the OCL specification of SWM smell `Empty Entity`. It defines an OCL operation that returns from the set of all entities in the web model (line 3) those which have neither attributes nor references (lines 4 and 5).

```
1  context WebModel
2  def: emptyEntity(): Set(Entity) =
3  Entity.allInstances() -> select (entity|
4          entity.attributes -> isEmpty() and
5          entity.references -> isEmpty() )
```

Listing 13.4: OCL specification of SWM model smell *Empty Entity*

In this specification mode, the specification wizard provides a dedicated input page for the appropriate OCL file. Moreover, the model

2 Please note that the code snippet is not complete since we use auxiliary methods *getAllEntities()* and *getAllDynamicPages()* which are not discussed in detail here.

smell designer has to specify the name of the operation to be executed by the corresponding OCL adapter. Then, EMF Refactor generates the complete smell-specific Java code and extends the list of supported model smells for SWM models.

As mentioned at several places throughout this thesis, some model smells can be detected by matching a corresponding pattern based on the abstract syntax of the modeling language. A representative of this kind of smells concerning the SWM language is `Equally Named Pages`. This smell detects pages within the hypertext layer having the same name. Such redundant page names potentially lead to inconsistent code that is generated from the model.

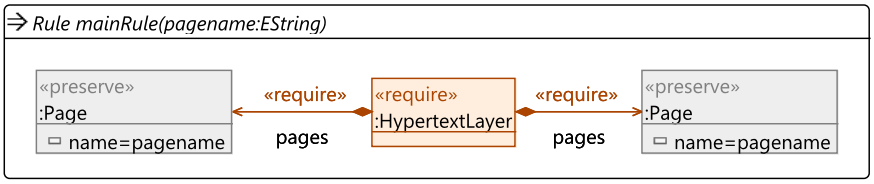


Figure 13.5: Henshin pattern rule specification for SWM model smell *Equally Named Pages*

Figure 13.5 shows a Henshin pattern rule defining smell `Equally Named Pages`. The pattern specifies two pages that must be found in the model (tagged by ⟪preserve⟫) and the containing hypertext layer as PAC (tagged by ⟪require⟫). Furthermore, the rule owns a parameter named *pagename* of type `EString`. It is used for specifying that the meta attributes *name* of each page node have the same value, i.e., the pages have the same name.

The smell detection tool in EMF Refactor uses Henshin's pattern matching algorithm to detect rule matches. Please note that the pattern rule must be named *mainRule* in order to be executed by the Henshin adapter. Then, the matches found represent the existence of model smells in the model. If the pattern is matched its preserved nodes represent those model elements which are involved in this specific smell occurrence. The specification wizard for pattern-based smells provides an import page for the appropriate Henshin file and finally generates the complete smell-specific Java code and extends the list of supported model smells for SWM models.

As discussed in Sections 6.2.2 and 13.2, metric `DPpE` (Dynamic Pages per Entity) might be helpful to detect a SWM model smell. If its value is less than 2, i.e., an entity is not referenced by at least two dynamic pages on average, this might be a hint for missing dynamic pages. This metric-based smell is called `Insufficient Number of Dynamic Pages` in the following.

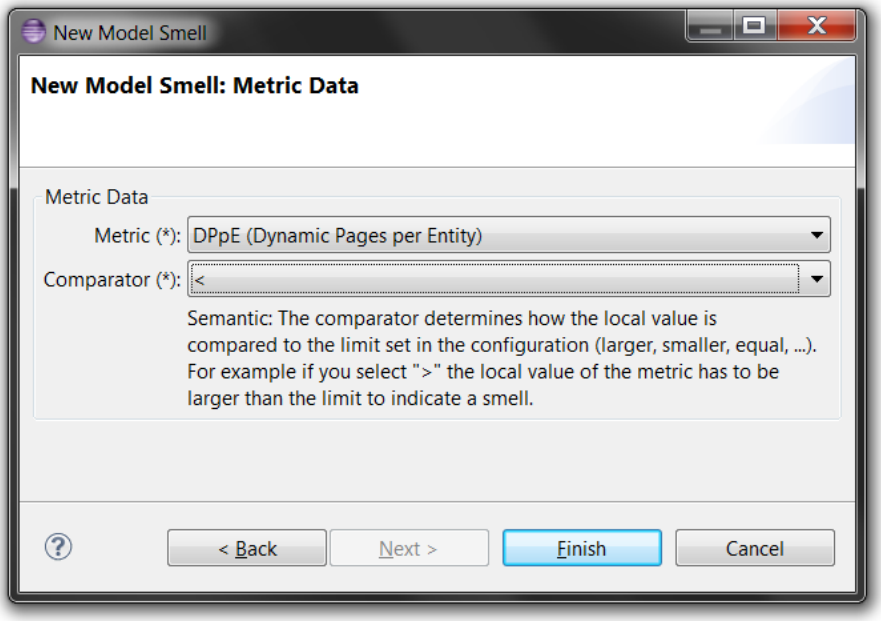


Figure 13.6: Specification of SWM model smell *Insufficient Number of Dynamic Pages* using metric *DPpE* (Dynamic Pages per Entity)

For the specification of metric-based model smells, EMF Refactor provides a dedicated specification wizard. On a specific page, the metric designer simply selects the corresponding metric as well as the appropriate comparator. For specifying e.g. smell `Insufficient Number of Dynamic Pages`, metric `DPpE` is combined with comparator $<$ as discussed above. Figure 13.6 shows the corresponding wizard page. Please note that the threshold value is not pre-set. This is done in the project-specific configuration as described in Section 12.3.

Throughout this thesis, refactoring is the technique of choice for eliminating model smells. So, after having specified appropriate model smells, the following section demonstrates how to define suitable refactorings in order to support the handling of *smelly* SWM models.

13.4 Specification of New Model Refactorings

Since EMF Refactor uses the LTK technology [67] as described in Section 11.2, a concrete refactoring specification requires up to three parts (i.e., specifications for initial checks, final checks, and the proper model changes). EMF Refactor supports three concrete mechanisms for EMF model refactoring specification. As for metrics and smells, refactorings can be specified using Java and the language API generated by EMF. A way to specify a model refactoring straight forwardly is to use Henshin. Finally, existing refactorings can be combined to more complex ones by using a domain-specific language, called CoM-

ReL (Composite Model Refactoring Language), as presented in Section 7 of this thesis.

The dialog for the specification of a new model refactoring starts with a page for inserting basic data like the name of the refactoring or the corresponding meta model and context type information.

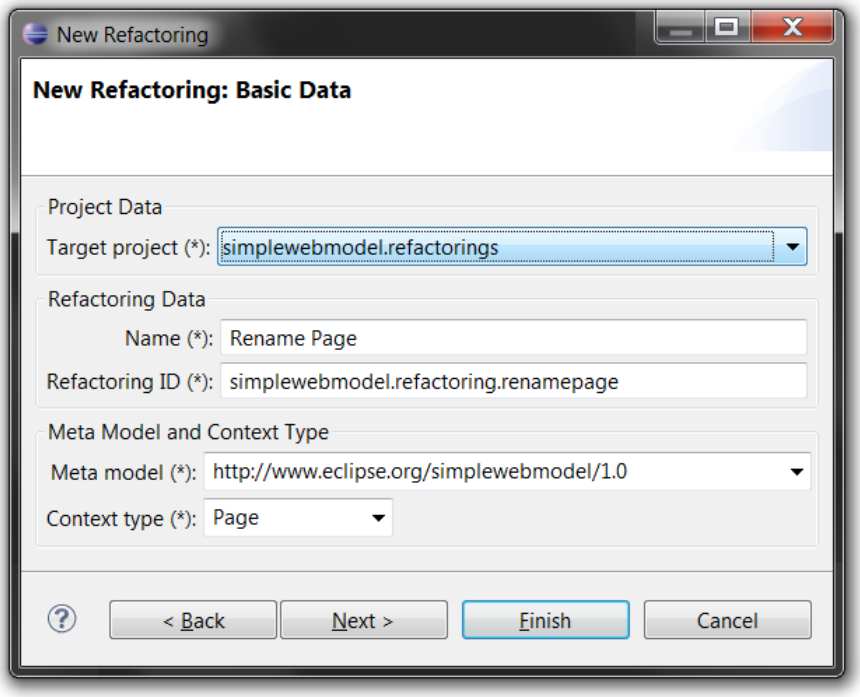


Figure 13.7: Wizard dialog for the specification of new model refactorings

A standard refactoring that should be provided for a DSML is the renaming of model elements. For example, refactoring `Rename Page` can be used to eliminate SWM model smell `Equally Named Pages` as discussed in the previous section. Figure 13.7 shows the dialog page for inserting basic information. After inserting these data, potential refactoring-specific parameters are defined using the second page of the specification wizard.

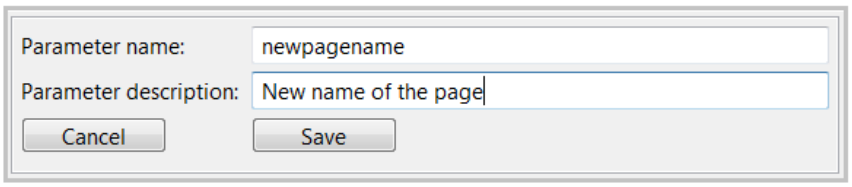


Figure 13.8: Parameter input specification of SWM model refactoring *Rename Page*

Besides the contextual element of type `Page`, refactoring `Rename Page` has one more parameter: the new name of the page. It is speci-

fied as shown in Figure 13.8. In addition to the name of the parameter, *newpagename*, a parameter description can be defined that will be used later on in the parameter input dialog during the refactoring.

If specification mode Java is selected, the generated refactoring-specific code contains three passages indicating those parts of the refactoring specification that have to be completed (i.e., specifications for initial checks, final checks, and the proper model changes).

```
1  RefactoringStatus result = new RefactoringStatus();
2  Page selectedEObject = (Page) getValue("selectedEObject");
3  String newpagename = (String) getValue("newpagename");
4  // begin custom code
5  List<Page> pages = getAllPages(root);
6  for (Page page : pages) {
7          if (page != selectedEObject &&
8              page.getName().equals(newpagename)) {
9                  result.addFatalError("There is already a page" +
10                         " named '" + newpagename + "'!");
11         }
12 }
13 // end custom code
14 return result;
```

Listing 13.5: Final precondition check of SWM model refactoring *Rename Page*

Refactoring `Rename Page` does not require any initial precondition checks. However, after inserting the new name of the page, a final check has to ensure that there is no page already having this name. Listing 13.5 shows the core Java specification of this check. Here, the name of each page (except for the contextual one) is compared to the specified name given by the parameter input (lines 7 and 8). If there is already a page with this name, an individual error message is added to the refactoring observer (object of LTK class `RefactoringStatus`; see lines 9 and 10 in Listing 13.5).

```
1  Page selectedEObject = (Page) getValue("selectedEObject");
2  String newpagename = (String) getValue("newpagename");
3  // begin custom code
4  selectedEObject.setName(newpagename);
5  // end custom code
6  return result;
```

Listing 13.6: Model change specification of SWM model refactoring *Rename Page*

Listing 13.6 shows the model change specification of refactoring `Rename Page`. Here, the custom code simply changes the name of the

contextual page object (*selectedEObject*) to the inserted one (line 5).[3]

A prominent way to specify EMF model refactorings is to use the model transformation language Henshin since refactorings can be seen as a specific kind of in-place model transformations. Here, EMF Refactor uses Henshin's model transformation engine for executing the refactoring as well as Henshin's pattern matching algorithm to detect violated preconditions.

In the previous section, SWM model smell `No Dynamic Page` indicating an entity which is not referenced by a dynamic page has been discussed. According to Section 6.2.2, this smell can be eliminated by a refactoring which inserts both an index page and a data page referencing the corresponding entity to the hypertext layer. This refactoring, called `Insert Dynamic Pages`, is called from an entity (the contextual element) and can be applied only if this entity is not referenced by a dynamic page already (initial precondition check). The names of both new pages can be derived from the name of the entity by appending *Index* and *Data*, respectively. So, the refactoring does not have any further parameters. Therefore, there are no final preconditions to be checked.

In EMF Refactor, the specification of precondition checks using Henshin rules is done in a similar way as for specifying model smells (see previous section). Each scenario that violates the corresponding precondition is specified in a separate Henshin rule in order to present reasonable error messages to the user. Here, such messages are encoded as descriptions of the corresponding rule. If the rule matches, i.e., the precondition is violated, the Henshin adapter passes the message to the `RefactoringStatus` object.

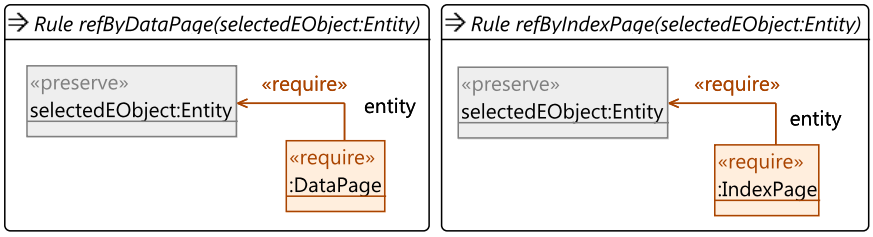


Figure 13.9: Henshin rule specification for the initial precondition check of SWM model refactoring *Insert Dynamic Pages*

Figure 13.9 shows two Henshin rules defining potential precondition violations of refactoring `Insert Dynamic Pages`. In both rules, node *selectedEObject* of type `Entity` represents the contextual model element. The rule on the left-hand side specifies that the contextual

3 Though renaming is a simple but common refactoring, the implementation of `Rename Page` in Listings 13.5 and 13.6 show that it is hard to define EMF-based refactoring in a generic way due to the diversity of precondition checking with respect to the given DSML (compare [132, 32]).

entity is referenced by a data page whereas the rule on the right-hand side specifies that it is referenced by an index page. Please note that both rules can be combined to one single rule using a node of type `DynamicPage` instead of two nodes of type `DataPage` respectively `IndexPage`. However, when using two rules the corresponding error messages are more meaningful (*There is already a data/index page referencing this entity!* compared to *There is already a dynamic page referencing this entity!*).

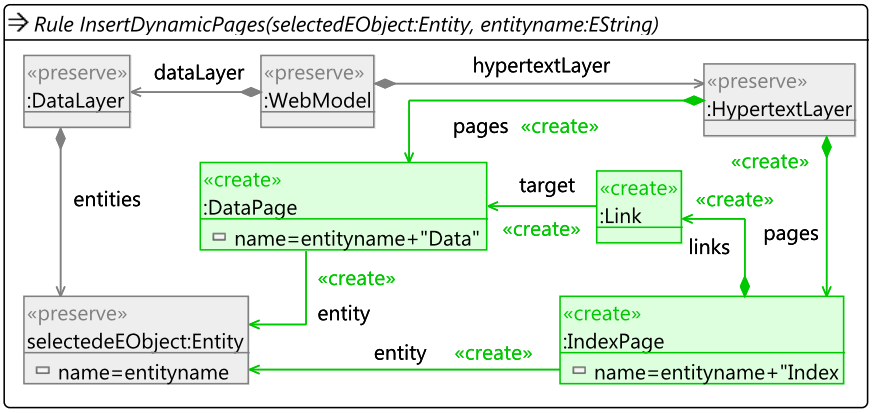


Figure 13.10: Henshin rule specification for the model change part of SWM model refactoring *Insert Dynamic Pages*

Figure 13.10 shows the Henshin rule specifying the model change part of SWM refactoring `Insert Dynamic Pages`. Nodes and edges tagged by ⟪preserve⟫ represent unchanged model elements whereas those tagged by ⟪create⟫ represent new ones. The rule inserts both a new data page and a new index page into the hypertext layer of the model. The names of the new pages are set as described above. Here, the rule uses an internal parameter *entityname* whose value is set by the match of the contextual entity (node *selectedEObject*). Finally, a link between the inserted index page and the new data page is inserted.

Please note that the Henshin adapter requires the following guidelines for refactoring specifications to work properly:

- The unit to be executed must be named *mainUnit*.
- The main unit must have a parameter named *selectedEObject*.
- The contextual node must be named *selectedEObject*.

In the specification process for refactorings specified in Henshin, the specification wizard provides an import page for the appropriate Henshin file(s) and finally generates the complete refactoring-specific Java code and extends the list of supported model refactorings for SWM models.

EMF Refactor supports to combine existing refactorings to more complex ones. Here, a domain-specific language, CoMReL (Composite Model Refactoring Language), is provided which implements the concepts discussed in Section 7 of this thesis.

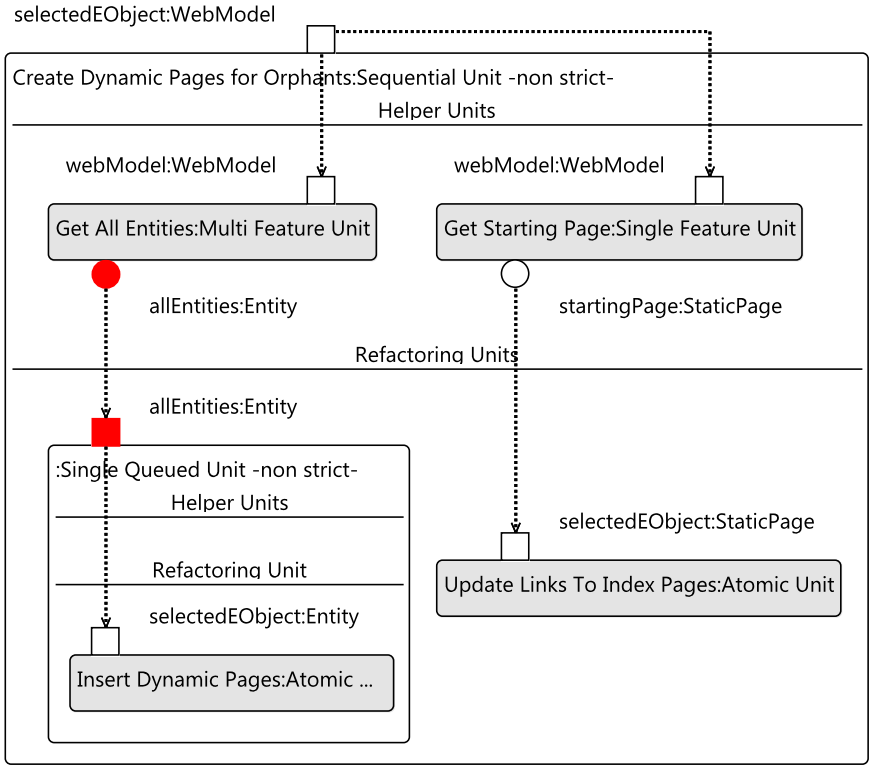


Figure 13.11: Unit specification of composite SWM model refactoring *Create Dynamic Pages for Orphants*

Figure 13.11 shows a visual representation of the specification model of refactoring *Create Dynamic Pages for Orphants*. This refactoring can be used to insert both an index page and a data page for each entity in the model which is not referenced by a dynamic page yet. The main refactoring unit *Create Dynamic Pages for Orphants* is a non-strict `Sequential Unit` consisting of a `SingleQueuedUnit` and a `AtomicUnit`. The `SingleQueuedUnit` is applied on each entity of the contextual web model. Here, the entities are obtained by helper unit *Get All Entities*. Finally, refactoring *Update Links To Index Pages* is applied on the starting page of the model (which is also obtained by an appropriate helper). This additional refactoring is required since the specification of the refactoring does not consider to add a link from the starting page to the newly created index page (see Figure 13.10).

The specification process for refactorings specified in CoMReL is similar to that for Henshin. Here, the specification wizard provides an import page for the appropriate CoMReL file and finally generates the complete refactoring-specific Java code and extends the list

of supported model refactorings for SWM models.

In this section, we specified several refactorings which are suited to erase model smells specified in the previous section. The following section demonstrates how this relation can be made more explicit within the EMF Refactor tool set.

13.5 Specification of Smell-Refactoring Relations

Section 12.4 presents mechanisms to provide modelers with a quick and easy way (1) to erase model smells by automatically suggesting appropriate model refactorings, and (2) to get warnings in cases where new model smells occur due to applying a model refactoring. In order to propose suitable refactorings respectively to inform about potential new smells, the tooling must be provided with information on the relations between model smells and model refactorings.

A pragmatic way is to manually define these relations. Here, the advantage is that the designers can adjust the implementation of model smells and model refactorings to the fact that they are going to be related. A manually defined relation is done by a designer with the definitive goal to erase a model smell using a given model refactoring.

Since there are two possible relationships for model smells and model refactorings, EMF Refactor provides two extension points for the manual definition of these relations as presented in Section 11.2. On the one hand, a *smell ID* is related to a list of *refactoring IDs* (in case of providing suitable refactorings for a given smell). On the other hand, a *refactoring ID* is related to a list of *smell IDs* (in case of possible new smells when applying a given refactoring).

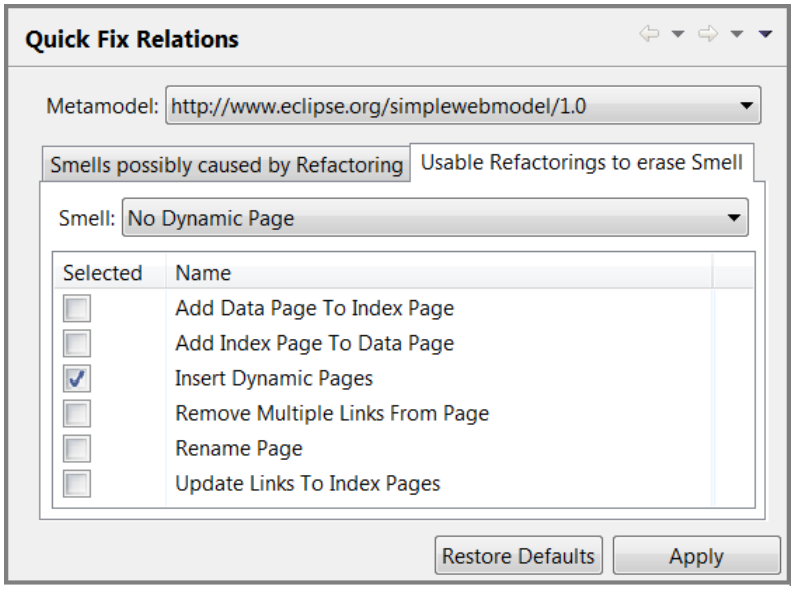


Figure 13.12: Manual configuration of refactorings being suitable to erase a given model smell

The definition of relations between model smells and model refactorings can be done in two ways: directly (by serving the corresponding extension point) or via a dedicated property page. This page provides graphical user interfaces for (de-)activating appropriate relations. Figure 13.12 shows the property page for (de-) activating relations between a given model smell and refactorings being suitable to eliminate this smell. Here, we address smell `No Dynamic Page` for SWM models (see meta model selection on the top). For this smell refactoring `Insert Dynamic Pages` is selected as a potential solution according to the discussions in the previous sections.

Since the application of a given refactoring poses a risk for inserting new model smell occurrences, EMF Refactor supports the manual configuration of this relationship between model refactorings and model smells.

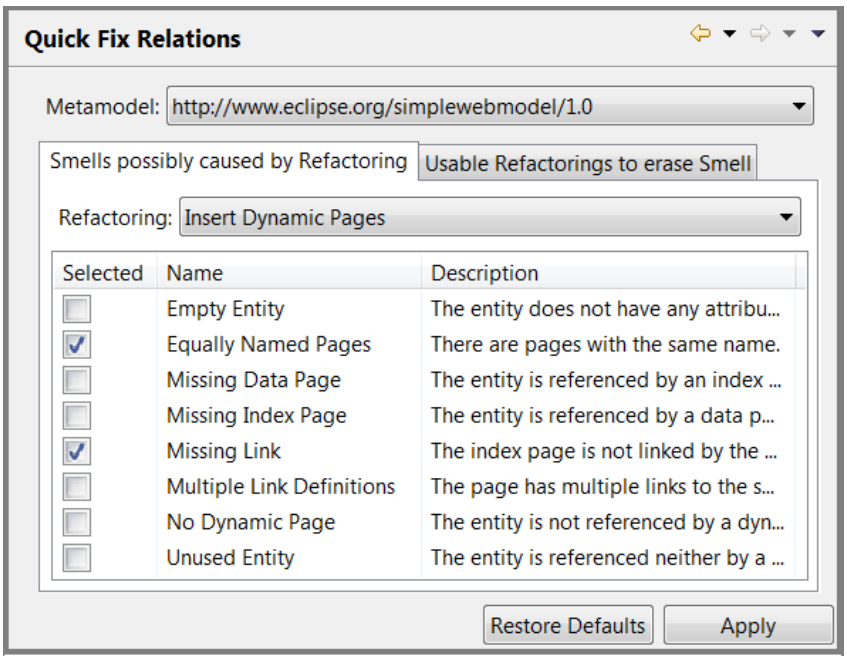


Figure 13.13: Manual configuration of potentially inserted smells after applying a given refactoring

Figure 13.13 shows an example for the selection of potentially new smells after applying a specific refactoring. Here, the afore mentioned SWM refactoring `Insert Dynamic Pages` is addressed and two smells are specified which can occur after applying the refactoring. On the one hand, smell `Equally Named Pages` occurs if the hypertext layer already contains a page with the same name as one of the derived names in the corresponding refactoring specification (see Figure 13.10 in the previous section). On the other hand, smell `Missing Link` occurs since the specification of the refactoring does not consider to add a link from the starting page to the newly created index page.

Please note that the relationships between model smells and refactorings need not be set for each project. Here, EMF Refactor uses

the Eclipse extension point technology to provide information about smell-refactoring relationships throughout the entire Eclipse system.

With this section, the description of the tool set EMF Refactor finishes. The next chapter of this thesis presents several evaluation tasks that have been performed in the context of this tool environment for EMF model quality assurance.

14

TOOL EVALUATION

In this Chapter, we evaluate the tool set EMF Refactor along two different perspectives. The chapter is structured as follows. First, Section 14.1 hypothesizes several claims to be addressed by the evaluation tasks described in Section 14.2. Then, we present the results of the evaluation in Section 14.3 followed by a discussion on threats to validity in Section 14.4.

14.1 GOALS AND HYPOTHESES

This section defines the goals and hypotheses we want to evaluate. In favor of a structured approach the following goals and hypotheses are distinguished along the perspectives *suitability* and *scalability*. The former subsumes aspects related to analysis capabilities like metrics calculation and model smell detection as well as capabilities concerning the active quality improvement of models in the sense of refactoring execution. The latter perspective especially tackles performance aspects like execution time.

The goals and hypotheses with respect to the suitability of the tool environment EMF Refactor are:

GOAL G_1: *The evaluation should show that the tools in EMF Refactor are suited to support the analysis and refactoring tasks of the presented model quality assurance process.* This includes that using the tools (1) prevents from incorrectly performing quality assurance tasks such as counting faults, and (2) provides results more quickly compared to manually performing the corresponding task.

GOAL G_2: *The specification technologies supported by EMF Refactor are suited to implement metrics, smells, and refactorings for any kind of EMF-based modeling language.*

HYPOTHESIS H_1: *EMF Refactor guides students being new in UML modeling along their way towards improved model quality by means of a sophisticated tool chain.*

HYPOTHESIS H_2: *The code generation facilities provided by EMF Refactor guide students on specifying new metrics, smells, and refactorings*

for EMF-based modeling languages and allows to concentrate on the essential specification part only.

The hypothesis concerning performance and scalability of the application modules in EMF Refactor is:

HYPOTHESIS H_3: *The application tools in EMF Refactor scale.*

Now that the goals and hypotheses are stated, the manner of how they are evaluated is described subsequently in turn followed by the actually evaluation.

14.2 EVALUATION TASKS

To evaluate the goals and hypotheses defined in the previous section we processed three different tasks. First, we provide a number of proof-of-concept implementations to evaluate G_1 and G_2. For evaluating H_1 and H_2, we performed two study experiments. Finally, we carried out several performance tests to evaluate H_3. This section describes these tasks in detail.

14.2.1 *Proof-of-concept implementations*

The suitability of EMF Refactor according to the quality assurance process defined in Part I of this thesis is evaluated by implementing a comprehensive catalog of model metrics, smells, and refactorings. These proof-of-concept implementations particularly target the EMF core meta model (Ecore), a commonly used meta model (UML2[1]), and a domain-specific meta model (SWM; see Sections 6.2 and 13.1).

	Model Metrics	Model Smells	Model Refactorings
Ecore	23	3	22
UML2	107	27	27
SWM	9	10	6

Table 14.1: Proof-of-concept implementations of metrics, smells, and refactorings for Ecore, UML2, and SWM models

Table 14.1 summarizes the implementations. The majority of implemented quality assurance techniques target UML2 models. For example, we provide implementations for measuring the structural quality conforming with well-known metrics like *Afferent* and *Efferent Coupling* [110, 84] or *DIT* and *MaxDIT* [24, 69]. Example UML2 smells and

1 We refer to UML2 being the standard EMF-based representation of UML2, i.e., `org.eclipse.emf.uml2.uml`.

refactorings are *Multiple Definitions of Classes with equal Names* [97], *Primitive Obsession* [11] and *Introduce Parameter Object* [64, 161]. [2]

We implemented the afore mentioned quality assurance techniques using different specification approaches. Table 14.2 summarizes the used approaches for concrete specifications of metrics, smells, and refactorings for UML2 models [3]. For comparison purposes, we implemented some refactorings using alternative approaches. Please note that each used approach has been selected freely by the designer, i.e., we did not evaluate the suitability of the supported approaches for each technique.

	UML2 Metrics	UML2 Smells	UML2 Refactorings
Java	13	9	24
OCL	47	1	–
Henshin	12	13	11
Comb.	35	–	–
Metric	–	4	–
CoMReL	–	–	16

Table 14.2: Used specification approaches for UML2 metrics, smells, and refactorings

Finally, we related 16 UML2 smells to 18 potentially suitable refactorings and 14 refactorings to 6 potentially occurring smells according to Tables 5.5 and 5.6 in Section 5.3.3.

14.2.2 *Study experiment I*

To evaluate goal G_1 and hypothesis H_1, we conducted the following experiment, referred to as `Ex_App` in the remainder of this chapter. The main subject of this experiment was the application of model quality assurance techniques to a given UML class model. In this experiment, we followed the four common ethical principles for research practices in empirical software engineering as discussed in [155] (informed consent, scientific value, confidentiality, and beneficence). The setting was as follows: [4]

Participants 20 undergraduate students participated in this study. The study was conducted as part of the course *Introduction to Software*

2 Lists of implemented metrics, smells, and refactorings for UML2 models can be found on the EMF Refactor web site [47].

3 Details on implemented smells and refactorings for UML2 class models can be found in Appendices D and F of this thesis.

4 The study material can be found in Appendix G of this thesis.

Engineering within the context of the B.Sc. study path held in winter term 2013/14 at Philipps-University Marburg, Germany.

Preliminary Studies In a 90 minutes theoretical lecture we introduced the participants to the topic of software quality and software quality assurance. In particular, we introduced to the quality assurance techniques metrics, smells, and refactoring. In a further 15 minutes introduction immediately before of the experiment we presented the adaptation of these techniques to the field of software modeling.

Study design We asked participants to perform model quality assurance techniques on the UML class model presented in Figure 12.1 on page 126. We considered class models for two reasons: first, because class diagrams are the mostly used UML diagram type [29], and second, since the students were especially introduced to UML class models in a preceding lecture a few weeks before the experiment took place. The experiment consisted of three main tasks with each being restricted to 20 minutes. In task `Ex_App_M`, the participants were asked to manually calculate a number of given metrics on the example class model. [5] In task `Ex_App_S`, the participants were asked to analyze the example class model with respect to 10 given model smell descriptions (3 metric-based and 7 pattern-based smells). In task `Ex_App_R`, the participants were asked to perform 5 refactorings and one manual change on the example class model in order to improve its structure. Finally, we randomly separated the participants into two groups. The groups were located in two different rooms to perform the tasks. The participants of group `M` (manual) were provided with an Eclipse IDE [50], Papyrus UML [59], and the example UML class model. The participants of group `T` (tooling) were additionally provided with EMF Refactor [47] and brief summaries how to use its functionalities.

Data collected We collected several measurements during the experiment. First, in a *pre-study questionnaire* we asked the participants about their personal skills, more precisely about their experience in software modeling, metrics calculation, model smell detection, and model refactoring. Here, we used a five-point Likert scale ranging from values `1` (beginner) to `5` (expert). Second, the participants returned the provided *forms* containing the results of the analyses tasks `Ex_App_M` and `Ex_App_S`. Third, the participants were asked to send the *Eclipse project* containing the refactored model after finishing task `Ex_App_R` to the experimenter. Fourth, in an *inter-study questionnaire* the students were asked to assess the difficulty of the tasks `Ex_App_M`,

5 In particular, 10 metrics to be calculated on the model level (sub task `Ex_App_M_1`), 10 metrics to be calculated on two different packages (sub task `Ex_App_M_2`), and 10 metrics to be calculated on three different classes (sub task `Ex_App_M_3`).

Ex_App_S, and Ex_App_R as well as to evaluate the provided time slot for performing the tasks. Here, we used a five-point Likert scale ranging from values 1 (very simple respectively much too short) to 5 (very difficult respectively much too long). Fifth, in a *post-study questionnaire* the participants of group M were asked how often during the experiment they thought that the modeling environment should provide functionality such as metrics calculation, smell detection, and refactoring. Here, we used a five-point Likert scale containing the values never, rarely, sometimes, often, and always. The participants of group T were asked how they appreciated the functionality of EMF Refactor. Here, we used a five-point Likert scale ranging from values 1 (not helpful at all) to 5 (very helpful). Finally, we asked all participants to give additional comments on the experiment.

14.2.3 *Study experiment II*

To evaluate hypothesis H_2, we conducted the following experiment, referred to as Ex_Spec in the remainder of this chapter. The main subject of this experiment was the specification of new model quality assurance techniques (model metrics, model smells, and model refactoring) for the DSML Simple Web Modeling Language (SWM) as presented in Sections 6.2 and 13.1. Also here, we followed the four common ethical principles for research practices in empirical software engineering as discussed in [155] (informed consent, scientific value, confidentiality, and beneficence). The setting was as follows: [6]

Participants 8 graduate students participated in this study. The study was conducted as part of the course *Model-driven Software Development* within the context of the M.Sc. study path held in winter term 2013/14 at Philipps-University Marburg, Germany.

Preliminary Studies In a 45 minutes theoretical lecture we introduced the participants to the topic of software model quality and model quality assurance. In particular, we introduced the quality assurance techniques model metrics, model smells, and model refactoring. In a further 45 minutes lecture we demonstrated the application of existing techniques as well as the specification of new techniques using the functionality of EMF Refactor. Finally, in a 15 minutes introduction immediately ahead of the experiment we presented the example language SWM.

Study design We asked participants to specify new model quality assurance techniques for textual SWM models using EMF Refactor. The grammar of SSM can be found in Listing 6.1 on page 68. The corresponding meta model is shown in Figure 13.1 on page 138. The

6 The study material can be found in Appendix H of this thesis.

experiment consisted of three main tasks. The time slots for processing the tasks were set to 40 minutes each. In task `Ex_Spec_M`, the participants were asked to specify 5 new SWM metrics of increasing complexity (sub tasks `Ex_Spec_M_1` to `Ex_Spec_M_5`) using either Java or OCL as concrete specification language. In task `Ex_Spec_S`, the participants were asked to specify 4 new SWM model smells of increasing complexity (sub tasks `Ex_Spec_S_1` to `Ex_Spec_S_4`) using Java as concrete specification language. In task `Ex_Spec_R`, the participants were asked to specify 3 new SWM model refactorings of increasing complexity (sub tasks `Ex_Spec_R_1` to `Ex_Spec_R_3`) using Java as concrete specification language. The participants were provided with an Eclipse IDE and a customized version of EMF Refactor including the SWM language. Finally, we provided a brief summary how to use the code generation functionality of EMF Refactor, three predefined plugin projects to be used as targets for the appropriate generated code, and an example SWM that can be used for testing purposes.

Data collected We collected several measurements during the experiment. First, in a *pre-study questionnaire* we asked the participants about their personal skills, more precisely about their proficiency with Java and OCL as well as their experience with EMF. Here, we used a ten-point Likert scale ranging from values `1` (beginner) to `10` (expert). Second, the participants were asked to send the provided *Eclipse projects* containing the specified techniques after finishing the tasks `Ex_Spec_M`, `Ex_Spec_S`, and `Ex_Spec_R` to the experimenter. Third, in an *inter-study questionnaire* the participants were asked to assess the difficulty of each sub task. Here, we used a ten-point Likert scale ranging from values `1` (very simple) to `10` (very difficult). Fifth, in a *post-study questionnaire* the participants were asked how much they appreciated (1) the possibility to use either Java or OCL for specifying new SWM metrics and (2) the code generation functionality of EMF Refactor. Here, we used a ten-point Likert scale ranging from values `1` (not helpful at all) to `10` (extremely helpful). Finally, we asked all participants to give additional comments on the experiment.

14.2.4 *Performance and scalability tests*

To evaluate the scalability of the application modules in EMF Refactor, we conducted several performance tests. We performed these tests on a Lenovo ThinkPad W500, Intel Centrino vPro 2.8GHz, 4MB RAM.

Metrics calculation For evaluating the scalability of the metrics calculation module we calculated a selected set of ten UML2 metrics on class models having 100, 500, 1.000, 5.000. 10.000, 50.000 and 100.000

elements and measured the duration of the calculation. Table 14.3 summarizes the selected metrics:

UML2 Metric	Description
`TNME`	Total number of elements in the model.
`MaxDIT`	Maximum of all depths of inheritance trees (context: model).
`MaxHAgg`	Maximum of aggregation trees (context: model).
`DNH`	Depth in the nesting hierarchy (context: package).
`NATIP`	Number of inherited attributes in classes within the package.
`NOPIP`	Number of inherited operations in classes within the package.
`HAgg`	Length of the longest path to the leaves in the aggregation hierarchy (context: class).
`MaxDITC`	Depth of Inheritance Tree (maximum due to multiple inheritance; context: class).
`NSUBC2`	Number of all children of the class.
`NSUPC2`	Total number of ancestors of the class.

Table 14.3: UML2 metrics used for performance and scalability testing

We selected these metrics to cover inheritance and nesting issues. Note that they are calculated on different context types (model, package, and class). We consider UML2 models only due to the variety of implemented metrics (see Table 14.2 on pagee 157).

Model instances are created using a basic model similar to the example class model shown in Figure 6.1 on page 56. To provide an adequate model instance of required size we duplicated respectively nested the root package. Doing this, we assure that the number of calculated metrics grows nearly linearly compared to the model size. For each model size, we repeated the metrics calculation ten times and reported the average time needed for metrics calculation.

Smell detection For evaluating the scalability of the smell detection module we analyzed UML2 class models with 100, 500, 1.000, 5.000. 10.000, 50.000 and 100.000 elements with respect to a set of seven selected smells for UML2 models and measured the time needed for smell detection. The selected smells are:

1. `Concrete Superclass` - The model contains an abstract class with a concrete superclass.

2. `Equal Attributes in Sibling Classes` - Each sibling class of a common parent contains an equal attribute.
3. `Specialization Aggregation` - The model contains a generalization hierarchy between associations.
4. `Speculative Generality (Abstract Class)` - The model contains an abstract class that is inherited by one single class only.
5. `Speculative Generality (Interface)` - The model contains an interface that is implemented by one single class only.
6. `Unused Class` - The model contains a class that has no child or parent classes, that is not associated to any other classes, and that is not used as attribute or parameter type.
7. `Unused Interface` - The model contains an interface that is not specialized by another interface, and not realized or used by any classes.

We selected the model smells with respect to their influence on quality aspect *confinement* (see Section 4.2). Smells 1 to 3 use consistent language concepts being more complex than necessary. Smells 4 and 6 can also be found in the example case presented in Section 6.1. Finally, smells 5 and 7 are similar to smells 4 an 6 but consider interfaces instead of classes. Again, we consider UML2 model smells only due to the same reasons mentioned above.

The model instances are constructed in the same way as in the metrics calculation case. For each model size, we repeated the smell detection ten times and reported the average time needed for smell detection.

Refactoring execution For evaluating the scalability of the refactoring execution module we applied 7 pretty complex UML2 refactorings on models with a larger refactoring context instead of large-scale models. Table 14.4 summarizes these refactorings and describes the context each refactoring has been applied on.

We measured the time in-between committing the refactoring (i.e., after parameter input) and finishing the corresponding model change. Moreover, we repeated each refactoring application ten times to address potential side effects and reported the average time needed for refactoring execution.

14.3 EVALUATION RESULTS

This section summarizes the results and observations made during the evaluation tasks described in the previous section. The following discussions are guided by the order of goals and hypotheses defined in Section 14.1 and refer to the corresponding task if appropriate.

Refactoring	Context
Extract Class	Refactoring application on a class having 10 attributes and 10 operations.
Extract Subclass	Refactoring application on a class having 10 attributes and 10 operations. The selected class has 10 child classes already. Each child class has 10 attributes and 10 operations.
Extract Superclass	Refactoring application on 10 classes having 10 equal attributes and 10 equal operations.
Inline Class	Refactoring application on a class having 10 attributes and 10 operations.
Introduce Parameter Object	Refactoring application on 9 parameters of an operation with 10 input parameters. The owning class has altogether 10 operations with 10 parameters each. Each operation has parameters equal to the selected ones.
Merge States	Refactoring application on a state with 5 incoming transitions. The parameter state has entry, doAction, and exit behaviour. The prameter state has 5 incoming transitions equal to the selected state. The owning region has 20 further states.
Remove Superclass	Refactoring application on a class having 10 attributes and 10 operations. The selected class has 10 child classes already. Each child class has 10 attributes and 10 operations.

Table 14.4: UML2 refactorings used for performance and scalability testing

14.3.1 *Suitability*

The goals and hypotheses defined with respect to the suitability of EMF Refactor are evaluated as follows.

G_1 – *General suitability, correctness, and efficiency of application modules*

The functionalities described in Chapters 12 and 13 as well as the entries in Table 11.2 on page 124 show that EMF Refactor fulfills the requirements defined in Chapter 9 and Section 11.1. All basic functionalities (metrics calculation, smell detection, and refactoring application) are provided as well as further functionalities such as configurability and metrics reporting. Furthermore, the proof-of-concept implementations summarized in Section 14.2.1 show that EMF Refac-

tor supports metrics calculation, smell detection, and refactoring of EMF-based models to a high extent.

To analyze whether using the tools prevent from incorrectly performing quality assurance tasks we partially use the results of experiment `Ex_App`. We start with a summary on the personal skills of the study participants.

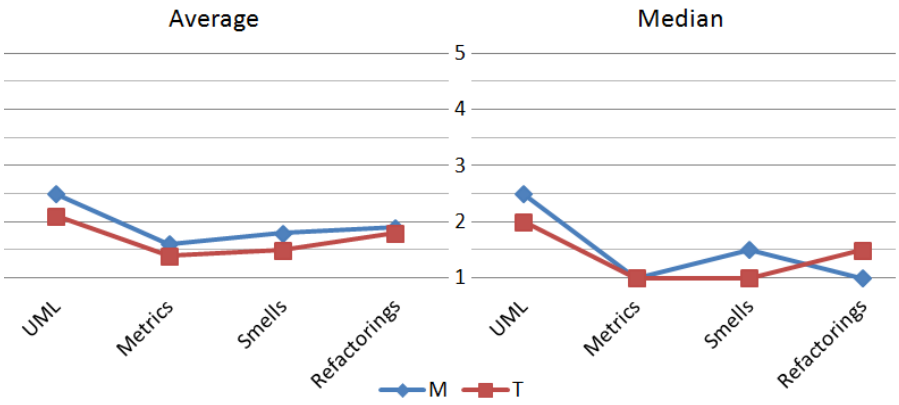


Figure 14.1: Personal skills of the participants in experiment `Ex_App`

Figure 14.1 shows the results of the *pre-study questionnaire* of experiment `Ex_App` for both groups `M` and `T`. The left diagram illustrates the average score of given estimations (ranging from `1` (beginner) to `5` (expert)) in the context of the participant's experience in software modeling, metrics calculation, model smell detection, and model refactoring. The diagram on the right-hand-side illustrates the corresponding median score. Both diagrams show that the participants of experiment `Ex_App` are obviously inexperienced in UML modeling (group `M`: average and median 2.5 each; group `T`: average 2.1 and median 2). Especially, they are inexperienced in applying model quality assurance techniques. Here, both average and median scores are less than 2. Moreover, the participants of group `T` seem to be slightly less experienced than those of group `M`. This means that the results of the following analysis of the returned *forms* and *Eclipse projects* are not influenced by different personal skills of both groups.

Figure 14.2 shows the percentages of correct results of the performed tasks `Ex_App_M`, `Ex_App_S`, and `Ex_App_R`. Task `Ex_App_S` is subdivided into two parts: one part addresses the correctness of the numbers of detected smells for each smell type, the other part addresses the correctness with respect to the number of model smell occurrences. Furthermore, we illustrate four kinds of correctness. For each group `M` and `T` we provide the correctness of the returned results concerning both the number of the submitted results and the maximal number of results.

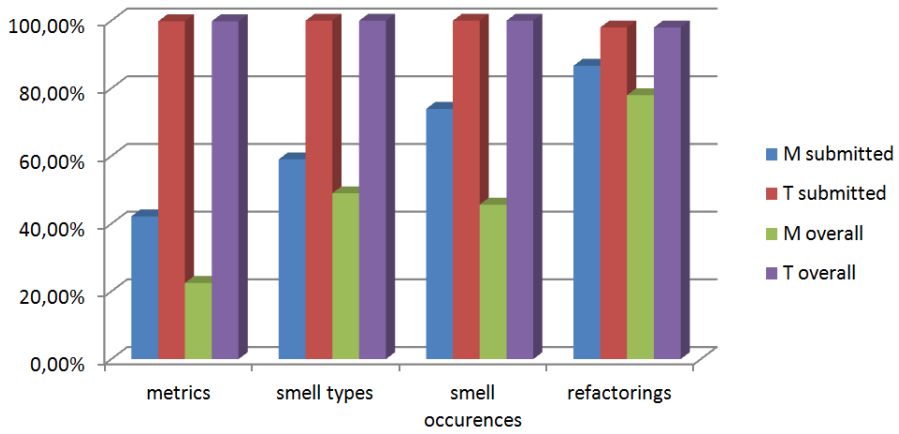


Figure 14.2: Percentages of correct results concerning experiment Ex_App

The results provided by group T are highly correct. In fact, 599 out of 600 submitted (and also maximal possible) metrics and all submitted model smells are correct as well as 49 out of 50 performed refactorings. For group M, only 135 out of the 320 submitted results are correct (42.2% respectively 22.5% according to the maximal number of results). For smell detection and refactoring, these values are slightly better nevertheless far from perfect. In summary, the results show that the use of EMF Refactor (group T) prevent from incorrectly performing quality assurance tasks as partially done by group M.

To analyze whether the application tools in EMF Refactor provide results more quickly compared to manually performing the corresponding task we also partially use the results of experiment Ex_App. Again, we start with an analysis of the returned *forms* and *Eclipse projects*.

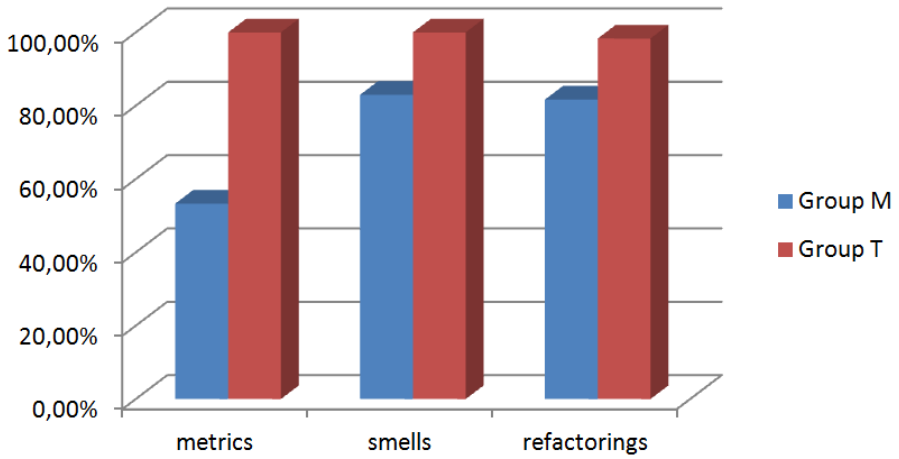


Figure 14.3: Percentages of performed tasks during experiment Ex_App

Figure 14.3 shows the percentages of the performed tasks Ex_App_M, Ex_App_S, and Ex_App_R. From altogether 600 metrics that should be maximally calculated the participants of group M calculated 320 (53.3%) whereas those of group T calculated each metric. Furthermore, in task Ex_App_S the participants of group M searched for 83 out of 100 possible smell types (83%). Again, the participants of group T searched for each smell type. Finally, the participants of group M performed 49 out of 60 demanded model changes (83.7%). Here, the participants of group T performed 59 out of 60 changes (98.3%).

The results show that during the 20 minutes time slot the participants of group T (which are even less experienced with the topic of the experiment according to the *pre-study questionnaire*, see discussion above) calculated above 87% more metrics than those of group M. Here, the speed-ups for smell detection and refactoring execution are approximately 20% each. However, these values might be even more distinctive if more time would be provided as the results of the *inter-study questionnaire* of experiment Ex_App show. Here, 5 out of 8 data items (62.5%) provided by the participants of group M claimed that the provided 20 minutes time slots for each task were *too short* respectively *much too short*. On the other hand, 6 out of 10 data items (60%) provided by the participants of group T claimed that these slots were *too long* respectively *much too long*.

G_2 – *Suitability of supported specification technologies*

The proof-of-concept implementations summarized in Section 14.2.1 show that each supported specification approach is suited to specify metrics, smells, and refactorings which can then be used by the corresponding application modules. Our experiences in using the various specification approaches show that using Java has been the favorite approach for implementing specifications, especially for implementing refactoring specifications. In fact, this may be due to the preferences of the designer and the progress of supported approaches by the corresponding tool. Independent of the preferred specification language, we feel confident that OCL is particularly suited for specifying metrics which can be directly deduced from the contextual model element using adequate meta attributes respectively references. Henshin transformations have been proven well-suited especially for specifying the model change part of a refactoring.

H_1 – *Guided model quality assurance*

To evaluate whether EMF Refactor guides inexperienced students along their way towards improved model quality we consider the results of experiment Ex_App. More concretely, we relate the number

of (correct) applied quality assurance techniques to the personal skills and the noticed severity of the performed tasks.

As shown in the diagram in Figure 14.1 on page 164, the participants of experiment Ex_App are obviously inexperienced in UML modeling, especially in applying model quality assurance techniques (see discussion on goal G_1).

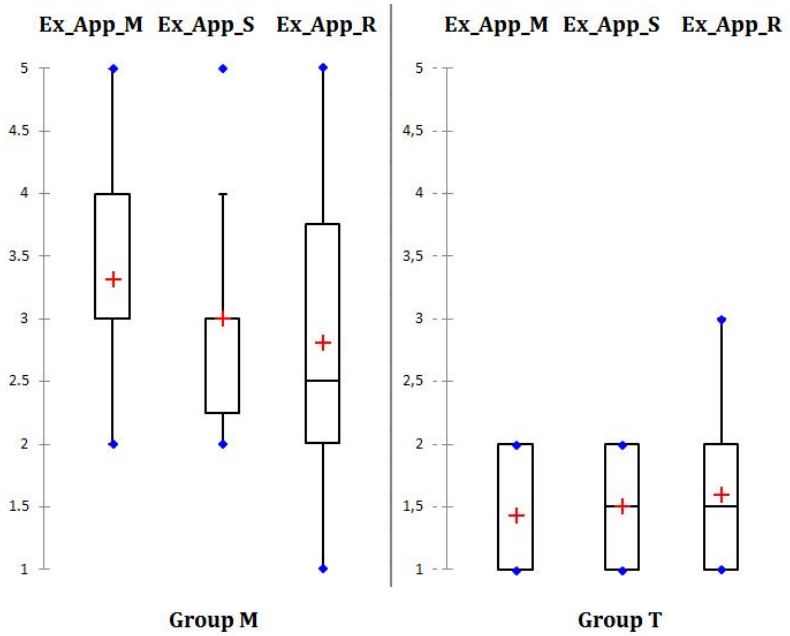


Figure 14.4: Difficulty scores for the tasks in experiment Ex_App

Despite of this inexperiences the entries in the *inter-study questionnaire* showed that the difficulties of the tasks in this experiment are at least rated as *simple* in general (see box plots in Figure 14.4). The average score for group T on the five-point Likert scale ranging from values 1 (very simple) to 5 (very difficult) is 1.48 and the median is 1. Here, the analysis tasks are scored as being slightly easier to perform (average 1.43 and median 1 for task Ex_App_M; average 1.5 and median 1.5 for task Ex_App_S) in relation to task Ex_App_R(average 1.6 and median 1.5). However, the scores of the manual group M are significantly higher than those of group T. Here, the average score is 3.13 and the median is 2.5.

These ratings are reflected in the submitted results of the tasks Ex_App_M, Ex_App_S, and Ex_App_R (*forms* and *Eclipse projects*). We analyze the results in detail in the discussions on goal G_1 with respect to correctness and efficiency. From this discussions, we can summarize that the participants of group T

- correctly calculated all metrics demanded in task Ex_App_M,
- correctly detected all smells demanded in task Ex_App_S, and

- successfully applied 98% of the refactorings demanded in task `Ex_App_R`.

Finally, in the *inter-study questionnaire* nearly all ratings (29 out of 30) returned by the participants of group `T` claimed that the functionality of EMF Refactor is either *helpful* or *very helpful* to perform the tasks of this experiment. Moreover, more than half of the ratings (16 out of 30) returned by the participants of group `M` claimed that during the experiment they *often* respectively *always* thought that the modeling environment should provide such a functionality (metrics calculation, smell detection, and refactoring).

In summary, the results presented above show that even inexperienced modelers are able to apply a huge number of model quality assurance techniques within a relatively short period of time (60 metric calculations, 23 smell detections, and 5 refactorings within 60 minutes) when using EMF Refactor.

H_2 – *Guided specification process and usefulness of code generation*

To evaluate whether the code generation facilities provided by EMF Refactor guide students on specifying new metrics, smells, and refactorings for EMF-based modeling languages and allows to concentrate on the essential specification part only we consider the results of experiment `Ex_Spec`. We start with relating the number of (correct) implemented quality assurance techniques to the personal skills and the noticed difficulties of the performed tasks.

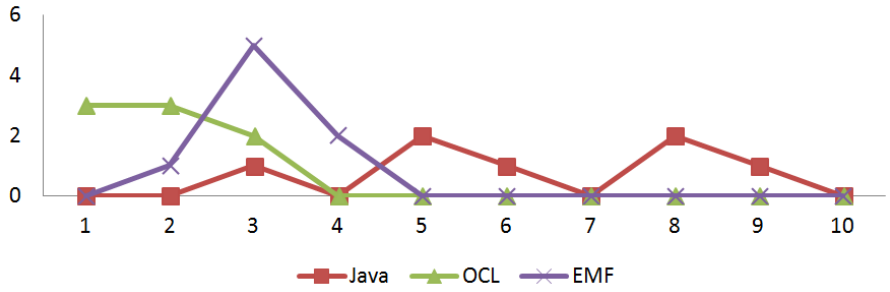


Figure 14.5: Personal skills of the participants in experiment `Ex_Spec`

Figure 14.5 shows the results of the *pre-study questionnaire* of experiment `Ex_Spec`. The diagram illustrates the number of given estimations per score (ranging from `1` (beginner) to `10` (expert)) in the context of the proficiency with Java and OCL and the experience with EMF. It shows that the proficiency with Java is pretty diverse among the participants of the study (average 6.28; median 6). However, the proficiency with OCL is very low (1.88 on average; median 2) and the participants have little to moderate experience with EMF only (average score 3.13; median 3).

Despite of this rather low proficiencies respectively inexperiences the entries in the *inter-study questionnaire* showed that the difficulties of the tasks in this experiment are rated as *moderate* in common. The average score on the ten-point Likert scale ranging from values `1` (very simple) to `10` (very difficult) is 4.5 and the median is 4. Here, the analysis tasks are scored as being easier (average 3.25 and median 3.5 for task `Ex_Spec_M`; average 3.38 and median 2 for task `Ex_Spec_S`;) whereas `Ex_Spec_R` is scored as being the most difficult one (average 5.63 and median 6) as expected due to the complexity of refactoring specifications as repeatedly mentioned throughout this thesis.

The returned *Eclipse projects* containing the specified techniques after finishing tasks `Ex_Spec_M`, `Ex_Spec_S`, and `Ex_Spec_R` lead to the following results:

- During the 40 minutes time slot of task `Ex_Spec_M` the participants implemented 3.6 metrics on average (11.2 minutes per metric on average).
- In 76% of the implemented metrics the participants used Java as specification language, in 12% they used OCL. The possibility to combine two metrics to more complex ones was used by every second participant.
- Just over half (52%) of the implemented metrics were specified correctly. Most of the erroneous implementations are caused by conceptional misunderstandings (8 cases) respectively incorrect usage of Java and OCL (3 cases). However, in 4 cases the code generation module lead to lost code due to overwriting files without warning.
- During the 40 minutes time slot of task `Ex_Spec_S` the participants implemented 3.6 model smells on average (less than 11 minutes per model smell on average).
- 72.4% of the implemented model smells were specified correctly. The erroneous implementations are caused either by conceptional misunderstandings (5 cases) or incorrect usage of Java (4 cases).
- During the 40 minutes time slot of task `Ex_Spec_R` the participants implemented 2.3 model refactorings on average (17.5 minutes per model refactoring on average).
- 62.5% of the implemented model refactorings were specified correctly. The erroneous implementations are caused either by conceptional misunderstandings (5 cases) or incorrect usage of Java (3 cases).

In summary, the results presented above show that even inexperienced designers are able to specify a number of new model metrics, smells, and refactorings within a relatively short period of time

(nearly 10 techniques within 120 minutes).[7]

In the *post-study questionnaire* of experiment `Ex_Spec` we asked the participants how much they appreciated (1) the possibility to use either Java or OCL for specifying new SWM metrics and (2) the code generation functionality of EMF Refactor.

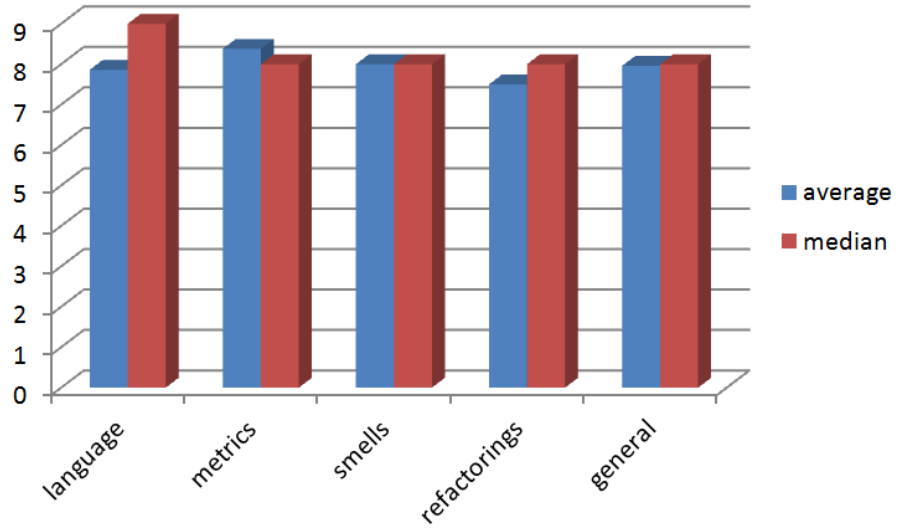


Figure 14.6: Evaluation of the helpfulness of the specification components of EMF Refactor

Figure 14.6 summarizes the evaluation results of these questions. It shows that the opportunity to choose between different languages is seen as *(very) helpful* (average score 7.86; median 9; see left pair of bars in Figure 14.6). Similar ratings are given to the code generation facilities of EMF Refactor. Here, the median is 8 for each component. The overall average value for the code generation components is 7.96 (see right pair of bars). The best average value is given to the metrics specification component (8.38) whereas those for the smells respectively refactoring specification components are scored slightly lower (8 and 7.5, respectively). Here, those participants who awarded moderate scores only (ranging from 4 to 6) criticized two suboptimal structures and gave valuable proposals to improve the generated code in order to make it even more reasonable for the designer.

In the *proof-of-concept implementations*, Java specifications of UML2 metrics use 15.2 LoC (Lines of Code) on average (min. 1 LoC; max. 36 LoC), whereas UML2 smells are implemented in 20.5 LoC on average (min. 13 LoC; max. 74 LoC). Refactoring specifications require 99.7 LoC on average (min. 8 LoC; max. 269 LoC). Here, about 20% (20.2

7 One comment in the *post-study questionnaire* stated:'Even without great experiences time saving was obvious.'

LoC on average) are used for specifying the model change part only, but almost 80% (79.5 LoC on average) for specifying the initial and final precondition checks. This shows that the complexity of refactoring specifications is particularly hidden in checking the corresponding preconditions (compare footnote in Section 13.3). In summary, specifications are compact and concentrate purely on the quality assurance technique to be specified.

14.3.2 *Performance and scalability*

The hypothesis concerning performance and scalability of the application modules in EMF Refactor is evaluated as follows.

H_3 – *Scalability of application modules*

Table 14.5 shows the results of the performance tests for (1) calculating 10 UML2 metrics and (2) the detection of 7 UML2 model smells on class models with 100 to 100 000 elements as described in Section 14.2.4.

# model elements	# metric instances	average time	# smell occurrences	average time
100	42	.365 sec	12	.475 sec
500	201	.563 sec	60	.550 sec
1 000	399	1.472 sec	120	.607 sec
5 000	2 008	8.494 sec	600	2.834 sec
10 000	4 016	37.705 sec	1 200	10.716 sec
50 000	20 068	8 m 36 sec	6 000	5 m 05 sec
100 000	40 136	33 m 54 sec	12 000	20 m 50 sec

Table 14.5: Results of the performance tests for metrics calculation and smell detection

The results show that the application modules for metrics calculation and smell detection are well-suited for small and mid-sized EMF-based models. For large-scale models (having more than 10.000 elements), reporting of a high number of calculated metrics (respectively detected smells) is provided in a satisfying time only. However, since static analyses normally do not need to be performed time-critically, this is no crucial limitation of our tool set. Furthermore, the configuration mechanism of our tools can be even used to deal with large-scale models efficiently. For example, the configuration of only a small number of relevant metrics and smells reduces the overall execution time. Moreover, a smell search can be performed on a subtree of the

model only, again reducing the overall execution time. However, potential inefficiencies need to be analyzed and performance-oriented technologies for metric computation and smell detection need to be discussed, e.g., performing metrics calculations in parallel.

Refactoring	min. time	max. time	aver. time
Extract Class	43 ms	110 ms	66 ms
Extract Subclass	178 ms	236 ms	196 ms
Extract Superclass	91 ms	119 ms	105 ms
Inline Class	17 ms	47 ms	34 ms
Introduce Parameter Object	85 ms	101 ms	93 ms
Merge States	78 ms	107 ms	88 ms
Remove Superclass	143 ms	231 ms	182 ms

Table 14.6: Results of the performance tests for refactoring application

Table 14.6 summarizes the results of the performance tests for the application of 7 UML2 model refactorings on model instances having a larger refactoring context as described in Section 14.2.4. The maximum time needed to apply a refactoring (without parameter input) has been 236 ms. So, the results show that the refactoring execution module is well-suited for applying refactorings even on large-scale refactoring contexts.

14.3.3 *Summary*

For summarizing the discussion of the results presented in the previous section, we claim the following statements along the goals and hypotheses defined in Section 14.1:

GOAL G_1: *The evaluation should show that the tools in EMF Refactor are suited to support the analysis and refactoring tasks of the presented model quality assurance process.* This includes that using the tools (1) prevents from incorrectly performing quality assurance tasks such as counting faults, and (2) provides results more quickly compared to manually performing the corresponding task. ✓

$\Longrightarrow$ This goal is proven by the presented architecture and functionality of EMF Refactor, the proof-of-concept implementations, and the results of experiment `Ex_App`.

GOAL G_2: *The specification technologies supported by EMF Refactor are suited to implement metrics, smells, and refactorings for any kind of EMF-based modeling language.* ✓

$\Longrightarrow$ This goal is proven by the presented proof-of-concept implementations.

HYPOTHESIS H_1: *EMF Refactor guides students being new in UML modeling along their way towards improved model quality by means of a sophisticated tool chain.* (✓)

$\Longrightarrow$ This hypothesis seems to be proven by the results of experiment `Ex_App` (see discussion on threats to validity in the subsequent section).

HYPOTHESIS H_2: *The code generation facilities provided by EMF Refactor guide students on specifying new metrics, smells, and refactorings for EMF-based modeling languages and allows to concentrate on the essential specification part only.* (✓)

$\Longrightarrow$ This hypothesis seems to be proven by the proof-of-concept implementations and the results of experiment `Ex_Spec` (see discussion on threats to validity in the subsequent section).

HYPOTHESIS H_3: *The application tools in EMF Refactor scale.* ✓ / $\lightning$

$\Longrightarrow$ This hypothesis is only partially proven by the results of the performance tests for the refactoring tools in EMF Refactor. For the analysis tools (metrics calculation and smell detection) this hypothesis is rejected by the results of the performance tests. However, we can split and lessen the hypothesis to H_3^*: *The refactoring tools in EMF Refactor scale whereas the analysis tools in EMF Refactor (metrics calculation and smell detection) scale for small and mid-sized models.* Then, H_8^* is proven by the results of the performance tests.

14.4 THREATS TO VALIDITY

In this section, we discuss several threats to validity concerning the evaluation tasks presented in Section 14.2 that should be considered when interpreting the corresponding results.

14.4.1 *Proof-of-concept implementations*

The first threat concerning our proof-of-concept implementations is the *selection of modeling languages* for those we implemented the quality assurance techniques. Here, we tried to select representatives of three kinds of modeling languages. We chose Ecore as MOF-based meta modeling language, UML2 as commonly used modeling language, and SWM as representative of a domain-specific modeling language that is frequently used in literature, for example in [21]. We are convinced that this selection is appropriate.

Another threat is the *selection of implemented techniques* for these languages. For UML, we implemented the vast majority of techniques

that we found in literature as summarized and discussed in Chapter 5 of this thesis. Some of these techniques we adapted to Ecore since UML class diagrams are closely related to Ecore models. However, the quality of MOF-based meta models may depend on different objectives than the quality of design models formulated in UML. For example, all editors and every transformation to other artifacts depend on the meta model, and therefore on its quality. To develop quality assurance techniques specifically tailored to meta modeling and to implement them for Ecore is an open research topic and a promising task for future work. Finally, we implemented the quality assurance techniques for SWM according to the example case discussed in Section 6.2 of this thesis. Again, we are convinced that this selection is appropriate.

A further threat is the *selection of the appropriate specification language* for these implementations. As we stated in Section 14.2.1, each used approach has been selected freely by the designer. We selected different specification techniques only for demonstrating the usefulness of the approaches and the flexibility of the tool set. Each quality assurance technique could be also have been implemented in Java due to its powerfulness. So in summary, the choices do not reflect suitability and efficiency issues. However, comparison studies concerning these issues may be directions for future work.

14.4.2 *Study experiment I*

Concerning the study design of experiment `Exp_App` one threat to validity is the *choice and number of the participants*. Since we demanded inexperienced modelers we chose the students of a beginner course to be suited as participants for the experiment. Furthermore, the students were introduced to UML (especially to UML class models) in a preceding lecture a few weeks ahead the experiment. The relatively small number of participants (20) does not allow for inferential and inferential statistics. However, the results presented in the previous section seem to be conclusive and we do not think that a similar experiment with a considerably higher number of participants would lead to considerable different results.

Another significant threat is the *potential communication between participants*. This might be crucial since we performed the experiment as part of a B.Sc. course having a grading system. To respond to this threat we anonymized the experiment and multiply mentioned that the results of the tutorial do not influence the final grade. Instead, we motivated the participants to be cooperative. Finally, we scattered the used computers over the room to have as many distance between the students as possible.

A further threat is the *selection of the modeling language and the model* for performing the tasks of experiment `Exp_App`. As already men-

tioned above we introduced the UML language, especially the use of UML class diagrams, in a preceding lecture a few weeks ahead the experiment. So, the participants were sufficiently familiar with the used modeling language though not experienced (as the results of the *pre-study questionnaire* show). Although the used model is constructed it is of suitable size. Furthermore, the model was clear and comprehensible for the students as clarified in the preliminary studies of the experiment.

Finally, the *selection of the quality assurance techniques* to be used represents a further threat to validity. We designed the experiment to address quality assurance techniques of different complexity. For metrics calculation, we selected three different contexts. For smell detection, we prepared a mixture of metric-based and pattern-based smells. Furthermore, we used simple (e.g., *Rename Class*) and more complex refactorings (e.g., *Remove Superclass*). Finally, we listed the techniques according to their increasing complexity and multiply advised not to perform each task in the order they listed.

14.4.3 *Study experiment II*

To interpret the results of experiment `Exp_Spec` several threats have to be considered that are similar to those for experiment `Exp_App`.

Again, the first threat to validity is the *choice and number of the participants*. The design of experiment `Exp_Spec` demanded for participants having moderate to good skills in Java and UML modeling. So, we decided to recruit graduate students of a M.Sc. course being suited as participants of the experiment. Furthermore, the students were introduced to OCL and EMF in preceding lectures a few weeks ahead the experiment. With respect to the small number of participants (7) we ague in the same way as above.

Like in `Exp_App`, the *potential communication between participants* is also a threat of this experiment. Here, we respond to this threat in the same way as above.

Another threat is the *selection of the modeling language* for specifying new model quality assurance techniques. We used the SWM language for two reasons. First, the students already used a variant of SWM in preceding lectures a few weeks ahead the experiment. Second, we think that the size of the SWM meta model is appropriate for the given tasks and its concepts are clear and highly comprehensible.

Finally, the *selection of the quality assurance techniques* to be specified represents a further threat to validity. Here, we tried to select techniques which address different language features like relations between the hypertext and the data layer or relations only within the data layer. Furthermore, the techniques were completely new to the students and we implemented them a couple of weeks ahead the experiment. So, we were familiar with potential implementation

problems which helped when performing the experiment. Finally, we listed the techniques according to their increasing complexity and multiply advised not to perform each task in the order they listed.

14.4.4 *Performance and scalability tests*

The first threat to validity to be discussed concerning the settings of the performance and scalability tests is the *choice of the model quality assurance techniques* that are performed during the tests. At first, we state that we chose UML because of the large number of implemented techniques. We selected the metrics in order to cover two language features, i.e., inheritance and nesting, and three different contexts. We calculated a few local metrics (for example, metric `NSUPC2` calculates the total number of ancestors of a class) as well as many global ones (for example, metric `MaxDIT`). Therefore, the selection of the metrics to be calculated influenced the execution time rather negatively than positively. The same arguments hold for the selection of the smells to be detected. Finally, we selected complex refactorings only resulting in more elaborated precondition checks and model change parts. Also here, the execution time is influenced rather negatively.

Another threat is the *creation methodology* of the example models. Here, we prepared a fictive UML class model used in Section 6.1 and duplicated respectively nested the root package in order to provide numbers of calculated metrics and smell occurrences that grow nearly linearly with respect to the model size. We concede that using real world models would strengthen the confidence in using the tools. However, we are convinced that for the purpose of performance testing the prepared models are appropriate.

Finally, a threat related to the previous one is the *preparation of the refactoring context*. Here, we are convinced that the prepared contexts do not occur in real world models in such extents. So also here, the execution time is influenced rather negatively.

15

CONCLUSION AND FUTURE WORK

This part of the thesis presented a tool environment for model quality assurance based on the Eclipse Modeling Framework (EMF), a common open source technology in model-based software development. It has been designed to support a syntax-oriented model quality assurance process that can be easily adapted to specific needs in model-based projects as discussed in the first part of this thesis. The entire tool set presented belongs to the Eclipse incubation project *EMF Refactor* [47] and is available under the Eclipse public license.

The EMF Refactor framework supports both the model designer and the model reviewer by obtaining metrics reports, by checking for potential model deficiencies (called model smells) and by systematically restructuring models using refactorings. Automatically proposed refactorings as quick fixes for occurring smells and information on implications of a selected refactoring concerning new model smells widen the provided functionality and support an integrated use of the quality assurance tools.

The sets of actually supported techniques can be configured separately for each project. Here, the configuration of model smells in combination with the specification of smell-refactoring relations might influence the section of model refactorings (and vice versa). Future releases of EMF Refactor should address these dependencies, e.g., by using a fix point analysis.

The main functionality of EMF Refactor is integrated into several editors. Here, not only standard tree-based EMF instance editors are supported but also graphical GMF-based editors as used by Papyrus UML and textual editors provided by Xtext. Moreover, we integrated our tool environment into the widely used EMF-based UML CASE tool IBM Rational Software Architect. Among other functionalities, each version provides a highlighting of model elements for smells in the corresponding model view and a preview of upcoming model changes when performing a refactoring using a tree-based visualization provided by EMF Compare. It is up to future work, to present the preview of refactoring effects also graphically respectively textually.

Model checks and refactorings can be specified by several specification mechanisms. The current version of EMF Refactor supports Java, OCL, and the model transformation language Henshin as possi-

ble specification approaches. However, further query languages like EMF Query and EVL as well as model transformation approaches such as EWL and ATL (in-place) are interesting alternatives to be used in future releases.

In EMF Refactor, metrics can be composed to more complex metrics and refactorings can be composed by using a dedicated language named CoMReL. It is up to future work to analyze the preconditions of component refactorings with respect to their execution order and to deduce a composite precondition therefrom. Here, we are currently working on a first approach for in-depth composition of refactorings for Henshin-specified ones using algebraic graph transformations and critical pair analysis.

In future releases, we will continue with making the quality assurance tools still more user-friendly. Besides support for further available quality assurance techniques and further specification languages, performance and scalability shall be further optimized. Here, potential inefficiencies need to be analyzed and performance-oriented technologies for metric computation and smell detection need to be discussed, e.g., performing metrics calculations in parallel. Another open issue is how to deal with false positives during model smell detection. These are concrete smell occurrences being actually non-issues which should be ignored. Here, we think of using mechanisms like `@SupressWarnings` in Java to indicate areas to be elided during a new smell search. In the context of EMF, `EAnnotations` might be useful.

Finally, we can think of providing tool support for further (constructive) model quality assurance techniques such as complex editing operations whose specification seems to be similar to those for model refactorings. Here, a potential example is to insert design patterns into the model. A combination with the already existing functionality would lead to an interesting feature: refactoring to patterns.

In summary, since EMF Refactor is based on EMF, quality assurance for a variety of modeling languages is supported (in contrast to UML CASE tools like RSA). It is the first tool providing metrics calculation for EMF-based models and is fully integrated into the modeling IDE (in contrast to SDMetrics). After detecting model smells, EMF Refactor provides a quick-fix mechanism by suggesting refactoring operations (in contrast to EVL and EMF Validation). Finally, EMF Refactor provides a homogeneous refactoring workflow in Eclipse by using LTK (in contrast to EWL). Therefore, we are convinced that the current version of EMF Refactor presented in this part contributes to make model-based and model-driven development more mature yielding software of higher quality.

16

THESIS CONCLUSION

Chapter 16 initially provides a summary of this thesis and concludes with an outlook on the current and future work

16.1 SUMMARY

The paradigm of model-based software development (MBSD) has become more and more popular since it promises an increase in the efficiency and quality of software development. In this paradigm, models play an increasingly important role and become primary artifacts in the software development process. As a consequence, software quality and quality assurance frequently leads back to the quality and quality assurance of the involved models.

Well-known quality assurance techniques for models are model metrics and model refactorings. They origin from corresponding techniques for software code by lifting them to models. Furthermore, the concept of code smells can be lifted to models leading to model smells. However, these techniques are mostly considered in isolation only.

To meet these problems, this thesis presents an integrated approach to perform model quality assurance systematically. In particular, we define a structured process for quality assurance of software models that can be adapted to project-specific and domain-specific needs. The process is structured into two sub-processes: a process for the specification of project-specific model quality assurance techniques, and a process for applying them on concrete software models within ongoing MBSD projects. The approach concentrates on quality aspects to be checked on the model syntax and is based on model analysis techniques, i.e., on reports on model metrics and on checks against the existence (respectively absence) of model smells. Finally, model refactoring is the technique of choice to eliminate a recognized model smell. Three example cases performing this model quality assurance process serve as proof-of-concept implementations and show its applicability and flexibility, and hence the usefulness of the approach.

Further (minor) conceptual contributions of this thesis are (1) the definition of a quality model for model quality that consists of high-level quality attributes and low-level characteristics, (2) overviews on metrics, smells, and refactorings for UML models discussed in litera-

ture including structured descriptions of each technique, and (3) an approach for composite model refactoring that concentrates on the specification of refactoring composition.

Since manually reviewing models is time consuming and error prone, several tasks of the proposed project-specific model quality assurance process should consequently be automated. Therefore, this thesis also presents a flexible tool environment for model quality assurance based on the Eclipse Modeling Framework (EMF). The tool set is part of the Eclipse Modeling Project (EMP) and belongs to the Eclipse incubation project *EMF Refactor* which is available under the Eclipse public license (EPL). We evaluated the suitability of the tools for supporting the techniques of the model quality assurance process by performing and analyzing several experiments and studies.

EMF Refactor supports both the model designer and the model reviewer by obtaining metrics reports, by checking for potential model deficiencies (called model smells) and by systematically restructuring models using refactorings. Automatically proposed refactorings as quick fixes for occurring smells and information on implications of a selected refactoring concerning new model smells widen the provided functionality and support an integrated use of the quality assurance tools. The functionality of EMF Refactor is integrated into several editors like standard tree-based EMF instance editors, graphical GMF-based editors as used by Papyrus UML, and textual editors provided by Xtext.

Model checks and refactorings can be specified by several specification mechanisms. Actually, EMF Refactor supports Java, OCL, and the model transformation language Henshin as possible specification approaches. Further specification languages can be inserted using suitable adapters. Finally, metrics can be composed to more complex metrics and refactorings can be composed by using a dedicated language named CoMReL (Composite Model Refactoring Language).

In summary, the author of this thesis is convinced that performing quality assurance processes is an essential task to obtain software products of high quality. Using the structured model quality assurance process and the corresponding tool environment presented in this thesis, model-based and model-driven development can be made more mature yielding software of higher quality.

16.2 OUTLOOK

Though the work presented in this thesis is rather comprehensive and elaborated, there are several directions for further enhancements. This final section gives outlook to current and future work for both conceptual and tool related issues.

On the conceptual side, further model quality assurance techniques may be considered. Especially, the use of modeling conventions that

have to be proven to be effective with respect to prevention of defects might be integrated into the quality assurance process. Here, adequate modeling conventions have to developed that are suitable to hinder the modeler from inserting smells.

Concerning model metrics, current work concentrates on automatically deducing metrics suites from meta model specifications. Here, the idea is to specify recurring patterns in MOF-based meta models using the abstract syntax and to derive several kinds of structural metrics therefrom. Here, a prototypical implementation based on Henshin (for pattern specification) and OCL (for metrics calculation) exists and will be shipped with the forthcoming major release of EMF Refactor.

Concerning model smells, there might be some which are difficult to describe by metrics or patterns. For example, *shotgun surgery* is a code smell which occurs when an application-oriented change requires changes in many different classes. This smell can be formulated also for models, but it is difficult to detect it by analyzing models. It is up to future work to develop an adequate technique for this kind of model smells.

Concerning model refactoring, we are currently working on a lightweight approach to synchronize model and code refactorings in order to fill the design/implementation gap [117]. Here, model-to-code correspondences are dynamically discovered by a matcher that takes e.g. names, types, and method signatures into account. The approach is supported by a prototypical implementation for synchronized refactorings of UML models and Java projects within the Eclipse IDE.

Concerning composite model refactoring, future work will concentrate on analyzing the preconditions of component refactorings with respect to their execution order and to deduce a composite precondition therefrom. Here, we think of using concepts from algebraic graph transformations like critical pair analysis (CPA) [35]. In a similar way, specifications of model smells and model refactorings could be analyzed in order to decide whether the refactoring (1) is usable to erase the smell, or (2) its application would insert a new one. However, in order to provide this functionality, an appropriate implementation of CPA is needed. Since Henshin actually does not provide CPA yet, current work concentrates on a prototype that translates Henshin transformation rules to AGG [12, 148] and uses AGG's CPA functionality.

Finally, we will continue with making our quality assurance tools still more user-friendly. Besides support for further available techniques and specification languages, performance and scalability shall be further optimized. Here, potential inefficiencies in the framework need to be analyzed and performance-oriented technologies for metric computation and smell detection need to be discussed. Another open issue is how to deal with false positives during model smell detection. These are concrete smell occurrences being actually non-

issues to be ignored. Here, we think of using mechanisms similar to `@SupressWarnings` in Java to indicate areas to be elided during a smell search. In the context of EMF, `EAnnotations` might be useful.

However, we are convinced that the current functionality provided by EMF Refactor represents a solid foundation for quality assurance of EMF-based models.

APPENDICES

The following appendices contain several additional material related to the main parts of this thesis. Appendices A to C contain catalogs with brief descriptions of metrics, smells, and refactorings for UML class models extracted from literature. More detailed specifications of selected smells and refactorings for UML class models can be found in Appendices D and E whereas Appendix F presents details on the implementations of the specified refactorings. Finally, appendices G and H contain material concerning the experiments presented in Chapter 14 of this thesis.

A

A CATALOG ON UML CLASS MODEL METRICS

This appendix contains a catalog on metrics for UML class models found in literature. First, we present a number of basic metrics just used to define more complex ones. These are just named, very shortly described, and equipped with references from literature. Later, more complex metrics are listed, shortly explained, and assigned to quality aspects which can potentially be measured by the metric.

A.1 BASIC METRICS

The basic metrics are ordered according to their contexts, i.e., the UML meta model element type the metric is calculated on. Some of them refer to the entire model, some to single packages, and some to classes. Within each category the metrics are presented in alphabetic order.

A.1.1 *Context element:* ***Model***

1. **NAggH** - Total number of aggregation hierarchies. [69]
2. **NAGM** - Total number of aggregations in the model. [86, 69]
3. **NAM** - Total number of attributes in classes in the model. [71]
4. **NASM** - Total number of associations in the model. [86, 69]
5. **NCM** - Total number of classes in the model. [86, 105, 70, 6, 72]
6. **NDep** - Total number of dependency relationships. [69]
7. **NIH** - Total number of inheritance hierarchies in the model. [84, 105, 70, 69, 6, 72]
8. **NIM** - Total number of inheritance relations in the model. [86, 69]
9. **NOPM** - Total number of operations in the model. [71]
10. **NPN** - Total number of packages in the model. [86]

A.1.2 *Context element:* ***Package***

1. **NACP** - Number of abstract classes within the package. [109, 84]
2. **NAggR** - Number of aggregation relationships within the package. [70, 72]
3. **NAP** - Total number of associations within the package. [70, 72]
4. **NCP** - Number of all classes in the package. [84]
5. **NIP** - Number of interfaces within the package. [109, 84]
6. **NPP** - Number of nested packages inside the package. [84]
7. **R** - Number of relationships between classes and interfaces within the package. [110]

A.1.3 *Context element:* ***Class***

1. **DAC** - Number of attributes that have another class as type. [99, 22, 72]
2. **DAC'** - Number of different classes that are used as types of attributes. [99, 72]
3. **EC_Attr** - Number of times the class is externally used as attribute type. [22]
4. **EC_Par** - Number of times the class is externally used as parameter type. [22]
5. **IC_Par** - Number of parameters in the class having another class or interface as their type. [22]
6. **NAA** - Total number of associations with other classes or with itself. [81, 70, 84, 69, 72]
7. **NAC** - Total number of associations with other classes. [86]
8. **NAI** - Number of attributes visible to subclasses (public and protected, including inherited attributes). [84]
9. **NAP** - Number of public attributes (including inherited attributes). [84]
10. **NASC** - Total number of associations. [100, 70, 72]
11. **NATC** - Total number of attributes (unweighted). [100, 86]
12. **NCM** - Number of class methods. [100, 70, 72]
13. **NIA** - Number of inherited associations. [84]

14. **NID** - Number of internal dependencies (within the package of the class). [109, 84]
15. **NIM** - Number of instance methods. [100, 70, 72]
16. **NIV** - Number of instance variables. [100, 70, 72]
17. **NLA** - Number of local associations. [84]
18. **NMA** - Number of methods defined in a subclass. [100, 70, 72]
19. **NMI** - Number of methods inherited by a subclass. [100, 70, 72]
20. **NMO** - Number of methods overridden by a subclass. [100, 70, 72]
21. **NODP** - Number of direct part classes which compose a composite class. [69, 70, 72]
22. **NOPC1** - Number of operations (unweighted). [24, 100, 86]
23. **NOM** - Number of local methods. [99, 72]
24. **NSUBC (NOC)** - Number of direct children. [24, 100, 105, 70, 84, 86, 8, 72, 113]
25. **NSUBC*** - Number of all children. [86]
26. **NSUPC** - Number of direct parents. [24, 94, 84, 86]
27. **NSUPC*** - Total number of ancestors. [86]
28. **PIM** - Number of public instance methods. [100, 70, 72]
29. **SIZE2** - Number of attributes plus number of local methods. [99, 72]

The catalog of complex metrics is similarly structured to the catalog of basic metrics. Since the complex metrics are the proper ones to measure model quality, they are presented in more detail. For each metric its name, a short description, the range of values, and its potential interpretation are given. The definition of a complex metric might rely on one or more basic metrics previously presented. Again, also the complex metrics are ordered according to their contexts. Within each category the metrics are presented in alphabetic order.

A.2.1 *Context element:* ***Model***

1. **AGvsC**

   DESCRIPTION: Relation between number of aggregations and number of classes. [71]. This metrics is defined as $AGvsC = (\frac{NAGM}{NAGM+NCM})^2$ where $NAGM$ is the total number of aggregations in the model, NCM is the total number of classes in the model, and $(NAGM + NCM) > 0$.

   RANGE: $0 \leqslant AGvsC < 1$

   INTERPRETATION: This metric is a complexity measurement for class diagrams. According to [71] and with regard to complexity, the worst case is when the metric value tends to be 1, and the best case when the metric value tends to be 0.

2. **ANA**

   DESCRIPTION: Average number of ancestors of all classes. [6]

   RANGE: $0 \leqslant ANA \leqslant$ total number of classes (NCM) -1

   INTERPRETATION: A class with many ancestors inherits possibly many features. For this reason a class model with a high ANA value can be considered as complex.

3. **AvsC**

   DESCRIPTION: Relation between the number of attributes and number of classes [71]. This metrics is defined as $AvsC = (\frac{NAM}{NAM+NCM})^2$ where NAM is the total number of attributes in the model, NCM is the total number of classes in the model, and $(NAM + NCM) > 0$.

   RANGE: $0 \leqslant AvsC < 1$

   INTERPRETATION: If the value is higher, classes have more attributes and the model can be considered to be more complex. It is also possible that the model contains unnecessary

information and does therefore not correspond to the modeling purpose. On the other hand, a lower value could be a hint for relevant but missing information.

4. **ASvsC**

   DESCRIPTION: Relation between number of associations and number of classes [71]. This metrics is defined as $ASvsC = (\frac{NASM}{NASM+NCM})^2$ where $NASM$ is the total number of associations in the model, NCM is the total number of classes in the model, and $(NASM + NCM) > 0$.

   RANGE: $0 \leqslant ASvsC < 1$

   INTERPRETATION: This metric is a complexity measurement for class diagrams. According to [71] and with regard to complexity, the worst case is when the metric value tends to be 1, and the best case when the metric value tends to be 0.

5. **DEPvsC**

   DESCRIPTION: Relation between number of dependencies and number of classes [71]. This metrics is defined as $DEPvsC = (\frac{NDep}{NDep+NCM})^2$ where $NDep$ is the total number of dependencies in the model, NCM is the total number of classes in the model, and $(NDep + NCM) > 0$.

   RANGE: $0 \leqslant DEPvsC < 1$

   INTERPRETATION: This metric is a complexity measurement for class diagrams. According to [71] and with regard to complexity, the worst case is when the metric value tends to be 1, and the best case when the metric value tends to be 0.

6. **GEvsC**

   DESCRIPTION: Relation between number of generalizations and number of classes [71]. This metrics is defined as $DEPvsC = (\frac{NIM}{NIM+NCM})^2$ where NIM is the total number of generalization relations in the model, NCM is the total number of classes in the model, and $(NIM + NCM) > 0$.

   RANGE: $0 \leqslant GEvsC < 1$

   INTERPRETATION: If the value is high, the classes are stronger coupled due to inheritance. Hence, the class model inheritance hierarchy can be considered as complex.

7. **MaxDIT**

   DESCRIPTION: Maximum of all depth of inheritance trees. [69]

   RANGE: $0 \leqslant MaxDIT \leqslant$ total number of classes (NCM) -1

INTERPRETATION: A class model with a deeper inheritance hierarchy can be considered as complex. In this cases the value of MaxDIT is high.

8. **MaxHAgg**

   DESCRIPTION: Maximum of aggregation trees. [69]

   RANGE: $0 \leqslant MaxHAgg \leqslant$ total number of classes (NCM) -1

   INTERPRETATION: A class model with deeper aggregation trees can be considered as complex. In that case, the value of MaxHAgg is high.

9. **MEvsC**

   DESCRIPTION: Ratio between number of methods (operations) and number of classes [71]. It is defined as $DEPvsC = (\frac{NOPM}{NOPM+NCM})^2$ where $NOPM$ is the total number of operations in the model, NCM is the total number of classes in the model, and $(NOPM + NCM) > 0$.

   RANGE: $0 \leqslant MEvsC < 1$

   INTERPRETATION: If the value is higher, the classes have a lot of methods (operations) and the model can be considered to be complex.

10. **M_{GH}**

    DESCRIPTION: Complexity due to generalization hierarchies. The detailed definition can be found in [71].

    RANGE: $0 \leqslant M_{GH}$

    INTERPRETATION: Higher values indicate more complex generalization hierarchies.

11. **M_{MI}**

    DESCRIPTION: Complexity due to multiple inheritance. The detailed definition can be found in [71].

    RANGE: $0 \leqslant M_{MI}$

    INTERPRETATION: Higher values indicate a higher complexity due to multiple inheritance.

12. **OA3**

    DESCRIPTION: Average of the weighted numbers of class responsibilities. [105, 70, 72]

    RANGE: $0 \leqslant OA3$

    INTERPRETATION: This metric measures the global complexity of the entire class model. Higher values indicate more complex models.

13. **OA4**

    DESCRIPTION: Standard deviation of the weighted numbers of class responsibilities. [105, 70, 72]

    RANGE: $0 \leqslant OA4$

    INTERPRETATION: This metric measures the global complexity of the entire class model. Higher values indicate more complex models.

14. **OA5**

    DESCRIPTION: Average of the number of direct dependencies of classes. [105, 70, 72]

    RANGE: $0 \leqslant OA5$

    INTERPRETATION: This metric measures the global complexity of the entire class model. Higher values indicate more complex models.

15. **OA6**

    DESCRIPTION: Standard deviation of the number of direct dependencies of classes. [105, 70, 72]

    RANGE: $0 \leqslant OA6$

    INTERPRETATION: This metric measures the global complexity of the entire class model. Higher values indicate more complex models.

16. **OA7**

    DESCRIPTION: Ratio between number of inherited responsibilities and total number of responsibilities. [105, 70, 72]

    RANGE: $0 \leqslant OA7 \leqslant 1$

    INTERPRETATION: This metric measures the level of reuse. If the value is higher, features are reused more often. According to [70], the aim of this metric is to measure the complexity of a class model.

A.2.2 *Context element:* ***Package***

1. **A**

    DESCRIPTION: Ratio between number of abstract classes (and interfaces) and total number of classes within the package (abstractness) [84, 109, 110]. It is defined as $A = \frac{NACP+NIP}{NCP+NIP}$ where $NACP$ is the number of abstract classes within the package, NIP is the number of interfaces within the package, and NCP is the number of classes within the package.

    RANGE: $0 \leqslant A \leqslant 1$

INTERPRETATION: A higher value indicates a heavier use of abstract classes and interfaces making the model harder to understand. This could be interpreted differentially. First, the modeler(s) could use the UML language feature of abstract classes respectively interfaces too exhaustively and therefore not in sync with the modeling purpose. Second, classes could be marked as abstract by mistake. Third, a high abstractness value could be a hint for relevant but missing concrete classes.

2. **AHF**

   DESCRIPTION: Ratio between the number of invisibile attributes and total number of attributes within a package (attribute hiding factor). [23, 70, 72]

   RANGE: $0 \leqslant AHF \leqslant 1$

   INTERPRETATION: Ideally all attributes should be hidden in classes. Hence, a value close to 1 is preferred.

3. **AIF**

   DESCRIPTION: Ratio between the number of inherited attributes and total number of attributes within the package (attribute inheritance factor). [23, 70, 72]

   RANGE: $0 \leqslant AIF \leqslant 1$

   INTERPRETATION: This metric measures the level of reuse. A higher value indicates a higher level of reuse. Nevertheless a very high value may indicate a complex model.

4. **Ca**

   DESCRIPTION: Number of classes in other packages that depend on classes within the package (afferent coupling). [109, 84, 110]

   RANGE: $0 \leqslant Ca \leqslant$ total number of classes within other packages

   INTERPRETATION: The value should be low.

5. **Ce**

   DESCRIPTION: Number of classes in other packages that the class within the package depend on (efferent coupling). [109, 84, 110]

   RANGE: $0 \leqslant Ca \leqslant$ total number of classes within other packages

   INTERPRETATION: The value should be low.

6. **DN**

DESCRIPTION: Normalized distance of the package from the main sequence. [84, 110]

RANGE: $0 \leqslant DN \leqslant 1$

INTERPRETATION: The main sequence is part of a theory of Martin which states that the abstractness A and the instability I of a package should be about the same. That is, abstractions have to be very stable, concrete implementations may change more [84]. The higher the value the more it is worse.

7. **DNH**

   DESCRIPTION: Depth in the nesting hierarchy. [84]

   RANGE: $0 \leqslant DNH \leqslant$ total number of packages in the model (NPN) -1

   INTERPRETATION: The nesting level of containment hierarchies should not be too deep, say 5 to 7 as maximum, according to [84].

8. **H**

   DESCRIPTION: Ratio between number of relationships between classes within the package and total number of classes within the package (Relational Cohesion). [109, 84, 110]

   RANGE: $0 \leqslant H \leqslant$ total number of relationships within the package

   INTERPRETATION: Classes inside a package should be strongly related what means the cohesion should be high. A high value indicates a high cohesion.

9. **I**

   DESCRIPTION: Ratio between efferent coupling and total coupling (Instability) [1]. [109, 84, 110]

   RANGE: $0 \leqslant I \leqslant 1$

   INTERPRETATION: The value should be low. If the value is high, classes are more coupled in between packages than within one package.

10. **MHF**

    DESCRIPTION: Ratio between the number of invisible methods and total number of methods within the package (method hiding factor). [23, 70, 72]

    RANGE: $0 \leqslant MHF \leqslant 1$

1 total coupling = afferent coupling + efferent coupling

INTERPRETATION: According to [23] the value should grow if the model is refined and gets more details. If the value is small, the model can be assumed to be of an earlier phase.

11. **MIF**

    DESCRIPTION: Ratio between the number of inherited methods and total number of methods within the package (method inheritance factor). [23, 70, 72]

    RANGE: $0 \leqslant MIF \leqslant 1$

    INTERPRETATION: This metric can be used to measure the level of reuse. A high value indicates a high level of reuse. According to [23] the value should not be too high (smaller than 0.7/0.8), because inheritance is wrongly used in this case with negative effects to the maintainability and defect density of the modeled solution. However, there has been no empirical evaluation.

12. **NAVCP**

    DESCRIPTION: Ratio between number of associations within the package and total number of classes within the package. [70, 72]

    RANGE: $0 \leqslant NAVCP \leqslant$ total number of associations within the package (NAP)

    INTERPRETATION: The more associations per class a package has, the more complex it is to maintain and understand. Hence, a higher value indicates higher complexity.

13. **PF**

    DESCRIPTION: Ratio between the actual number of different possible polymorphic situations and its maximum. [23, 70, 72]

    RANGE: $0 \leqslant PF \leqslant 1$

    INTERPRETATION: PF measures the potential polymorphism. If the value is low, many methods are overridden and the usage of polymorphism is high.

14. **PK1**

    DESCRIPTION: Usage of classes of other packages by classes of this package. [105, 70, 72]

    RANGE: $0 \leqslant PK1$

    INTERPRETATION: The aim of this metric is to measure inter-package coupling. The metric measures the dependency of the classes of a given package from exterior classes.

15. **PK2**

DESCRIPTION: Reuse of package classes by classes of other packages. [105, 70, 72]

RANGE: $0 \leqslant PK2$

INTERPRETATION: The aim of this metric is to measure inter-package coupling. A low value says that packages are not strongly coupled. The metric measures the dependency of exterior classes to package classes.

16. **PK3**

DESCRIPTION: Average number of other package's classes usages by classes of a package. [105, 70, 72]

RANGE: $0 \leqslant PK3$

INTERPRETATION: The aim of this metric is to measure inter-package coupling. It is the average value of PK1 metric for all packages.

A.2.3 *Context element:* ***Class***

1. **APPM**

DESCRIPTION: Average number of parameters of all methods within the class. [100, 70, 72]

RANGE: $0 \leqslant APPM \leqslant$ total number of parameters of all methods within the class

INTERPRETATION: According to [100], parameters require more effort from clients, and high and low numbers of parameters imply a style of design. Lorenz and Kidd suggest 0.7 parameters per method as upper threshold.

2. **CBC**

DESCRIPTION: Number of attributes and associations with class types (Coupling between classes). [86]

RANGE: $0 \leqslant CBC$

INTERPRETATION: A high value indicates a class that is strongly coupled to other classes.

3. **CBO**

DESCRIPTION: Total number of classes a class is coupled to. [24, 84, 8, 113]

RANGE: $0 \leqslant CBO \leqslant$ total number of classes in the model (TNC) -1

INTERPRETATION: A class that depends on too many other classes can indicate a bad modular design. Highly coupled classes are harder to test and maintain. A high CBO value indicates a class that is highly coupled.

4. **CL1**

   DESCRIPTION: Weighted number of responsibilities. [105, 70, 72]

   RANGE: $0 \leqslant CL1$

   INTERPRETATION: This metric is defined as measurement for the class complexity. Higher values indicates more complex classes.

5. **CL2**

   DESCRIPTION: Weighted number of dependencies. [105, 70, 72]

   RANGE: $0 \leqslant CL2$

   INTERPRETATION: This metric is defined as measurement for the class complexity. Higher values indicates more complex classes.

6. **DAM**

   DESCRIPTION: Ratio between number of private and protected attributes and total number of attributes (data access metric). [6, 72]

   RANGE: $0 \leqslant DAM \leqslant 1$

   INTERPRETATION: This metric is used to measure the encapsulation. If the value is close to 1, the encapsulation is good.

7. **DCC**

   DESCRIPTION: Number of different classes the class is directly related to (direct class coupling) [6, 72]. This metric is similar to metric CBO but it considers only coupling because of attributes or operation parameters.

   RANGE: $0 \leqslant DCC \leqslant$ total number of classes in the model (TNC) -1

   INTERPRETATION: A high DCC value indicates a class that is highly coupled.

8. **DIT**

   DESCRIPTION: Depth of inheritance tree. [24, 105, 72, 86, 8, 72, 113]

   RANGE: $0 \leqslant DITC$

   INTERPRETATION: The higher the DIT, the greater the chance of reuse becomes. However, a high value of DIT can cause program comprehension problems. [86]

9. **HAgg**

   DESCRIPTION: Length of the longest path to the leaves in the aggregation hierarchy. [69, 70, 72]

RANGE: $0 \leqslant HAgg$

INTERPRETATION: The metric measures the class complexity due to aggregation relations. A higher value indicates a higher complexity.

10. **MAgg**

DESCRIPTION: Number of direct or indirect whole classes within an aggregation hierarchy. [69, 70, 72]

RANGE: $0 \leqslant MAgg$

INTERPRETATION: This metric measures the class complexity due to multiple aggregation relations [70]. A higher value indicates a higher complexity.

11. **MFA**

DESCRIPTION: Ratio between number of inherited methods and total number of instance methods (measure of functional abstraction). [6, 72]

RANGE: $0 \leqslant MFA \leqslant 1$

INTERPRETATION: This metric is defined to assess design property inheritance. If the value is high, many operations are inherited.

12. **NASC**

DESCRIPTION: Number of linked associations including aggregations. [86]

RANGE: $0 \leqslant NASC$

INTERPRETATION: This metric is useful for estimating the static relationships between classes. [86]

13. **NATC2**

DESCRIPTION: Number of attributes weighted by their visibility kind e.g. 1.0 for public, 0.5 for protected, and 0 for private. [86]

RANGE: $0 \leqslant NATC2$

INTERPRETATION: Due to encapsulation, the value of this metric should not be not too high.

14. **NDepIn**

DESCRIPTION: Number of distinct classes depending on the class. [105, 69, 70, 72]

RANGE: $0 \leqslant NDepIn$

INTERPRETATION: The greater the number of classes is that depend on a given class, the greater the inter-class dependency and therefore the greater the design complexity of

such a class. The inter-class dependency is also called export coupling, which if misused could be a potential source of design complexity [70]. A higher NDepIn value indicates a complex class.

15. **NDepOut**

    DESCRIPTION: Number of classes on which the class depends. [69, 70, 72]

    RANGE: $0 \leqslant NDepOut$

    INTERPRETATION: The greater the number of classes on which a given class depends, the greater the inter-class dependency and therefore the greater the design complexity of such a class. This inter-class dependency is also called import coupling, which if misused could be a potential source of design complexity. It is better to minimize NDepOut value, since, higher values represent a situation in which many dependencies are spreading across the class diagram [70]. A higher NDepOut value indicates a complex class.

16. **NP**

    DESCRIPTION: Total number of direct or indirect part classes of a whole class. [69, 70, 72]

    RANGE: $0 \leqslant NP$

    INTERPRETATION: The metric measures the class complexity due to aggregation relations. A higher value indicates a higher complexity.

17. **NW**

    DESCRIPTION: Number of direct or indirect whole classes of a part class. [69, 70, 72]

    RANGE: $0 \leqslant NW$

    INTERPRETATION: The metric measures the class complexity due to aggregation relations. A higher value indicates higher complexity.

18. **RFC**

    DESCRIPTION: Number of methods plus number of used methods of other classes (Response for a Class). [24, 84, 8, 113]

    RANGE: $0 \leqslant RFC$

    INTERPRETATION: The number should not be too high. A small response set is better [84].

19. **SIX**

DESCRIPTION: Ratio between weighted number of overridden methods and total number of methods (specialization index). [100, 70, 72]

RANGE: $0 \leqslant SIX \leqslant$ total number of methods (NOPC1)

INTERPRETATION: Lorenz and Kidd [100] have commented that this weighted calculations has done a good job on identifying classes worth looking at for their placement in the inheritance hierarchy and for design problems.

20. **WMC [NOPC2]**

DESCRIPTION: Weighted methods per class [24, 84, 86, 8, 72, 113]. NOPC2 is a special version of metric WMC that weights the methods according their visibility.

RANGE: $0 \leqslant WMC$

INTERPRETATION: A class should have a reasonable responsibility. A good value for WMC depends on the weighting and other criteria e.g. project phase etc.

B

A CATALOG ON UML CLASS MODEL SMELLS

This appendix contains a catalog on smells for UML class models found in literature. The smells in this catalog are presented in alphabetic order. For each smell we give a name, a short description, and a corresponding source.

1. **Association Class** - Association classes cannot be directly translated to common programming languages. They defer the decision which class(es) will be responsible to manage the association attributes eventually. [66, 119]
2. **Attribute Name Overridden** - The class defines a property with the same name as an inherited attribute. During code generation, this may inadvertently hide the attribute of the parent class. [83]
3. **Attributes On Interfaces** - The interface has attributes or outgoing associations. This rather appears to be a concession to certain component technologies, and should be avoided otherwise. [119]
4. **Classes without Methods** - A class without any method does not provide any functionality. [97]
5. **Concrete Superclass** - An abstract class being subclass of a concrete class reflects poor design and a conflict in the model's inheritance hierarchy. [96]
6. **Data Class** - Classes with attributes, getters, and setters only. [11]
7. **Data Clumps** - The same (three or four) data items can be found in lots of places (fields in classes or parameters in method signatures). They really ought to be an object. [11]
8. **Dependency Cycle** - Cycles in the dependency graph should be avoided. The elements participating in the cycle cannot be tested, reused, or released independently. [110]
9. **Descendant Reference** - The reference to the descendent class and the inheritance links back to the class effectively form a dependency cycle between these classes. [83, 133]

10. **Diamond Inheritance** - This smell is based on the multiple inheritance concept of UML. It occurs when the same predecessor is inherited by a class several times and is known in literature as 'diamond' inheritance problem for object-oriented techniques using multiple inheritance and was first discussed by Sakkinen [136].

11. **Large Class** - The class has too many features (too many properties and/or operations) belonging to different concerns. There is a significant difference in the relative size of this class to other classes. [5]

12. **Lazy Class** - Lazy classes are small, have few methods, and little behavior. They stand out in a class diagram because they are so small. [5]

13. **Long Parameter List** - The operation has a long list of parameters that makes it really uncomfortable to use the operation. [11]

14. **Middle Man** - Objects hide internal details but encapsulation leads to delegation. [11]

15. **Multiple Definitions of Classes with Equal Names** - Several classes have the same name. The set of classes with same name may be contained in one diagram or in different diagrams. [97]

16. **N-ary Aggregation** - People are often confused by the semantics of n-ary associations. N-ary associations cannot be directly translated to common programming languages. [66, 119]

17. **No Attribute Type** - Without a type, the attribute has no meaning in design, and code generation will not work. [66]

18. **No Parameter Type** - Without a type, the parameter has no meaning in design, and code generation will not work. [66]

19. **No Specification** - Abstract classes cannot be instantiated. Without specializations that can be instantiated, the abstract class is useless. [133]

20. **Primitive Obsession** - People new to objects are reluctant to use small objects for small tasks. [11]

21. **Public Attribute** - The non-constant attribute is public. External read/write access to attributes violates the information hiding principle. Allowing external entities to directly modify the state of an object is dangerous. [133]

22. **Specialization Aggregation** - People are often confused by the semantics of specialized associations. The suggestion is therefore to model any restrictions on the parent association using constraints. [119]

23. **Speculative Generality** - Hooks and special cases to handle things that are not required. [11]

24. **Unnamed Element** - The model element, i.e., package, class, interface, data type, attribute, operation, or parameter, has no name. [83]

25. **Unused Class** - The class has no child classes, dependencies, or associations, and it is not used as parameter or property type. [133]

26. **Unused Interface** - The interface is not implemented anywhere, has no associations, and is not used as parameter or attribute type. [133]

C

A CATALOG ON UML CLASS MODEL REFACTORINGS

In this appendix follows a list of refactorings for UML class models found in literature. The refactorings in this catalog are presented in alphabetic order. For each refactoring we give a name, a short description, and at least one corresponding source. We further distinguish basic and complex refactorings. We can consider basic refactorings as atomic ones, while complex refactorings are composed from basic ones. Detailed specifications of selected refactorings can be found in the subsequent Appendix E of this thesis.

C.1 BASIC REFACTORINGS

1. **Add Parameter** - An operation needs more information from its callers. This refactoring adds a parameter to an operation. [30, 150]
2. **Create Associated Class** - Creates an empty class and connects it with a new association to the source class from where it is extracted. The multiplicities of the new association are1 at both ends. [107, 150]
3. **Create Subclass** - A class has features that are not used in all instances. The refactoring creates a subclass for that subset of features. [150]
4. **Create Superclass** - Creates a superclass for at least one class and is normally followed by refactorings 'Pull Up Property' and 'Pull Up Operation'. [141, 150, 107, 160]
5. **Hide Property** - This refactoring makes a public attribute [property] of a given class private, and creates the associated getter [and setter] operation. [128, 129]
6. **Move Operation** - This refactoring moves an operation from one class to a related one. It is often applied when some class has too much behavior or when classes collaborate too much. [107, 150]

7. **Move Property** - A property is better placed in another class which is related to the class. This refactoring moves the property from one class to another. [107, 150, 93]

8. **Pull Up Operation** - This refactoring pulls an operation of a class to its superclass or to an implemented interface. Usually this refactoring is used simultaneously on several classes which inherit from the same superclass or implement the same interface. The aim of this refactoring is often to extract identical operations. [137, 107, 150]

9. **Pull Up Property** - This refactoring pulls a property from a class or a set of classes to some superclass. It can also be applied when the superclass is an interface. [14, 107, 30, 150]

10. **Push Down Operation** - An operation of a superclass is pushed down to all its subclasses. This refactoring can also be used to push down an operation from an interface to its implementations or sub-interfaces. [107, 150, 93]

11. **Push Down Property** - The attribute (property) is used by few subclasses only. This refactoring moves the attribute (property) to the using subclasses only. [107, 30, 150]

12. **Remove Parameter** - A parameter is no longer needed to specify an operation. The refactoring removes this parameter from the operation. [30, 150]

13. **Rename Class** - The current name of a class does not reflect its purpose. This refactoring changes the name of the class to a new name. This refactoring can also be applied to interfaces. [30]

14. **Rename Operation** - The current name of the operation does not reflect its purpose. This refactoring changes the name of the operation to a new name. [107, 30]

15. **Rename Property** - The current name of an attribute (property) does not reflect its purpose. This refactoring changes the name of the attribute (property). [107, 30]

16. **Replace Delegation With Inheritance** - This refactoring is the opposite of the following refactoring 'Replace Inheritance With Delegation'. [5]

17. **Replace Inheritance With Delegation** - Often it is useful to work with composition and delegation rather than with inheritance. If you create a model for an application which should be realized in a programming language that does not have multiple inheritance, this principle might become important. This refactoring removes the inheritance relation, adds an association, and adds a delegating method. [5]

1. **Extract Class** - This refactoring extracts interrelated features from a class to a new separated class. [107, 150]

2. **Extract Subclass** - There are features in a class required for a special case only. This refactoring extracts a subclass containing these features. The subclass can also be an interface if the class considered is already an interface. [150]

3. **Extract Superclass** - There are two or more classes with similar features. This refactoring creates a superclass and moves the common features to the superclass. The refactoring helps to reduce redundancy by assembling common features spread throughout different classes. [141, 106, 150, 107, 160]

4. **Inline Class** - There are two classes connected by a 1:1 association. One of them has no further use. This refactoring merges these classes. [150, 93]

5. **Introduce Parameter Object** - There is a group of parameters that naturally go together. This refactoring replaces a list of parameters with one object. This parameter object is created for that purpose. [11, 131, 161, 104]

6. **Remove Middle Man** - A middle man class is doing mostly simple delegation. This refactoring removes this class and associates the corresponding classes directly. [30, 93]

D

SPECIFICATIONS OF UML CLASS MODEL SMELLS

In this appendix we describe selected smells for UML class models found in literature. For each model smell a short description is given as well as possible indicators to detect this smell in a given model. Furthermore, we present a list of quality characteristics and quality goals affected by this smell. Lists of refactorings suitable for eliminating the smell and an example complete each model smell description.

D.1 attribute name overridden

DESCRIPTION The class defines a property with the same name as an inherited attribute. For this smell, it is essential that the property redefines the inherited attribute in order to conform to the UML specification. The redefinition of attributes might be confusing to model viewers. Furthermore, this smell might produce conflicts in model-driven processes. During code generation, this smell may inadvertently hide the attribute of the parent class. [83]

EXAMPLE Figure D.1 shows an attribute *horsepower* in class *Car* that redefines the equally named attribute in abstract superclass *Vehicle*. This is done to specialize the type of the attribute, i.e., there is a restriction of the attribute's type. Sometimes, such a redefinition might be confusing and decreases the model's comprehensibility.

DETECTION This smell can be detected by matching a corresponding pattern based on the abstract syntax of UML. Figure D.2 shows the Henshin rule that specifies this pattern-based smell. This rule defines two UML `Properties` (named *attribute_1* and *attribute_2*) which have the same name (specified by the internal rule parameter *attributename*). Furthermore, there are two additional conditions which must be fulfilled. First, the owning class of property *attribute_1* must inherit property *attribute_2*. This is defined by using a so-called *positive application condition* (PAC) named *InheritedAttribute* and represented by tags $\langle\langle require \rangle\rangle$. Second, property *attribute_1* must NOT redefine property *attribute_2* since otherwise this would represent a non-smelly modeling.

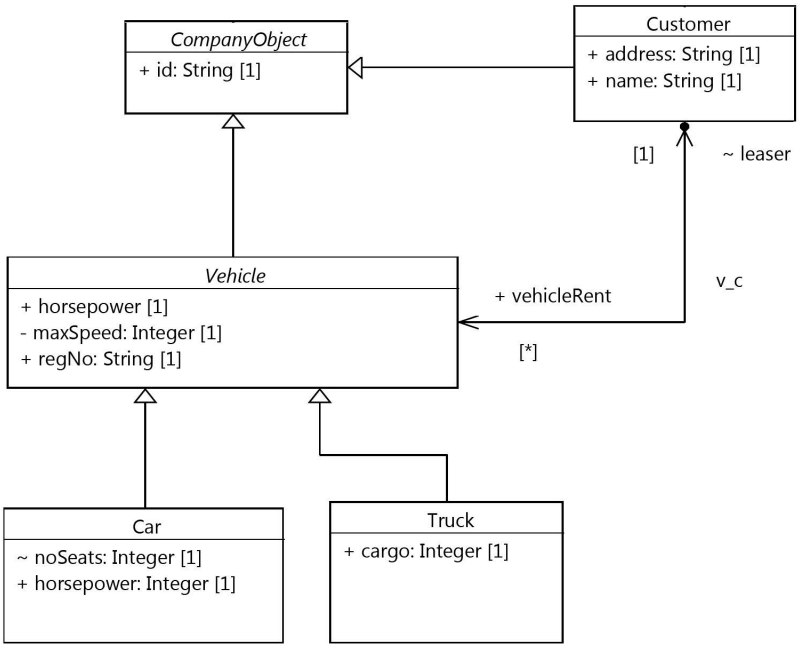


Figure D.1: Example UML model smell *Attribute Name Overridden*

The absence of this redefinition relationship is defined by a so-called *negative application condition* (NAC) named *NoRedefinition* and represented by tags ⟨⟨*forbid*⟩⟩. In summary, the Henshin pattern rule in Figure D.2 specifies two equally named attributes of a class (one direct and one inherited attribute) which are not related by a redefinition relationship.

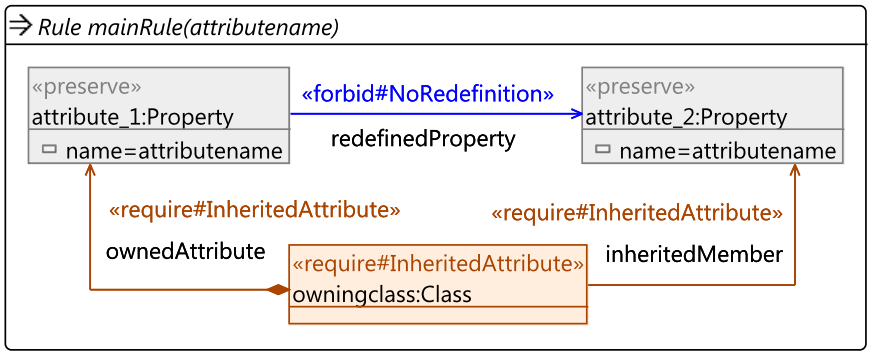


Figure D.2: Henshin pattern rule specification of UML model smell *Attribute Name Overridden*

USABLE UML MODEL REFACTORINGS *Rename Property* for renaming one of the involved attributes. Furthermore, the redefinition relation has to be deleted.

AFFECTED QUALITY CHARACTERISTICS AND GOALS Redefined attributes may lead to more complexity and might be a typical case for redundant modeling. Simplicity, Redundancy → Comprehensibility, Consistency, Confinement, Changeability, Correctness

D.2 concrete superclass

DESCRIPTION An abstract class that is a subclass of a non-abstract class reflects poor design and a conflict in the model's inheritance hierarchy. In other words, if an abstract class has any superclasses these have to be abstract as well. [96]

EXAMPLE Figure D.3 shows an example class hierarchy. Abstract class *PublicBuilding* together with its subclasses *Library* and *Church* represent a valid class hierarchy whereas class *House* exactly addresses smell *Concrete Superclass*. If this class were also an abstract class, for example named *Building*, the entire class hierarchy would be valid again.

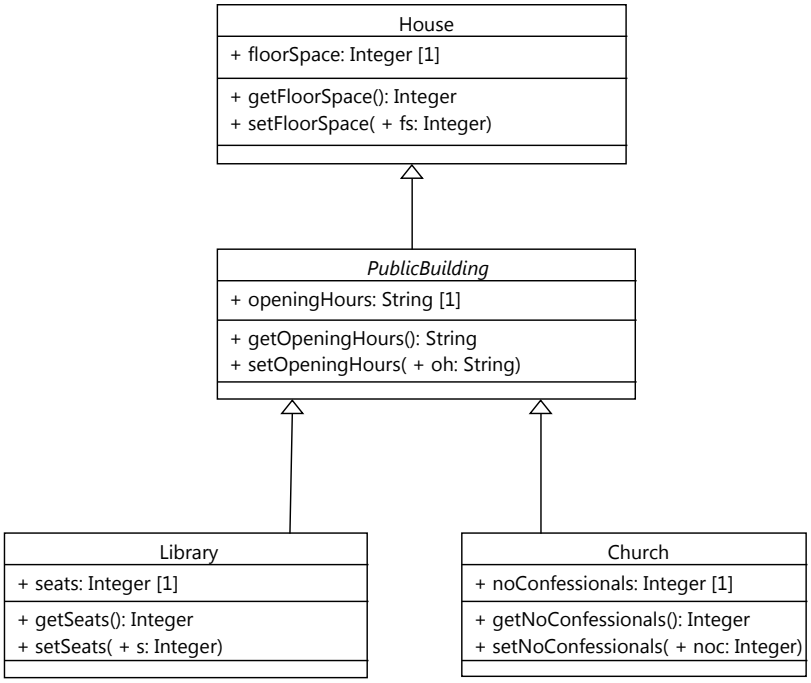


Figure D.3: Example UML model smell *Concrete Superclass*

DETECTION This smell can be either detected by matching corresponding patterns based on the abstract syntax of UML, or by evaluating model metric *Number of non-abstract superclasses*. Figure D.4 shows the Henshin rule that specifies this pattern-based smell. This pattern rule defines a UML `Class` *abstractclass* whose meta attribute *isAbstract* is set to *true*. Furthermore, there is one

additional PAC (*HasConcreteSuperclass*) that specifies that this abstract class has a generalization relationship to another class (*concreteclass*) whose meta attribute *isAbstract* is set to *false*. In summary, the Henshin pattern rule in Figure D.4 specifies an abstract class that has a direct concrete superclass.

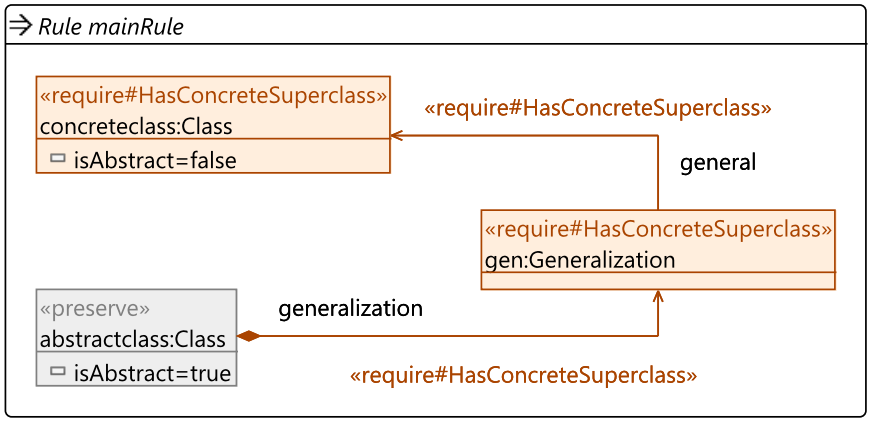


Figure D.4: Henshin pattern rule specification of UML model smell *Concrete Superclass*

USABLE UML MODEL REFACTORINGS No existing model refactoring can be used to eliminate this smell. Either it has to be developed, or the smell has to be eliminated directly, for example by making the class non-abstract as well.

AFFECTED QUALITY CHARACTERISTICS AND GOALS Concrete superclasses of abstract subclasses may not reflect a model aspect in the right way. Furthermore, this may lead to more complex models that are harder to understand. Precision, Simplicity → Correctness, Comprehensibility

D.3 data clumps

DESCRIPTION Fowler describes this smell as interrelated data items which often occur as 'clump' in the model. Often, there are the same three or four data items together in lots of places, either attributes in classes or parameters in operation signatures. They really ought to be an object [11]. Zhang et al. [161] present a more precise pattern-based definition of this model smell. For attributes they define:

1. More than three attributes stay together in more than one class.

2. These attributes should have same signatures (same names, same types, and same visibility).
3. These data fields may not group together in the same order.

For parameters they define:

1. More than three input parameters stay together in more than one operations' declaration.
2. These parameters should have same signatures (same names, same types).
3. These parameters may not group together in the same order.

EXAMPLE In Figure D.5 there are attributes *customerName*, *customerStreet customerZip*, and *customerCity* that occur in altogether four different classes.

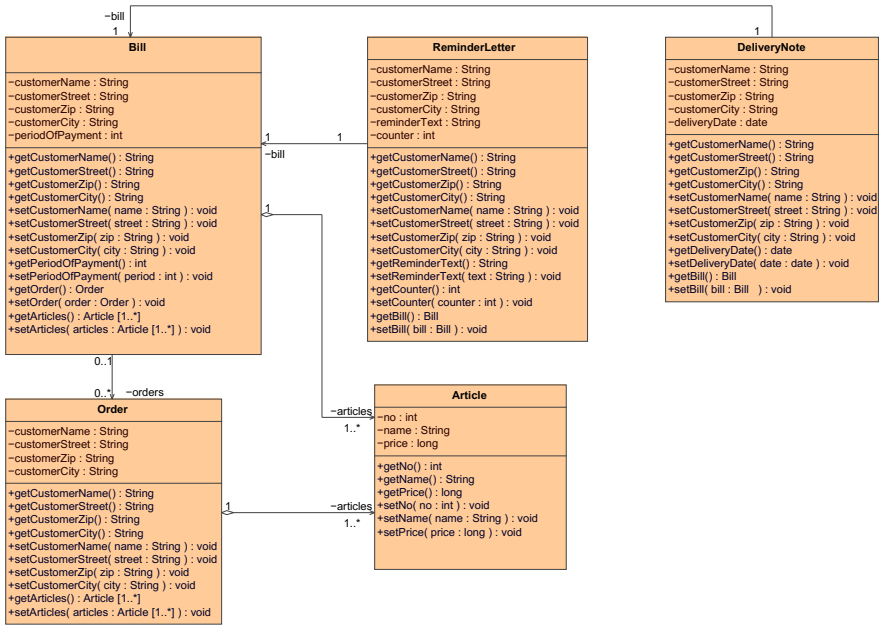


Figure D.5: Example UML model smell *Data Clumps*

DETECTION This smell can be detected by matching corresponding patterns based on the abstract syntax of UML. A more common alternative of this smell, independent from the number of involved elements, is to implemented this smell in a more general way than Zhang et al. (see description above) who specified a fix number of features (three). In contrast, we specify this smell using the two metrics *Number of Equal Attributes with other Classes* (for attributes) and *Number of Equal Input Parameters in Sibling Operations* (for parameters) and provide individual threshold values.

USABLE UML MODEL REFACTORINGS *Extract Class* for extracting the involved attributes into a new class, *Introduce Parameter Object* for extracting the involved parameters into a new class.

AFFECTED QUALITY CHARACTERISTICS AND GOALS Data clumps represent redundantly modeled aspects. They may be harder to understand and may not conform to a modular design. Redundancy, Simplicity, Cohesion/Modular Design → Comprehensibility, Changeability, Correctness

D.4 large class

DESCRIPTION A class should model an entity representing one single aspect of a given domain. So, its features (attributes and operations) should be balanced well. A class having too much features belonging to different concerns hints for too much information that should be expressed by this class. Often, this is the central class of a diagram. In this case, the surrounding classes may be inordinately small, which is also a smell. In any case, the significant difference in the relative sizes of the classes is the important thing. [11, 5]

EXAMPLE In Figure D.6 it is obvious that class *Bill* represents this model smell. Except for its remarkable number of operations which are mainly accessors or mutators, this class owns much more attributes than the average attribute number of the other classes.

DETECTION This model smell can be easily detected by observing the class diagram with all members shown. Another check is to use metrics *Number of Attributes* and *Number of Operations* to determine the relative sizes of the classes in a computational way. A further simplification is to use the corresponding class metric *Number of Features* and evaluating its value with respect to those concerning other classes. This metric can be specified by OCL query

```
self.ownedAttribute -> size()
+ self.ownedOperation -> size()
```

that returns the sum of the number of owned attributes and the number of owned operations of the contextual class.

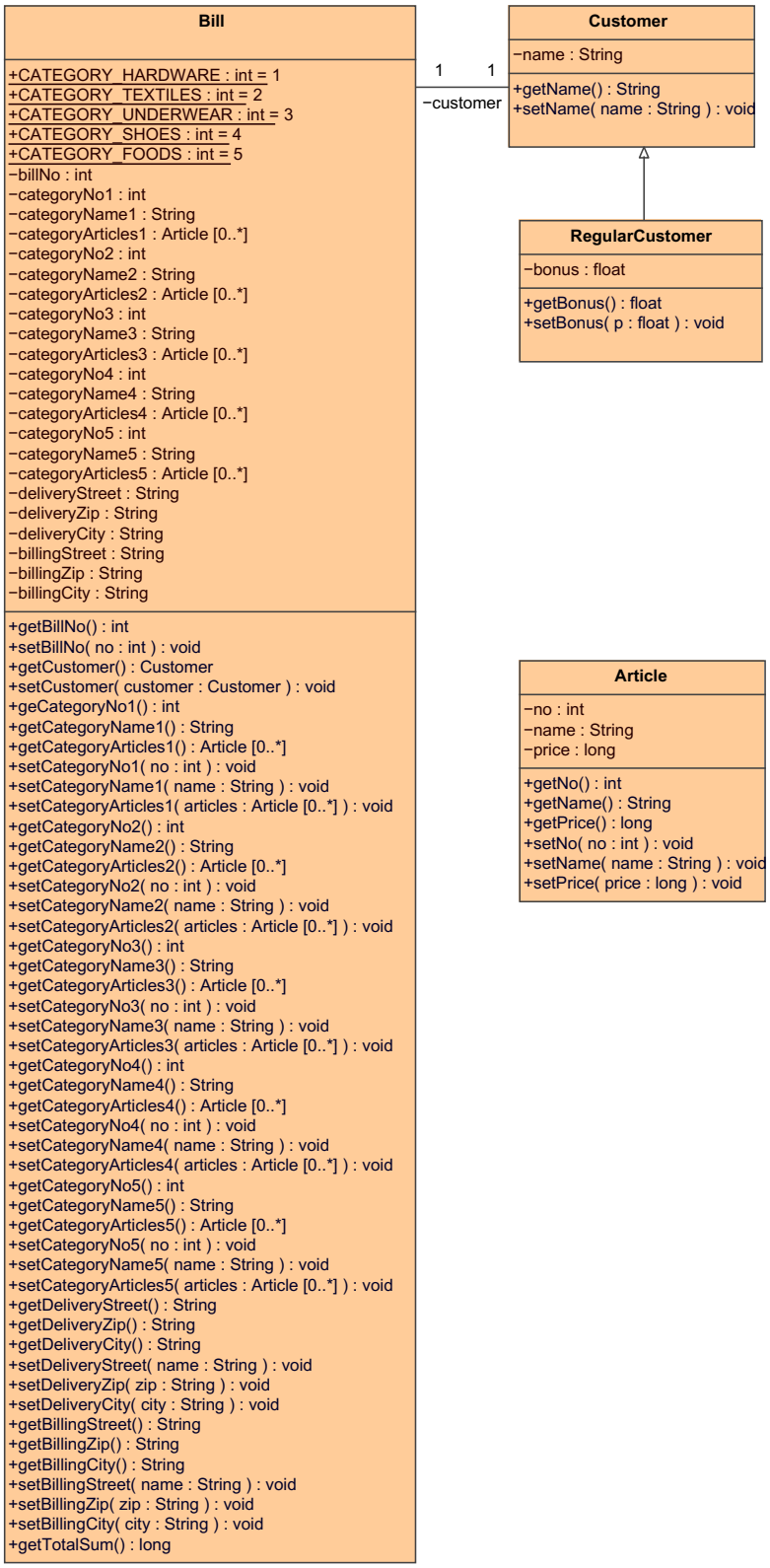


Figure D.6: Example UML model smell *Large Class*

USABLE UML MODEL REFACTORINGS *Extract Class*, *Extract Superclass*, or *Extract Subclass* for extracting information in a new class. *Move Property*, *Move Operation* for moving information to an associated class.

AFFECTED QUALITY CHARACTERISTICS AND GOALS Large classes do not represent a good modular design and may contain redundant information. Presentation, Cohesion/Modular Design, Redundancy → Comprehensibility, Changeability, Correctness

D.5 long parameter list

DESCRIPTION An operation has a long list of parameters that makes it really uncomfortable to use the operation. Long parameter lists are hard to understand and difficult to use. Furthermore, using long parameter lists is not intended by the object-oriented paradigm. An operation should have only as much parameters as needed for solving the corresponding task. It is recommended to pass only those parameters that cannot be obtained by the owning class itself. [11, 131]

EXAMPLE In Figure D.7 class *CustomerRelationshipManager* owns two operations each having a long parameter list. Here, this smell can easily be detected by observation.

DETECTION This smell can be simply detected by observing the model or by calculating the corresponding class metric *Number of Input Parameters* and evaluating its value with respect to a predefined threshold value. Metric *Number of Input Parameters* can be specified by OCL expression

```
self.ownedParameter
-> select(direction = ParameterDirectionKind::_in or
direction = ParameterDirectionKind::inout)
-> size()
```

that returns the number of owned parameters of a given operation with direction *in* respectively *inout*.

USABLE UML MODEL REFACTORINGS *Introduce Parameter Object* for extracting information to a new class. *Remove Parameter* for removing not needed information.

AFFECTED QUALITY CHARACTERISTICS AND GOALS Long parameter lists may be harder to understand and may contain redundant information. Presentation/Aesthetics, Simplicity, Cohesion/Modular Design → Comprehensibility, Changeability, Correctness

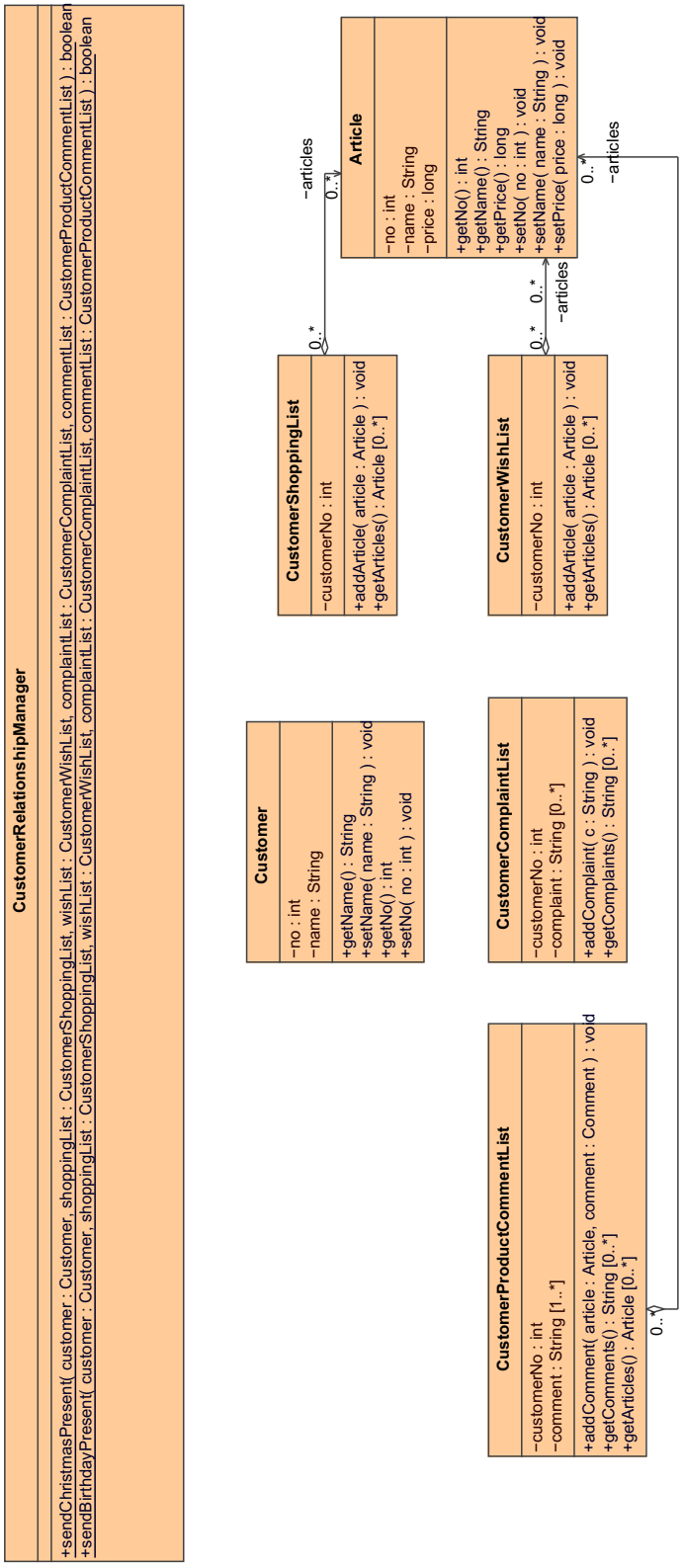


Figure D.7: Example UML model smell *Long Parameter List*

D.6 multiple definitions of classes with equal names

DESCRIPTION This smell occurs if in a single model more than one class has the same name. The different classes with the same name may be defined in the same diagram or in different diagrams. It is essential that equally named classes are owned by distinct packages (namespaces) in order to respect the uniqueness of qualified names in UML. Equally named classes could lead to misunderstandings of the modeled aspects. Furthermore, this smell will cause problems during code generation in a model-driven process. [97]

EXAMPLE Figure D.8 shows class *Customer* in package *Rental* owning attributes *name* and *address* as well as associating its rented vehicle. Furthermore, there is another class *Customer* in package *Accounting* modeling the aspect that a *Customer* holds at least one account. This situation reflects a typical case of redundant modeling that can be made more concrete by smell *Multiple Definitions of Classes with Equal Names*.

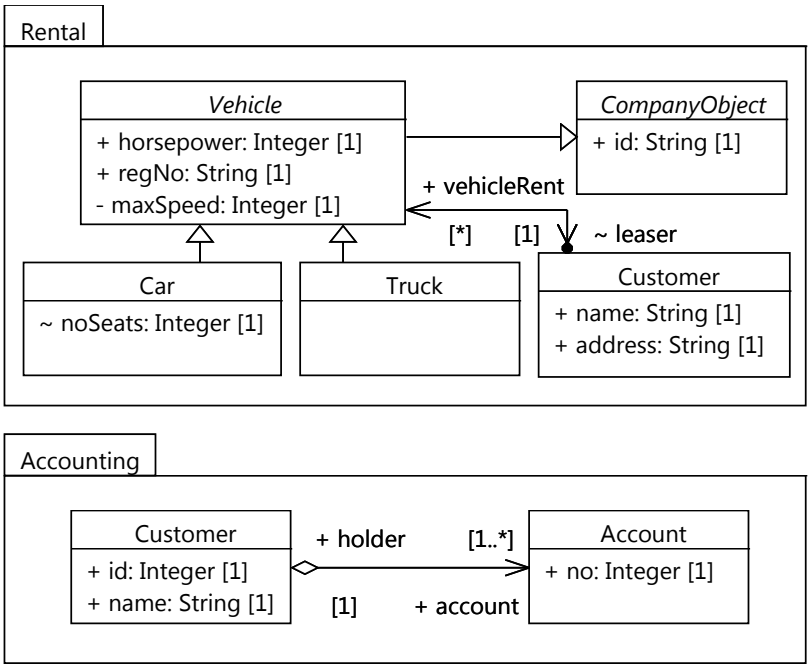


Figure D.8: Example UML model smell *Multiple Definitions of Classes with Equal Names*

DETECTION This smell can be detected by matching a corresponding (anti-) pattern based on the abstract syntax of UML. Figure D.9 shows the Henshin rule that specifies this pattern-based smell. This pattern rule defines two UML `Classes` (named *class_1* and *class_2*) which have the same name (specified by the internal

rule parameter *classname*). As mentioned above, it is essential that these classes are owned by distinct packages. This additional constraint is specified by PAC *DifferentPackages*. In summary, the Henshin pattern rule in Figure D.9 specifies two equally named classes that are owned by different packages.

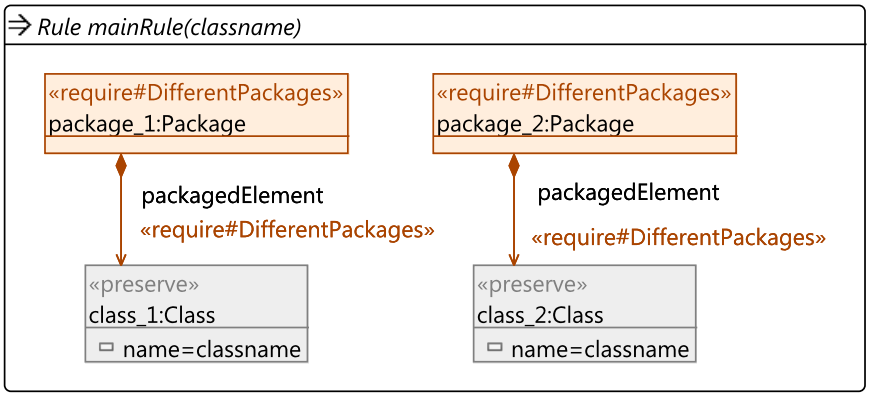


Figure D.9: Henshin pattern rule specification of UML model smell *Multiple Definitions of Classes with Equal Names*

USABLE UML MODEL REFACTORINGS *Rename Class* for renaming one involved class.

AFFECTED QUALITY CHARACTERISTICS AND GOALS Equally named classes are redundancy at its best. Redundancy → Correctness, Consistency, Comprehensibility, Changeability

D.7 primitive obsession

DESCRIPTION In this smell, primitive data types like `Boolean` and `Integer` are used to encode data that would be better modeled as a separate class. Mostly this is done since developers are reluctant to use small classes for small tasks. Here, the use of even small classes might be a better choice to increase the understandability of the model. Furthermore, it is against the object-oriented paradigm to treat domain objects, even small ones, as primitive type instead of a class modeling its constituent parts. [11]

EXAMPLE In Figure D.10 this smell occurs twice. First, there are four constant attributes which would be better modeled as enumeration. Furthermore, there are many attributes of primitive type `Integer` which partially adhere to each other and technically present a point respectively coordinate.

DETECTION This smell can hardly be detected. An indicator for this smell may be a high value of metric *Number of Constant At-*

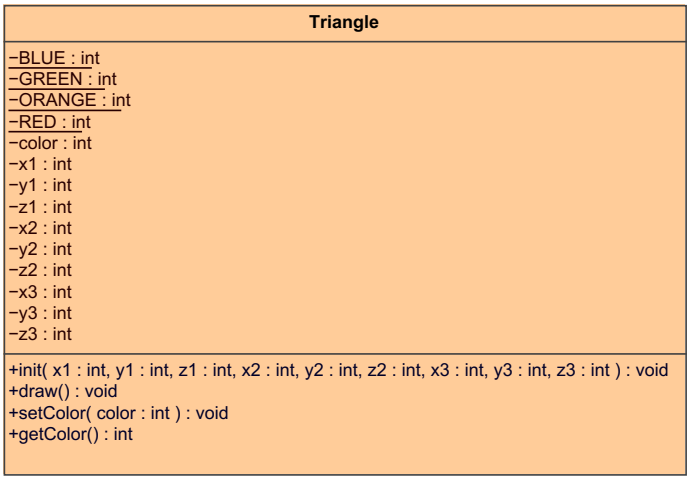


Figure D.10: Example UML model smell *Primitive Obsession*

tributes since this might be a hint for the misuse of an enumeration. Also, a high value of metric *Number of Primitive typed Attributes* might indicate the existence of this smell [104]. Figure D.11 shows the implemented specifications of both metrics by using corresponding Henshin pattern rules. The rule on the top of Figure D.11 defines the abstract syntax pattern of an attribute of a given `Class` (named *context*) that has some `PrimitiveType` as type. In EMF Refactor, the number of occurrences of this pattern w.r.t. a contextual class is counted and represents the value of the corresponding metric. The bottom of Figure D.11 shows the Henshin pattern rule of metric *Number of constant Attributes*. It defines the pattern of an attribute of a given `Class` (named *context*) whose meta attribute *isReadOnly* is set to *true*, i.e this class attribute has a constant value. Again, we can count the occurrences of this pattern with respect to a contextual class. In summary, this smell can be implemented in two metric-based variants (whereas the corresponding metrics can be implemented by patterns!).

USABLE UML MODEL REFACTORINGS *Extract Class* or *Introduce Parameter Object* for extracting information into a new class.

AFFECTED QUALITY CHARACTERISTICS AND GOALS Using primitive types instead of small classes might show problems in modular design. Furthermore, this might be imprecise and might reflect semantic misunderstandings. Cohesion/Modular Design, Semantic Adequacy, Precision, Simplicity → Correctness, Confinement, Comprehensibility

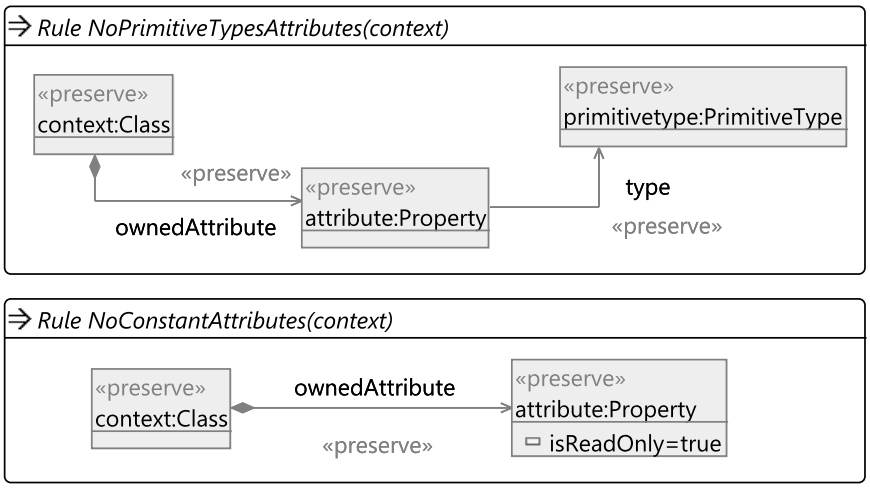


Figure D.11: Henshin pattern rule specifications of UML model metrics *Number of primitive typed Attributes* and *Number of constant Attributes* used for specifying UML smell *Primitive Obsession*

D.8 specialization aggregation

DESCRIPTION The association is a specialization of another association. This means, that there is a generalization relation between the two involved associations. People are often confused by the semantics of specialized associations. The suggestion is therefore to model any restrictions on the parent association using constraints. [119]

EXAMPLE Figure D.12 shows class *Journey* that is subclassed by class *AirJourney*. Also there is a similar class inheritance hierarchy including classes *Route* and *AirRoute*. Furthermore, there is an association between both subclasses *Journey* and *Route*. This association is also specialized by a corresponding association. In fact, this association hierarchy might be confusing.

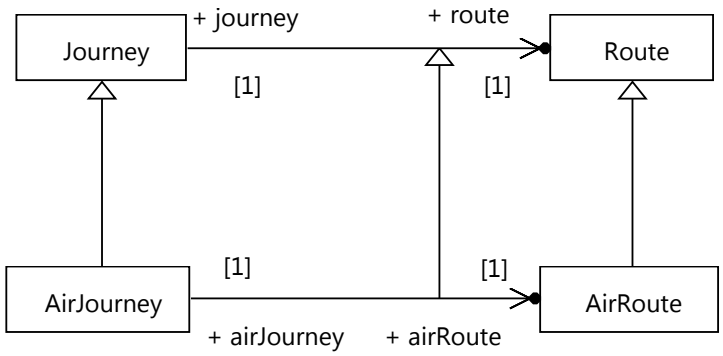


Figure D.12: Example UML model smell *Specialization Aggregation*

DETECTION This smell can be detected by matching a corresponding (anti-) pattern based on the abstract syntax of UML. Figure D.13 shows the Henshin rule that specifies this pattern-based smell. This pattern rule defines two UML `Associations` (named *assoc_1* and *assoc_2*) which are related by a `Generalization` relationship (specified by PAC *HasGeneralization*). In summary, the Henshin pattern rule in Figure D.13 specifies an association that specializes another one in the inheritance hierarchy of the corresponding classes.

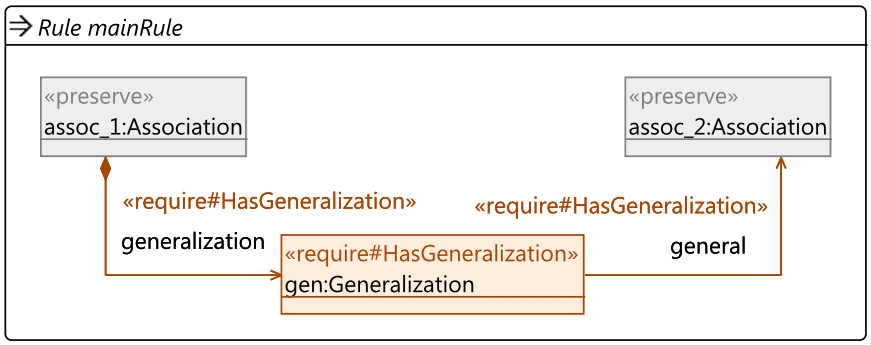


Figure D.13: Henshin pattern rule specification of UML model smell *Specialization Aggregation*

USABLE UML MODEL REFACTORINGS No existing model refactoring can be used to eliminate this smell. Either it has to be developed, or the smell has to be eliminated directly, for example by restructuring the model considering this specific aspect.

AFFECTED QUALITY CHARACTERISTICS AND GOALS Specialized associations are hard to understand and might represent redundant modeling since involved classes can be already specializations. Simplicity, Redundancy → Comprehensibility

D.9 speculative generality

DESCRIPTION Often, developer model special cases but it is not essential to hold this information in the model. This is done since the developer intends to use this specific information sometime. In such cases this additional elements should be excluded to avoid an increase in the complexity of the model. Not required information might lead to an ambiguous model. This kind of smell includes: abstract classes that are not doing much, methods with unused parameters, methods named with odd abstract names [11]. Zhang et al. [161] present a more precise pattern-based definition of this model smell. First, the involved element has to be an abstract class or an interface. The smell occurs if

this element has not been inherited/implemented, or is only inherited/implemented by one single class/interface.

EXAMPLE Figure D.14 shows two abstract classes *AbstractLong* and *AbstractDouble* that are only inherited by one single concrete class each. These classes might be modeled to address future concerns. But in fact, they are non-essential and shall be removed.

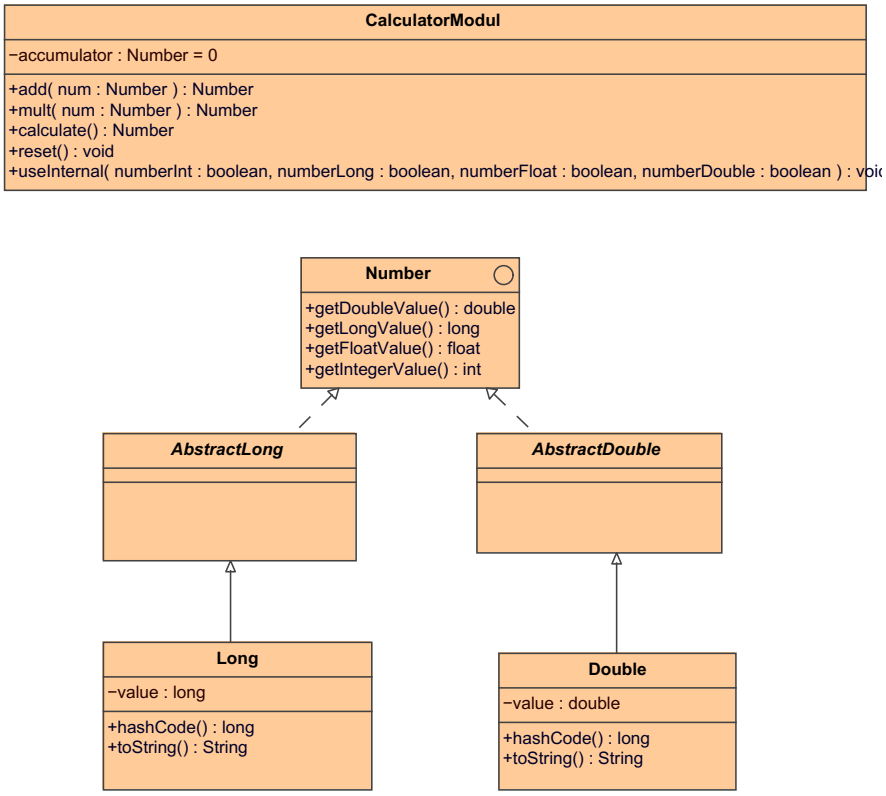


Figure D.14: Example UML model smell *Speculative Generality*

DETECTION This smell can be detected by matching a corresponding pattern based on the abstract syntax of UML in the case of abstract classes and interfaces. Furthermore, it can be checked whether corresponding metrics like *Number of direct subclasses* and *Number of implementing classes* are evaluated to zero respectively one. Figure D.15 shows the Henshin rules that specify this pattern-based smell. The rule on the top of Figure D.15 defines the abstract syntax pattern of an abstract UML `Class` (*abstractclass*; the meta attribute *isAbstract* is set to *true*) that has one concrete subclass *subclass_1*; the meta attribute *isAbstract* is set to *false*). Furthermore, this abstract class must not have any further concrete subclasses (specified by NAC *HasNoFurtherConcreteSubclass*). The rule on the bottom of Figure D.15 de-

fines an `Interface` that is realized by a `Class` (*implementing-class*). Furthermore, NAC *HasNoFurtherImplementation* forbids the presence of another `InterfaceRealization`. In summary, this smell can be implemented in two pattern-based variants.

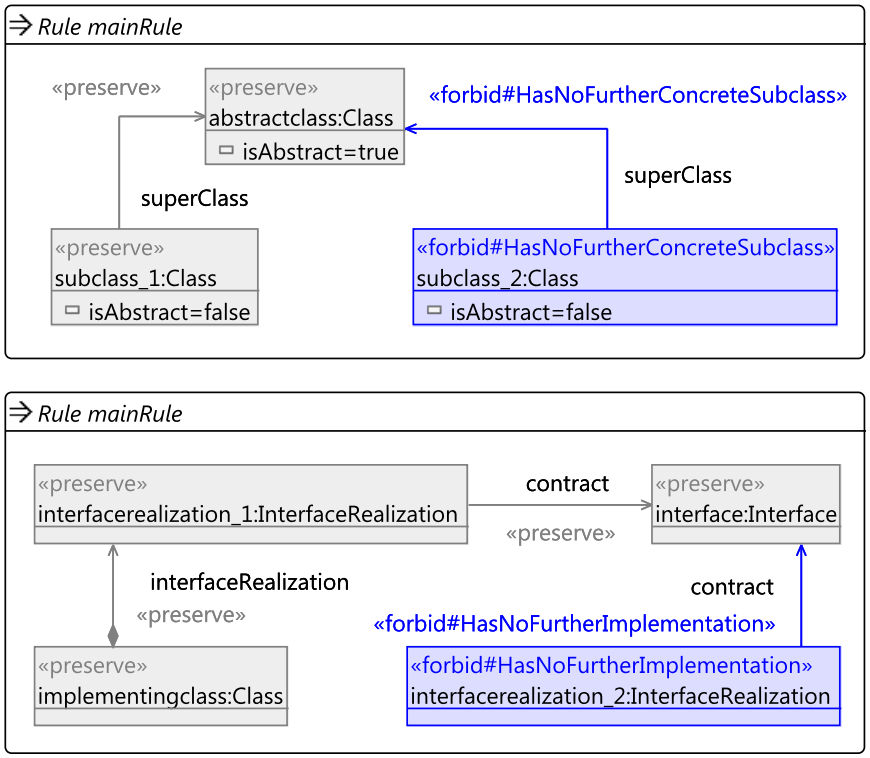


Figure D.15: Henshin pattern rule specifications of UML model smell *Speculative Generality*

USABLE UML MODEL REFACTORINGS *Inline Class*, *Remove Superclass* for removing needless classes. *Remove Parameter* for removing needless parameters. *Rename Operation* for giving an operation a more concrete name.

AFFECTED QUALITY CHARACTERISTICS AND GOALS This smell may lead to more complex models that might be harder to understand. Simplicity, Presentation → Comprehensibility, Confinement

D.10 unnamed element

DESCRIPTION The model element, i.e., package, class, interface, data type, attribute, operation, or parameter, has no name. This smell summarizes corresponding smells such as *Unnamed Class* and *Unnamed Attribute*. According to the UML specification this is no misuse, i.e., the model is still valid. But on the one hand an

unnamed element could lead to misunderstandings of the modeled aspect, on the other hand unnamed elements will cause problems during code generation in a model-driven process. However, a model element should be given a descriptive name that reflects its purpose. [83]

EXAMPLE Figure D.16 shows classes *SoccerClub*, *Date*, and *Person* related by several associations. Among others, there is an association between classes *SoccerClub* and *Person* but without any names, neither an association name nor corresponding role names. Here, it is very hard to understand the meaning of the association. Does it mean players, trainers, or even board members?

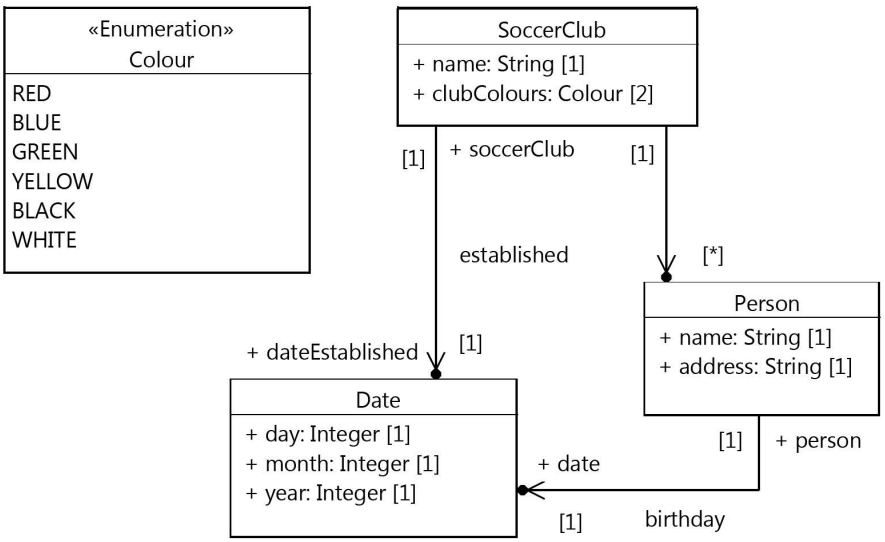


Figure D.16: Example UML model smell *Unnamed Element*

DETECTION This smell can be detected by matching a corresponding (anti-) patterns based on the abstract syntax of UML. Figure D.17 shows the Henshin rules that specify this pattern-based smell. Each rule specifies a corresponding meta model element (`Property`, `Class`, `DataType`, `Interface`, `Operation`, `Package`, and `Parameter`) whose meta attribute *name* is not set (specified by `name=""`). The rule concerning an unnamed `Property` has an additional PAC *IsUnnamedAttribute* that ensures that the `Property` is in fact an attribute of a `Classifier` (i.e., `Class` or `Interface`). The rule concerning an unnamed `Parameter` ensures that only unnamed input parameters are detected (specified by rule attribute *dir* and attribute condition `dir!=return`). In summary, this smell can be implemented in seven pattern-based variants.

USABLE UML MODEL REFACTORINGS *Rename Class*, *Rename Attribute*, etc. for giving the model element a proper name.

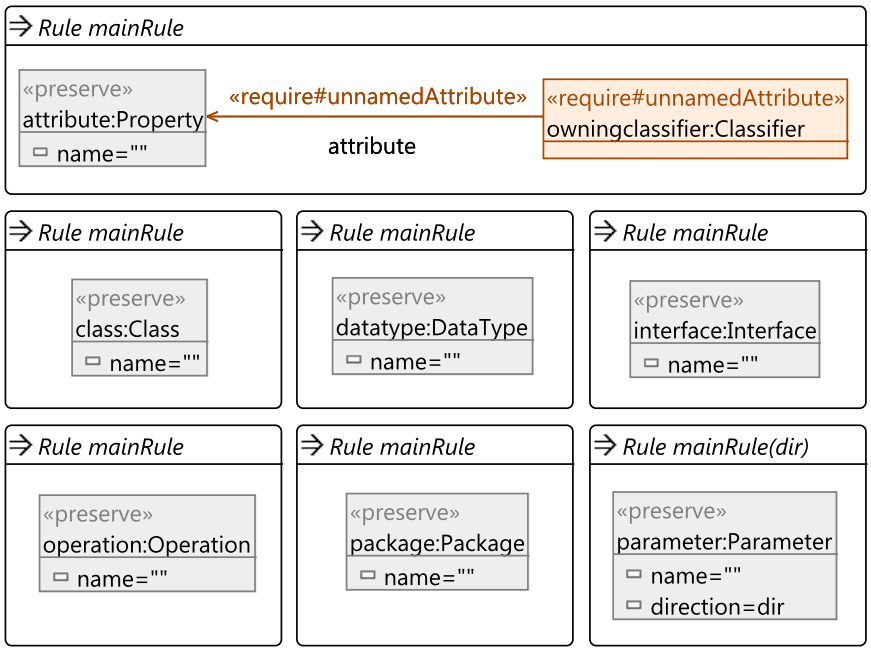


Figure D.17: Henshin pattern rule specifications of UML model smell *Unnamed Element*

AFFECTED QUALITY CHARACTERISTICS AND GOALS A model element without an appropriate name may reflect a real world aspect imprecise and incorrect. Furthermore, they might be harder to understand. Simplicity, Conformity, Precision → Consistency, Comprehensibility, Correctness, Completeness

D.11 unused element

DESCRIPTION An unused model element is useless and indicates incorrect modeling. Either the element represents a valid domain object, i.e., there are missing relationships to further objects, or the modeler wanted to delete the element from the model but removed it only from the diagram. For example, an unused class has no child classes, dependencies, or associations and it is not used as parameter or property type. [133]

EXAMPLE An example is given in the description of this smell (*Unused Class*).

DETECTION This smell can be detected by matching corresponding patterns based on the abstract syntax of UML. This patterns have to be formulated in a way that the contextual element (class, for example) is not allowed to have any specific relationships to other elements. However, these patterns have to be defined specific to the considered contextual element type.

Another way to define this smell is to determine specific metrics and to check whether these metrics are evaluated to zero each. For example, checking specific smell *Unused Class* requires model metrics *Number of direct children*, *Total number of dependencies*, *Number of associated classes*, *Number of times the class is externally used as attribute type*, and *Number of times the class is externally used as parameter type*. Figure D.18 shows altogether four Henshin rules that specify this pattern-based smell. The top rule specifies the abstract pattern of an unused UML `Class`. It consists of five NACs which must hold altogether at the same time:

- The class must not have any subclasses (NAC *hasNoChildClass*).
- The class must not have any superclasses (NAC *hasNoParentClass*).
- The class must not be used as attribute or parameter type (NAC *isNoType*).
- The class must not implement any interfaces (NAC *doesNotRealizeAnInterface*).
- The class must not use any interfaces (NAC *doesNotUseAnInterface*).

The lower left rule specifies the abstract pattern of an unused UML `Enumeration`. NAC *isNotUsedAsType* specifies that this enumeration is not used as type of a `TypedElement` (attribute, for example). The rule in the middle specifies the abstract pattern of an unused UML `Interface`. It consists of three NACs which must hold altogether at the same time:

- The interface must not be implemented by a class (NAC *noInterfaceRealization*).
- The interface must not be used by a class (NAC *noInterfaceUsage*).
- The interface must not have any subinterfaces (NAC *noInterfaceSpecialization*).

The lower right rule specifies the abstract pattern of an unused UML `Package`. NAC *noElements* specifies that this package does not have any packaged elements in it. In summary, this smell can be implemented in four pattern-based variants.

USABLE UML MODEL REFACTORINGS Here, there is no refactoring needed. Just remove the element from the model or continue modeling missing relationships.

AFFECTED QUALITY CHARACTERISTICS AND GOALS A model element that is not used may reflect an imprecise modeling. Precision → Correctness, Confinement

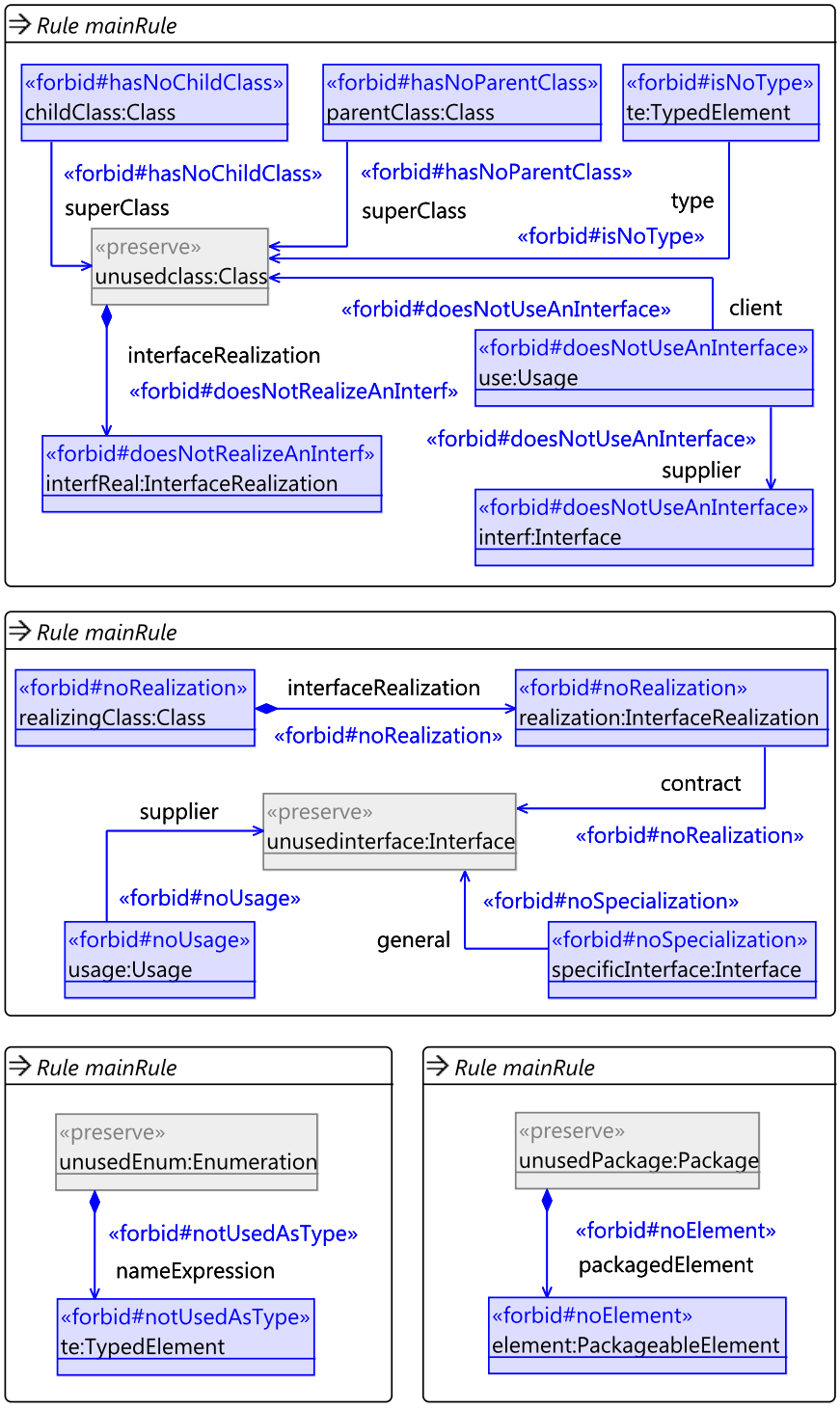


Figure D.18: Henshin pattern rule specifications of UML model smell *Unused Element*

E

SPECIFICATIONS OF UML CLASS MODEL REFACTORINGS

In this appendix we describe selected refactorings for UML class models found in literature. For each model refactoring a short description, the contextual meta model element type for invoking the refactoring, and the input parameters of the refactoring are given. Furthermore, we present preconditions that have to be checked, either before or after parameter input by the refactoring user, as well as postconditions that specify the behavior preservation of the refactoring. We then specify the transformation that has to be performed after the precondition checks have passed. Finally, an example completes each model refactoring description.

E.1 add parameter

DESCRIPTION An operation needs more information from its callers. Therefore, this refactoring adds a parameter to an operation. [30, 150]

EXAMPLE Figure E.1 shows an operation *setVisible* in class *Square* that ought to get an additional parameter *vis* of type *Boolean*. This parameter has to be added.

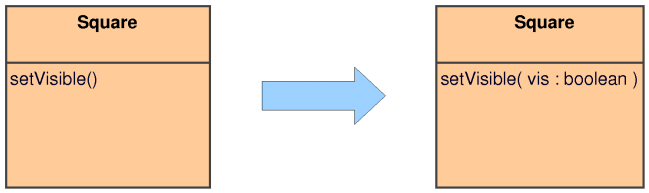


Figure E.1: Example UML model refactoring *Add Parameter*

CONTEXTUAL ELEMENT `Operation`

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked.

REFACTORING PARAMETERS (1) `parameterName` - Name of the new parameter. (2) `parameterType` - Type of the new parameter.

FINAL PRECONDITIONS CHECK (1) The contextual operation does not already have a parameter named `parameterName`. (2) There is no operation with the same name as the contextual operation and with the same parameter list (equal parameter names and types) as the contextual operation including a new parameter named `parameterName` of type `parameterType` in the class and its inheritance hierarchy owning the contextual operation.

MODEL TRANSFORMATION Add a new parameter named `parameterName` of type `parameterType` to the parameter list of the contextual operation.

POSTCONDITIONS There is a new parameter named `parameterName` of type `parameterType` in the parameter list of the contextual operation.

E.2 create associated class

DESCRIPTION This refactoring creates an empty class and connects it with a new association to the source class from where it is extracted. The multiplicity of the new association is 1 at both ends. Usually, refactorings *Move Property* and *Move Operation* are the next steps after this refactoring. [107, 150]

EXAMPLE Figure E.2 shows class *Bill* storing name, street, and residence of a customer. Improve the design with a special class *Address* and use refactorings *Move Property* and *Move Operation* afterwards to fill class *Address*. Please note that in this example we used a variation of this refactoring that creates private association ends as well as their corresponding getter and setter operations.

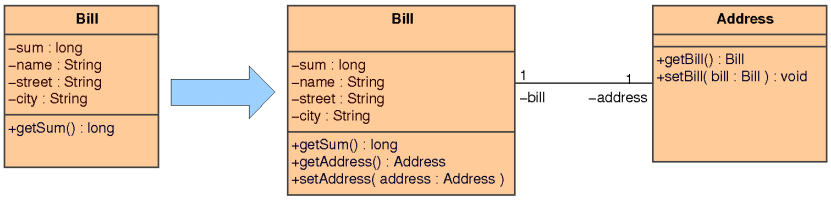


Figure E.2: Example UML model refactoring *Create Associated Class*

CONTEXTUAL ELEMENT `Class`

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked.

REFACTORING PARAMETERS (1) `className` - Name of the new associated class. (2) `namespaceName` - Namespace of the new associated class given by a qualified name. (3) `associationName`

- Name of the new association. (4) `endName1` and (5) `endName2` - Names of the association ends of the new association.

FINAL PRECONDITIONS CHECK (1) There does not already exist a classifier named `className` in the namespace named `namespaceName`. (2) The contextual class does not already have an attribute named `endName1`.

MODEL TRANSFORMATION (1) Create a new class named `className` in the namespace named `namespaceName` with default visibility. (2) Insert an association named `associationName` and multiplicity $1:1$ between the contextual class and the newly created class. (3) Name the new association end typed by the newly created class `endName1`. (4) Name the new association end typed by the conceptual class `endName2`.

POSTCONDITIONS (1) There is a new class named `className` in the namespace named `namespaceName` with default visibility. (2) There is an association named `associationName` and multiplicity $1:1$ between the contextual class and the newly created class. (3) The name the new association end typed by the newly created class is `endName1`. (4) The name the new association end typed by the conceptual class is `endName2`.

E.3 create subclass

DESCRIPTION A class has features (attributes or operations) that are not used in all instances. This refactoring creates a subclass for that subset of features. However, the new subclass has no features. Usually, refactorings *Push Down Property* and *Push Down Operation* are the next steps after this refactoring. [150]

EXAMPLE Figure E.3 shows class *File* modeling all aspects a file can have. Some aspects are just important for directories. Create a subclass and move these features down to this kind of file via refactorings *Push Down Property* and *Push Down Operation*.

CONTEXTUAL ELEMENT `Class`

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked.

REFACTORING PARAMETERS (1) `className` - Name of the new subclass. (2) `namespaceName` - Namespace of the new subclass given by a qualified name.

FINAL PRECONDITIONS CHECK There does not already exist a classifier named `className` in the namespace named `namespaceName`.

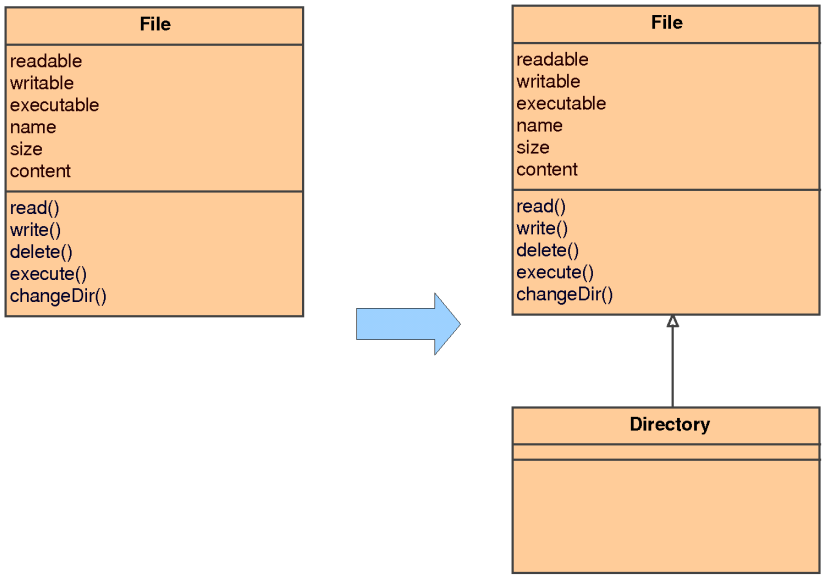


Figure E.3: Example UML model refactoring *Create Subclass*

MODEL TRANSFORMATION (1) Create a new class named `className` in the namespace named `namespaceName` with default visibility. (2) Insert an inheritance relation from the newly created class to the contextual class.

POSTCONDITIONS (1) There is a new class named `className` in the namespace named `namespaceName` with default visibility. (2) There is an inheritance relation from the newly created class to the contextual class.

E.4 create superclass

DESCRIPTION This refactoring can be applied when there are at least two classes with similar features (attributes or operations). The refactoring creates a superclass for this set of classes and is normally followed by refactorings *Pull Up Property* and *Pull Up Operation*. So, the refactoring helps to reduce the duplicate common features spread throughout different classes. [141, 150, 107, 160]

EXAMPLE Figure E.4 shows classes *Car* and *Bike* which have common attributes and operations. Create a new superclass and move these common features to this new class by refactorings *Pull Up Property* and *Pull Up Operation*.

CONTEXTUAL ELEMENTS Set of `Classes`

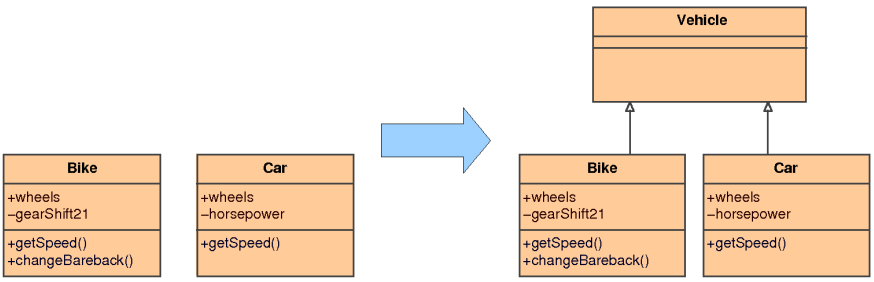


Figure E.4: Example UML model refactoring *Create Superclass*

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked.

REFACTORING PARAMETERS (1) `className` - Name of the new superclass. (2) `namespaceName` - Namespace of the new superclass given by a qualified name.

FINAL PRECONDITIONS CHECK There does not already exist a classifier named `className` in the namespace named `namespaceName`. If so, it must be an empty class, i.e., it has no attributes, no operations, no superclasses, and no inner classes; it is not associated to other classes; it does not implement any interfaces, and is not referred to as type of an attribute, operation or parameter.

MODEL TRANSFORMATION (1) Create a new class named `className` in the namespace named `namespaceName` with default visibility if such a class does not exist yet. (2) Insert inheritance relations from each contextual class to the newly created class respectively existing class.

POSTCONDITIONS (1) There is a new class named `className` in the namespace named `namespaceName` with default visibility. (2) There is an inheritance relation from each contextual class to the newly created class.

E.5 extract associated class

DESCRIPTION This refactoring extracts interrelated features (attributes and operations) from a class to a new separated class. [107, 150]

EXAMPLE Figure E.5 shows class *Bill* that contains the attributes of the customer's address. Extract these attributes in an own class *Address*.

CONTEXTUAL ELEMENT `Class`

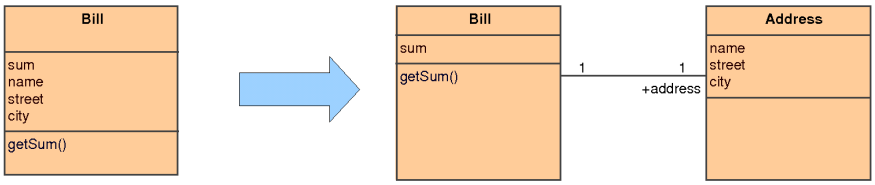


Figure E.5: Example UML model refactoring *Extract Associated Class*

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked. However, the initial preconditions of the involved refactorings have to be checked properly.

REFACTORING PARAMETERS Each parameter of refactoring *Create Associated Class*. Additionally, a list of attributes and operations which have to be moved to the new associated class.

FINAL PRECONDITIONS CHECK The contextual class owns each attribute and operation of the corresponding input lists. Additionally, the final preconditions of the involved refactorings have to be checked properly.

MODEL TRANSFORMATION (1) Use refactoring *Create Associated Class* on the contextual class with the given parameters. (2) Use refactoring *Move Property* on each attribute of the appropriate parameter list with the corresponding parameter. (3) Use refactoring *Move Operation* on each operation of the appropriate parameter list with the corresponding parameter.

POSTCONDITIONS In each step the postconditions of the used refactorings have to be checked. No additional postconditions are required.

E.6 extract subclass

DESCRIPTION There are features (attributes and operations) in a class required for a special case only. This refactoring extracts a subclass containing this features. [150]

EXAMPLE In Figure E.6 the association end *container* of class *File* is just relevant for directories. Create a subclass *Directory* and push down the association end *container*.

CONTEXTUAL ELEMENT `Class`

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked. However, the initial preconditions of the involved refactorings have to be checked properly.

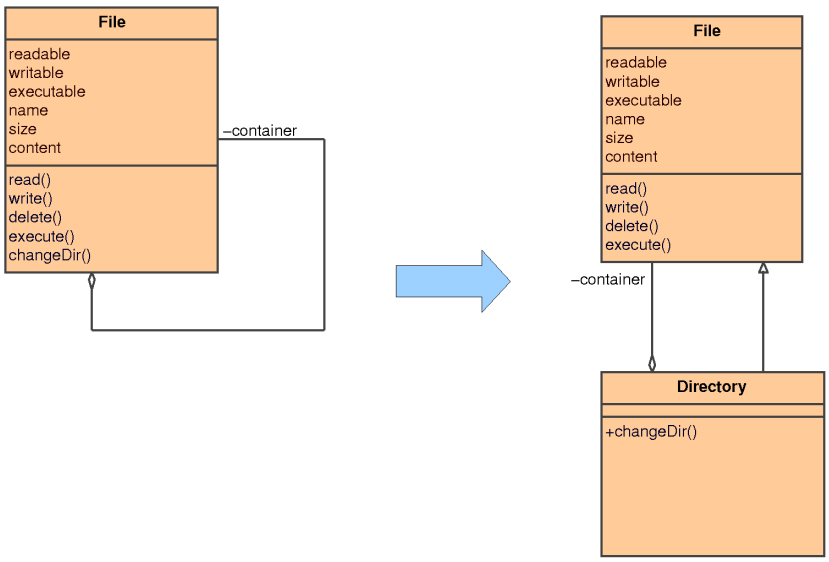


Figure E.6: Example UML model refactoring *Extract Sublass*

REFACTORING PARAMETERS Each parameter of refactoring *Create Associated Class*. Additionally, a list of attributes and operations which have to be pushed to the new subclass.

FINAL PRECONDITIONS CHECK The contextual class owns each attribute and operation of the corresponding input lists. Additionally, the final preconditions of the involved refactorings have to be checked properly.

MODEL TRANSFORMATION (1) Use refactoring *Create Subclass* on the contextual class with the given parameters. (2) Use refactoring *Push Down Property* on each attribute of the appropriate parameter list. (3) Use refactoring *Push Down Operation* on each operation of the appropriate parameter list.

POSTCONDITIONS In each step the postconditions of the used refactorings have to be checked. No additional postconditions are required.

E.7 extract superclass

DESCRIPTION There are two or more classes with similar features. This refactoring creates a new superclass and moves the common features to the superclass. The refactoring helps to reduce redundancy by assembling common features spread throughout different classes. [141, 106, 150, 107, 160]

EXAMPLE Figure E.7 shows classes *Bike* and *Car* that have common attributes and operations. Extract these common features to a new superclass *Vehicle*.

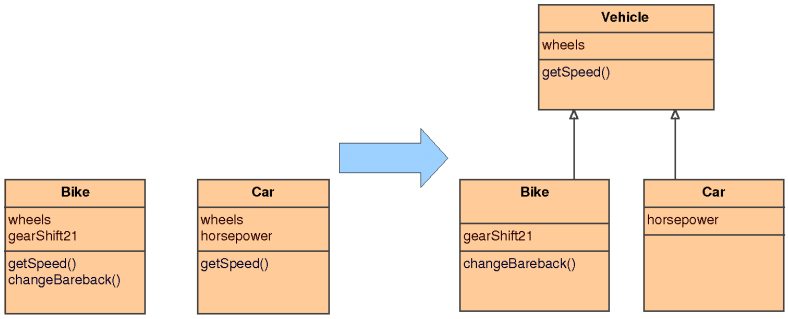


Figure E.7: Example UML model refactoring *Extract Superlass*

CONTEXTUAL ELEMENTS Set of `Classes`

INITIAL PRECONDITIONS CHECK The contextual classes have similar features, i.e., attributes with the same name, type, visibility and multiplicity, or operations with the same name, visibility and parameter list. Additionally, the initial preconditions of the involved refactorings have to be checked properly.

REFACTORING PARAMETERS Each parameter of refactoring *Create Superclass*. Additionally, a list of attributes and operations which have to be pushed to the new subclass is taken from one contextual class.

FINAL PRECONDITIONS CHECK There are no final preconditions that need to be checked. However, the final preconditions of the involved refactorings have to be checked properly.

MODEL TRANSFORMATION (1) Use refactoring *Create Superclass* on the contextual classes with the given parameters. (2) Use refactoring *Pull Up Property* on each attribute of the appropriate parameter list with the corresponding parameter. (3) Use refactoring *Pull Up Operation* on each operation of the appropriate parameter list with the corresponding parameter.

POSTCONDITIONS In each step the postconditions of the used refactorings have to be checked. No additional postconditions are required.

E.8 inline class

DESCRIPTION There are two classes connected by a 1:1 association. One of them has no further use. This refactoring merges these classes. [150, 93]

EXAMPLE In Figure E.8 class *ZipCode* only contains the zip code number and is only referenced from class *Address*. Thus, both classes can be merged.

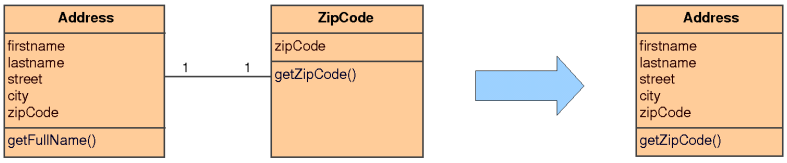


Figure E.8: Example UML model refactoring *Inline Class*

CONTEXTUAL ELEMENT `Class`

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked. However, the initial preconditions of the involved refactorings have to be checked properly.

REFACTORING PARAMETERS Each parameter of refactoring *Remove Empty Associated Class*. Additionally, a list of attributes and operations which have to be moved to the associated class.

FINAL PRECONDITIONS CHECK The contextual class owns each attribute and operation of the corresponding input lists. Additionally, the final preconditions of the involved refactorings have to be checked properly.

MODEL TRANSFORMATION (1) Use refactoring *Move Property* on each attribute of the appropriate parameter list with the corresponding parameter. (2) Use refactoring *Move Operation* on each operation of the appropriate parameter list with the corresponding parameter. (3) Use refactoring *Remove Empty Associated Class* on the contextual class.

POSTCONDITIONS In each step the postconditions of the used refactorings have to be checked. No additional postconditions are required.

E.9 introduce parameter object

DESCRIPTION There is a group of parameters that naturally go together. This refactoring replaces a list of parameters with one object. This parameter object is created for that purpose. [11, 131, 161, 104]

EXAMPLE In Figure E.9 a date range is used in several operations. Use refactoring *Introduce Parameter Object* to build class *DateRange*.

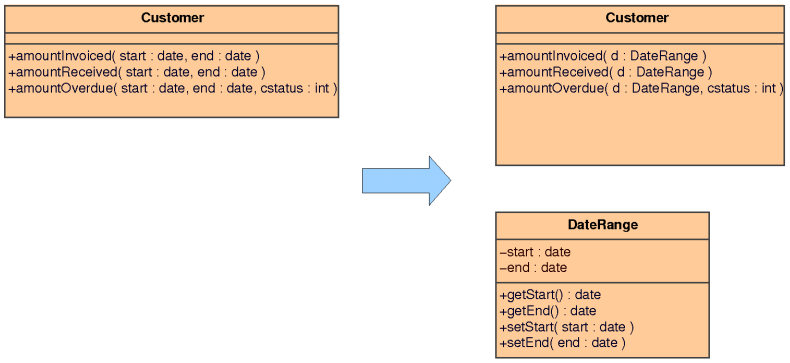


Figure E.9: Example UML model refactoring *Introduce Parameter Object*

CONTEXTUAL ELEMENTS List of `Parameters`

INITIAL PRECONDITIONS CHECK All contextual parameters belong to the same operation.

REFACTORING PARAMETERS (1) `className` - Name of the new parameterclass. (2) `namespaceName` - Namespace of the new parameterclass given by a qualified name.

FINAL PRECONDITIONS CHECK There does not already exist a classifier named `className` in the namespace named `namespaceName`.

MODEL TRANSFORMATION (1) Create a new class named `className` in the namespace named `namespaceName` with default visibility. (2) Create for each contextual parameter a private attribute with getter and setter operations. (3) Replace the parameter list in all operations of the class owning the operation with the contextual parameters with a new parameter with type of the parameter class. Use refactorings *Add Parameter* and *Remove Parameter* for this purpose.

POSTCONDITIONS (1) There is a new class named `className` in the namespace named `namespaceName` with default visibility. (2) For each contextual parameter there is a private attribute with getter and setter operations. (3) There is a new parameter with type of the parameter class in all operations of the class owning the operation with the contextual parameters.

E.10 move operation

DESCRIPTION This refactoring moves an operation of a class to an associated class. It is often applied when some class has too much behavior or when classes collaborate too much. In most cases, the visibility of the operation should be the same as before. In

some cases, when the operation is *private* or it is moved between classes belonging to different packages, this is not enough. [107, 150]

EXAMPLE In Figure E.10 operation *clearScreen* is better placed in class *Display* and therefore should be moved.

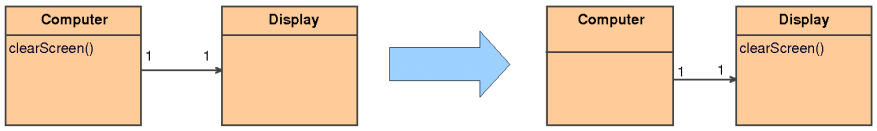


Figure E.10: Example UML model refactoring *Move Operation*

CONTEXTUAL ELEMENT `Operation`

INITIAL PRECONDITIONS CHECK The owning class of the contextual operation has an association with multiplicities 1-1 to another class. This association must be navigable in both directions.

REFACTORING PARAMETERS `className` - Name of the target class.

FINAL PRECONDITIONS CHECK (1) The owning class of the contextual operation has an association with multiplicities 1-1 to class named `className`. (2) There is no operation with the same name as the contextual operation and with the same parameter list (equal parameter names and types) as the contextual operation in the class and its inheritance hierarchy named `className`.

MODEL TRANSFORMATION Move the contextual operation to the associated class named `className`. If needed change the visibility of the contextual operation.

POSTCONDITIONS (1) The contextual operation does not exist anymore in its source class. (2) The contextual operation exists in the target class. (3) The contextual operation is still visible for the source class in the target class.

E.11 move property

DESCRIPTION A property (attribute) is better placed in another class which is associated to this class. This refactoring moves this property to the associated class. In most cases, the visibility of the property should be the same as before. In some cases, when the property is *private* or it is moved between classes belonging to different packages, this is not enough [107, 150, 93]

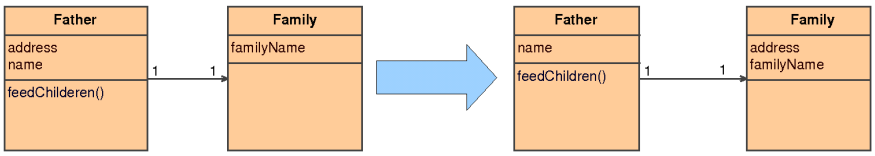


Figure E.11: Example UML model refactoring *Move Property*

EXAMPLE Figure E.11 shows attribute *address* that is better placed in class *Family* and therefore should be moved.

CONTEXTUAL ELEMENT `Property`

INITIAL PRECONDITIONS CHECK The owning class of the contextual property has an association with multiplicities 1-1 to another class. This association must be navigable in both directions.

REFACTORING PARAMETERS `className` - Name of the target class.

FINAL PRECONDITIONS CHECK (1) The owning class of the contextual property has an association with multiplicities 1-1 to class named `className`. (2) There is no attribute with the same name as the contextual property in the class named `className`. (3) There is no attribute with the same name as the contextual property in the inheritance hierarchy of the class named `className`.

MODEL TRANSFORMATION Move the contextual property to the associated class named `className`. If needed change the visibility of the contextual property.

POSTCONDITIONS (1) The contextual property does not exist anymore in its source class. (2) The contextual property exists in the target class. (3) The contextual property is still visible for the source class in the target class.

E.12 pull up operation

DESCRIPTION This refactoring pulls an operation of a class to its superclass. Usually it is used simultaneously on several classes which inherit from the same superclass. The aim of this refactoring is often to extract identical operations. In some cases, when the property is *private* or it is moved between classes belonging to different packages, this is not enough. [137, 107, 150]

EXAMPLE In Figure E.12 on page 241 the same operation *switchOn* exists in class *DesktopPC* and *LaptopPC*. Move it to superclass *Computer*.

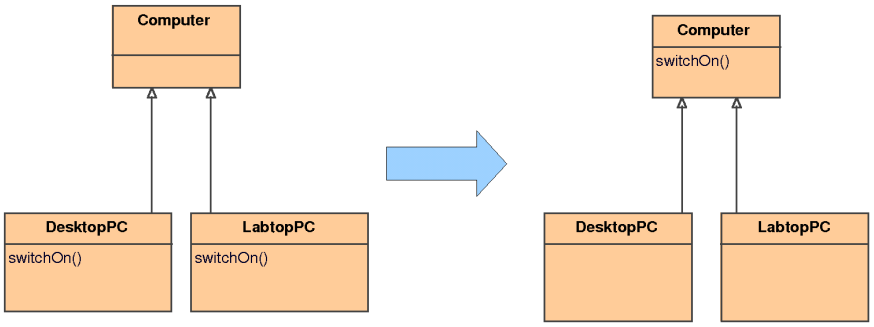


Figure E.12: Example UML model refactoring *Pull Up Operation*

CONTEXTUAL ELEMENT `Operation`

INITIAL PRECONDITIONS CHECK The contextual operation is owned by a class that has a superclass.

REFACTORING PARAMETERS `className` - Name of the superclass the contextual operation has to be pulled up to.

FINAL PRECONDITIONS CHECK (1) The owning class of the contextual operation has a superclass named `className`. (2) The superclass does not own an operation with the same name as the contextual operation and with the same parameter list (equal parameter names and types) as the contextual operation. (3) There is no operation with the same name as the contextual operation and with the same parameter list (equal parameter names and types) as the contextual operation in the inheritance hierarchy of the superclass that is visible to it. (4) Each subclass of this superclass owns an operation with the same name and parameter list as the contextual operation.

MODEL TRANSFORMATION (1) Move the contextual operation to the class named `className`. (2) Remove each operation with the same name and parameter list as the contextual operation from the subclasses of the class named `className`. (3) If needed change the visibility of the contextual operation.

POSTCONDITIONS (1) The contextual operation is owned by the superclass. (2) The subclasses of the superclass do not own an operation with the same name parameter list as the contextual operation. (3) The contextual operation is still visible in all subclasses of the superclass.

E.13 pull up property

DESCRIPTION This refactoring removes one property (attribute) from a class or a set of classes and inserts it to one of its superclasses.

In most cases, the visibility of the property should be the same as before. In some cases, when the property is *private* or it is moved between classes belonging to different packages, this is not enough. [14, 107, 30, 150]

EXAMPLE In Figure E.13 on page 242 the same attribute *heatEmission* exists in class *DesktopPC* and *LaptopPC*. Move it to superclass *Computer*. It must be still visible in the subclasses. Therefore, change its visibility to *private* or *protected*.

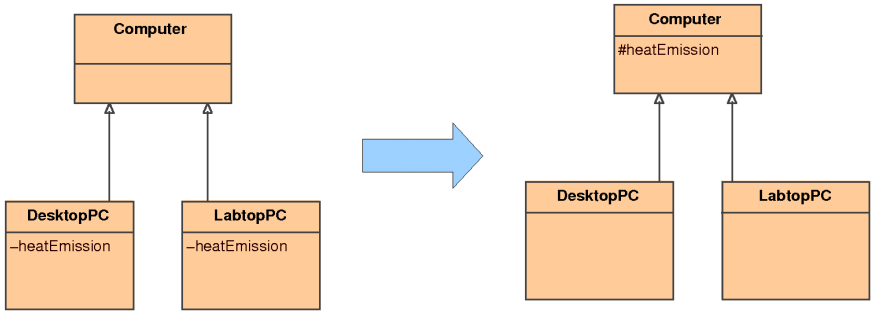


Figure E.13: Example UML model refactoring *Pull Up Property*

CONTEXTUAL ELEMENT `Property`

INITIAL PRECONDITIONS CHECK The contextual property is owned by a class that has a superclass.

REFACTORING PARAMETERS `className` - Name of the superclass the contextual property has to be pulled up to.

FINAL PRECONDITIONS CHECK (1) The owning class of the contextual property has a superclass named `className`. (2) The superclass does not own an attribute with the same name as the contextual property. (3) There is no attribute with the same name as the contextual property in the inheritance hierarchy of the superclass that is visible to it. (4) Each subclass of this superclass owns an attribute with the same name, type, visibility, and multiplicity as the contextual property.

MODEL TRANSFORMATION (1) Move the contextual property to the class named `className`. (2) Remove each attribute with the same name, type, visibility, and multiplicity as the contextual property from the subclasses of the class named `className`. If needed change the visibility of the contextual property.

POSTCONDITIONS (1) The contextual property is owned by the superclass. (2) The subclasses of the superclass do not own an attribute with the same name as the contextual property. (3) The contextual property is still visible in all subclasses of the superclass.

E.14 push down operation

DESCRIPTION This refactoring pushes an operation from the owning class down to all its subclasses. If it makes sense, the operation can be removed from some of these afterwards. Sometimes, it also makes sense to keep an operation in all subclasses to hide it from the superclass. [107, 150, 93]

EXAMPLE In Figure E.14 the operation *paint* detects what to paint, a triangle or a square. Better move the operation to the subclasses *Triangle* and *Square*, so you do need a subclass detection.

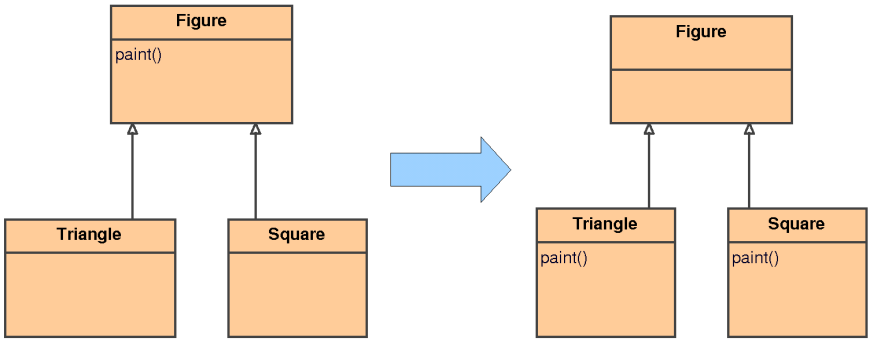


Figure E.14: Example UML model refactoring *Push Down Operation*

CONTEXTUAL ELEMENT `Operation`

INITIAL PRECONDITIONS CHECK (1) The owning class of the contextual operation has subclasses. (2) No subclass of this owning class contains an operation with the same name and the same parameter list (equal parameter names and types) as the contextual operation.

REFACTORING PARAMETERS This refactoring does not have any more parameters.

FINAL PRECONDITIONS CHECK There are no final preconditions that need to be checked.

MODEL TRANSFORMATION (1) Add a copy of the contextual operation to each subclass. (2) Remove the contextual operation.

POSTCONDITIONS (1) A copy of the contextual operation exist in each subclass. (2) The contextual operation does not exist anymore.

E.15 push down property

DESCRIPTION An attribute (property) is used only by some subclasses. Move the attribute to only these subclasses. More gen-

erally, this refactoring moves the attribute to all subclasses. If it makes sense, the attribute can be removed from some of these afterwards. Sometimes, it also makes sense to keep an attribute in all subclasses to hide it from the superclass. [107, 30, 150]

EXAMPLE Figure E.15 shows class *Figure* that has two length attributes. One has no use in the case that the figure is a square. Thus, attribute *lengthB* is pushed down. After this refactoring it might be removed from class *Square*.

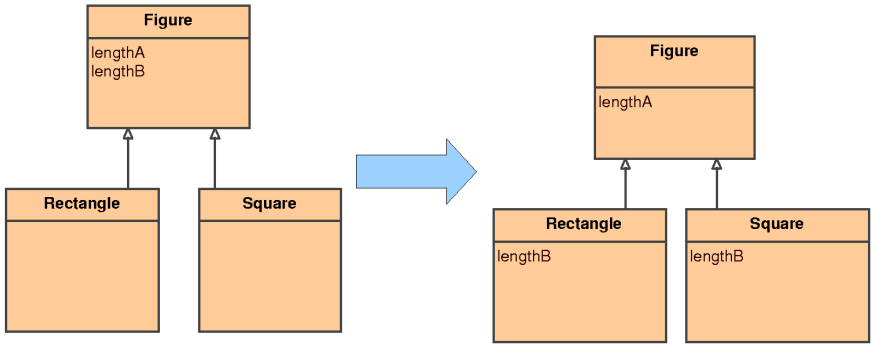


Figure E.15: Example UML model refactoring *Push Down Property*

CONTEXTUAL ELEMENT `Property`

INITIAL PRECONDITIONS CHECK (1) The owning class of the contextual property has subclasses. (2) No subclass of this owning class contains an attribute with the same name as the contextual property.

REFACTORING PARAMETERS This refactoring does not have any more parameters.

FINAL PRECONDITIONS CHECK There are no final preconditions that need to be checked.

MODEL TRANSFORMATION (1) Add a copy of the contextual property to each subclass. (2) Remove the contextual property.

POSTCONDITIONS (1) A copy of the contextual property exist in each subclass. (2) The contextual property does not exist anymore.

E.16 remove empty associated class

DESCRIPTION There is an empty class that is associated to another class. An associated class is empty if it has no features except for possible getter and setter operations for the corresponding association end. Furthermore, it has no inner classes, subclasses,

or superclasses, it does not implement any interfaces, and it is not referred to as type of an attribute, operation or parameter. [150, 93]

EXAMPLE Figure E.16 shows an empty class *ZipCode* that is associated to class *Address*. This empty class has to be deleted by this refactoring.

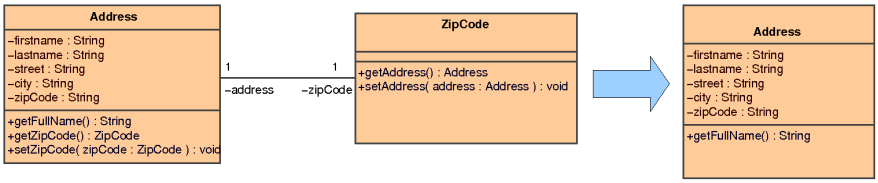


Figure E.16: Example UML model refactoring *Remove Empty Associated Class*

CONTEXTUAL ELEMENT `Class`

INITIAL PRECONDITIONS CHECK (1) The contextual class is associated to exactly one other class. (2) The contextual class has no attributes except for a possibly owned association end. (3) The contextual class has no operations except for possible getter and setter operations for the corresponding association end. (4) The contextual class has no inner class. (5) The contextual class has no subclass. (6) The contextual class has no superclass. (7) The contextual class does not implement any interfaces. (8) The contextual class is not referred to as type of an attribute, operation or parameter.

REFACTORING PARAMETERS This refactoring does not have any more parameters.

FINAL PRECONDITIONS CHECK There are no final preconditions that need to be checked.

MODEL TRANSFORMATION (1) Remove the association to the contextual class together with its getter and setter operations for the corresponding association ends. (2) Delete the contextual class.

POSTCONDITIONS The contextual class does not exist anymore.

E.17 remove empty subclass

DESCRIPTION A superclass has an empty subclass which shall be removed. This class is not associated to another class. It has no features, no inner classes, no further subclasses, and is not associated to other classes. It does not implement any interfaces,

and it is not referred to as type of an attribute, operation or parameter. [150]

EXAMPLE In Figure E.17 class *SpecialRecord* is superclass of class *Record*. Class *SpecialRecord* has become empty after some other refactorings. Remove the useless superclass *SpecialRecord*.

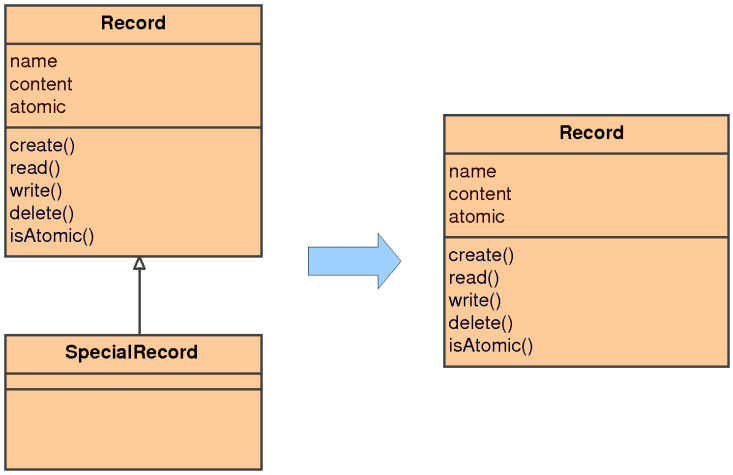


Figure E.17: Example UML model refactoring *Remove Empty Subclass*

CONTEXTUAL ELEMENT `Class`

INITIAL PRECONDITIONS CHECK (1) The contextual class is a subclass of at least one subclass. (2) The contextual class has no attributes. (3) The contextual class has no operations. (4) The contextual class has no further subclasses. (5) The contextual class is not associated to other classes. (6) The contextual class has no inner classes. (7) The contextual class does not implement any interfaces. (8) The contextual class is not referred to as type of an attribute, operation or parameter.

REFACTORING PARAMETERS This refactoring does not have any more parameters.

FINAL PRECONDITIONS CHECK There are no final preconditions that need to be checked.

MODEL TRANSFORMATION (1) Remove the inheritance relations of the contextual class to its superclasses. (2) Delete the contextual class.

POSTCONDITIONS The contextual class does not exist anymore.

E.18 remove empty superclass

DESCRIPTION A set of classes has an empty superclass which shall be removed. This class is not associated to another class. It has no features, no inner classes, and is not associated to other classes. It does not implement any interfaces, and it is not referred to as type of an attribute, operation or parameter. [150]

EXAMPLE Figure E.18 shows class *RequestHandler* that is superclass of class *HttpRequestHandler*. Class *RequestHandler* has become empty after some other refactorings. Remove the useless superclass *RequestHandler*.

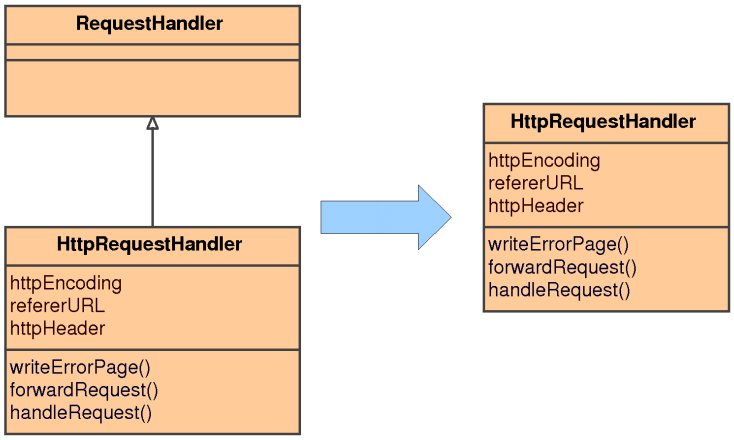


Figure E.18: Example UML model refactoring *Remove Empty Superclass*

CONTEXTUAL ELEMENT `Class`

INITIAL PRECONDITIONS CHECK (1) The contextual class is a superclass of a set of subclasses. (2) The contextual class has no attributes. (3) The contextual class has no operations. (4) The contextual class is not associated to other classes. (5) The contextual class has no inner classes. (6) The contextual class does not implement any interfaces. (7) The contextual class is not referred to as type of an attribute, operation or parameter.

REFACTORING PARAMETERS This refactoring does not have any more parameters.

FINAL PRECONDITIONS CHECK There are no final preconditions that need to be checked.

MODEL TRANSFORMATION (1) If the contextual class has again superclasses, move all inheritance relations of all its subclasses to this superclasses. (2) If the contextual class has no further superclasses, remove all inheritance relations from all its subclasses to the contextual class. (3) Delete the contextual class.

POSTCONDITIONS (1) The contextual class does not exist anymore. (2) All classes still inherit all features of potential superclasses.

E.19 remove parameter

DESCRIPTION A parameter is no longer needed by the implementation of an operation. Therefore, this refactoring removes this parameter from the parameter list of the corresponding operation. [30, 150]

EXAMPLE Figure E.19 shows parameter *length* in operation *calcArea*. This parameter is unneeded since class *Triangle* already contains every needed information for area calculation. So, this parameter shall be removed.

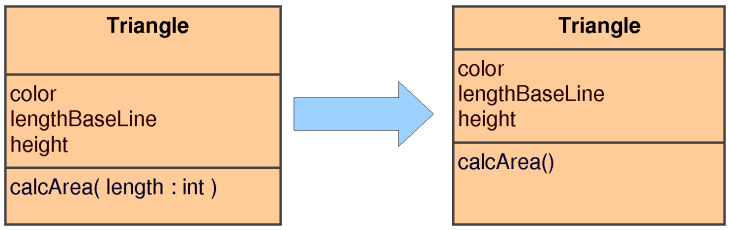


Figure E.19: Example UML model refactoring *Remove Parameter*

CONTEXTUAL ELEMENT `Parameter`

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked.

REFACTORING PARAMETERS This refactoring does not have any more parameters.

FINAL PRECONDITIONS CHECK There is no operation with the same name as the operation owning the contextual parameter and with the same parameter list (equal parameter names and types) as the operation owning the contextual parameter excluding the contextual parameter in the class and its inheritance hierarchy owning the operation with the contextual parameter.

POSTCONDITIONS The contextual parameter does not exist in the parameter list of its previous owning operation.

E.20 remove superclass

DESCRIPTION There is a set of classes having a superclass that does not make sense anymore. Remove this superclass after pushing remaining features down. [141, 150, 107, 160]

EXAMPLE Figure E.20 shows class *HttpRequestHandler* that has an unused superclass *RequestHandler* that has just one attribute and one operation. Move these features down and remove the superclass *HttpRequestHandler*.

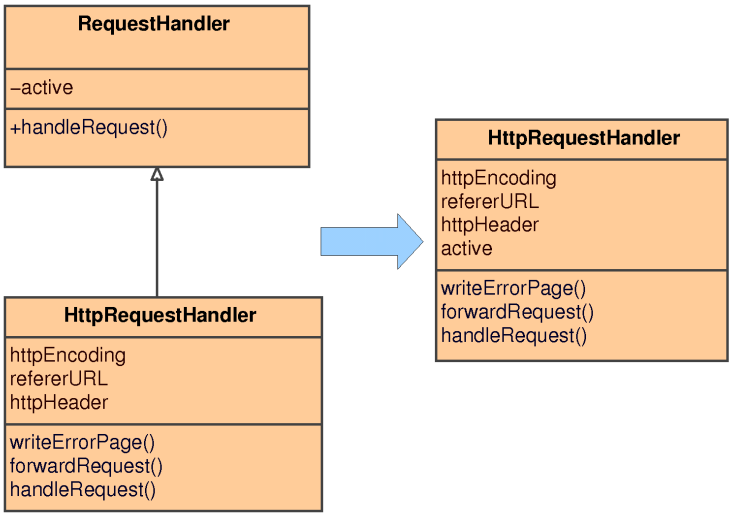


Figure E.20: Example UML model refactoring *Remove Superlass*

CONTEXTUAL ELEMENT `Class`

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked. However, the initial preconditions of the involved refactorings have to be checked properly.

REFACTORING PARAMETERS This refactoring has no additional parameter. A list of attributes and operations which have to be pushed to the existing subclasses is taken from the contextual class.

FINAL PRECONDITIONS CHECK There are no final preconditions that need to be checked. However, the final preconditions of the involved refactorings have to be checked properly.

MODEL TRANSFORMATION (1) Use refactoring *Push Down Property* on each attribute of the appropriate parameter list. (2) Use refactoring *Push Down Operation* on each operation of the appropriate parameter list. (3) Use refactoring *Remove Empty Superclass* on the contextual class.

POSTCONDITIONS In each step the postconditions of the used refactorings have to be checked. No additional postconditions are required.

E.21 rename class

DESCRIPTION The current name of a class does not reflect its purpose. This refactoring changes the name of the class to a new name. [30]

EXAMPLE In Figure E.21 there is a typing mistake in class *WeeelChair*. Correct the mistake.

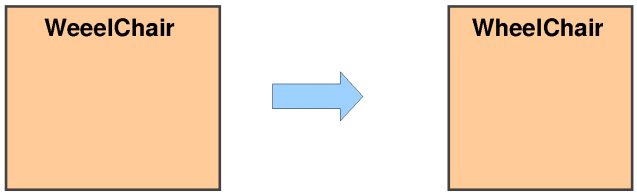


Figure E.21: Example UML model refactoring *Rename Class*

CONTEXTUAL ELEMENT `Class`

INITIAL PRECONDITIONS CHECK The namespace of the contextual class is a package.

REFACTORING PARAMETERS `newName` - New name of the contextual class.

FINAL PRECONDITIONS CHECK There is no classifier in the owning package of the contextual class named `newName`.

MODEL TRANSFORMATION Change the name of the contextual class to `newName`.

POSTCONDITIONS The name of the contextual class is `newName`.

E.22 rename operation

DESCRIPTION The current name of an operation does not reflect its purpose. This refactoring changes the name of the operation. [107, 30]

EXAMPLE Figure E.22 shows class *Book* that owns an operation *gettitle*. Since *camel case* makes it easier to read the operation's name is changed to *getTitle*.

CONTEXTUAL ELEMENT `Operation`

INITIAL PRECONDITIONS CHECK There are no initial preconditions that need to be checked.

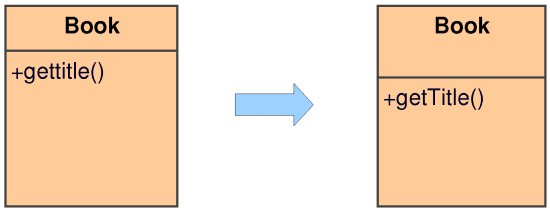


Figure E.22: Example UML model refactoring *Rename Operation*

REFACTORING PARAMETERS `newName` - New name of the contextual operation.

FINAL PRECONDITIONS CHECK (1) There is no operation named `newName` and with the same parameter list (equal parameter names and types) as the contextual operation in the class owning the contextual operation. (2) There is no operation named `newName` and with the same parameter list (equal parameter names and types) as the contextual operation in the inheritance hierarchy of the class owning the contextual operation.

MODEL TRANSFORMATION Change the name of the contextual operation to `newName`.

POSTCONDITIONS The name of the contextual operation is `newName`.

E.23 rename property

DESCRIPTION The current name of an attribute or association end does not reflect its purpose. This refactoring changes the name of the property. [107, 30]

EXAMPLE In Figure E.23 class *Wall* owns an attribute *wallColor*. Change the name to *color* since *Wall* is already the class name.

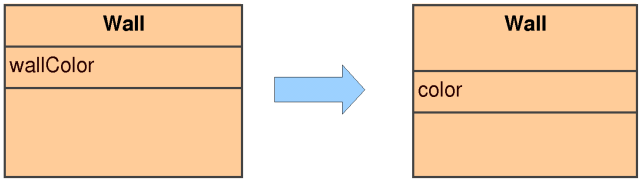


Figure E.23: Example UML model refactoring *Rename Property*

CONTEXTUAL ELEMENT `Property`

INITIAL PRECONDITIONS CHECK The contextual property is an attribute or association end of a class.

REFACTORING PARAMETERS `newName` - New name of the contextual attribute or association end.

FINAL PRECONDITIONS CHECK (1) There is no attribute named `newName` in the class owning the contextual attribute. (2) There is no attribute named `newName` in the inheritance hierarchy of the class owning the contextual attribute or association end.

MODEL TRANSFORMATION Change the name of the contextual attribute or association end to `newName`.

POSTCONDITIONS The name of the contextual attribute or association end is `newName`.

F

IMPLEMENTATIONS OF UML MODEL REFACTORINGS

In this appendix we describe the implemented specifications of selected refactorings for UML class models found in literature. For a structured specification and discussion of these refactorings we refer to Appendix E of this thesis. For each implemented model refactoring short descriptions of the refactoring, its contextual element and its parameters are given as well as a detailed description of its concrete implementation. Here, we either discuss the corresponding Henshin rules, Java code snippets, or CoMReL specifications. Furthermore, we present a list of tests that have been performed.

F.1 add parameter

DESCRIPTION An operation needs more information from its callers. Therefore, this refactoring adds a parameter to an operation. [30, 150]

CONTEXTUAL ELEMENT `Operation`

REFACTORING PARAMETERS (1) `parameterName` - Name of the new parameter. (2) `parameterType` - Type of the new parameter.

IMPLEMENTATION Refactoring *Add Parameter* has been implemented in Java code using the UML2EMF API.

```
173 public RefactoringStatus checkInitialConditions() {
174     RefactoringStatus result = new RefactoringStatus();
175     org.eclipse.uml2.uml.Operation selectedEObject =
176       (org.eclipse.uml2.uml.Operation) dataManagement.
177           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
178     // test: the selected operation must be owned by a class
179     String msg = "This refactoring can only be applied" +
180             " on operations which are owned by a class!";
181     if (selectedEObject.getClass_() == null) result.addFatalError(msg);
182     return result;
183 }
```

Figure F.1: Initial Check Implementation of UML class model refactoring *Add Parameter*

Figure F.1 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`). Here, the only check is whether the contextual `Operation` (named *selectedEObject*) is owned by a `Class` (line 181) since this refactoring should not be applied on interface operations. If this precondition is violated, an appropriate error message is returned (lines 179–181).

Figure F.2 shows the concrete implementation of the final precondition check (Java method `checkFinalConditions()`). Besides the contextual `Operation` *selectedEObject*, this check additionally considers the user input parameters *paramName* and *paramType*. Here, the following checks are performed:

1. The contextual `Operation` must not own a `Parameter` named as specified in *paramName* (lines 203–209).
2. There must be a `Type` element in the model named as specified in *paramType* (lines 211–216).
3. There must be at most one `Type` element in the model named as specified in *paramType* (lines 219–221).
4. The owning `Class` of the contextual `Operation` must not own a similar `Operation` after inserting the new `Parameter` (lines 226–232).

```
192 public RefactoringStatus checkFinalConditions(){
193     RefactoringStatus result = new RefactoringStatus();
194     org.eclipse.uml2.uml.Operation selectedEObject =
195       (org.eclipse.uml2.uml.Operation) dataManagement.
196           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
197     String paramName =
198       (String) dataManagement.getInPortByName("paramName").getValue();
199     String paramType =
200       (String) dataManagement.getInPortByName("paramType").getValue();
201     // test: the selected operation must not own a parameter with the
202     // specified name
203     String msg = "The selected operation already owns a parameter " +
204         "named '" + paramName + "'!";
205     EList<Parameter> inputParameters =
206         UmlUtils.getInputParameterList(selectedEObject.getOwnedParameters());
207     for (Parameter p : inputParameters) {
208       if (p.getName().equals(paramName)) result.addFatalError(msg);
209     }
210     // test: the specified type must exist
211     EList<Type> types = UmlUtils.getTypes(selectedEObject.getModel());
212     EList<NamedElement> typesWithName =
213             UmlUtils.getNamedElementsFromList(types, paramType);
214     msg = "The specified type does not exist in the model!";
215     if (typesWithName.isEmpty()) {
216       result.addFatalError(msg);
217     } else {
218       // test: the specified type exists multiply
219       msg = "The specified type exists multiply in the model!";
220       if (typesWithName.size() > 1) {
221         result.addFatalError(msg);
222       } else {
223         // test: the owning class must not own an operation with the name
224         // of the contextual operation and a similar parameter list after
225         // inserting a parameter with the given name and type
226         msg = "The owning class already owns an operation named '" +
227             selectedEObject.getName() + "' having the same signature " +
228             "(type and parameter list) after inserting a parameter named '" +
229             paramName + "' with type '" + paramType + "'!";
230         Class cl = selectedEObject.getClass_();
231         if (classOwnsOperation(cl, selectedEObject, paramName, paramType))
232                                 result.addFatalError(msg);
233         // test: the owning class must not own an operation with the name
234         // of the contextual operation and a similar parameter list after
235         // inserting a parameter with the given name and type
236         msg = "The owning class already inherits an operation named '" +
237             selectedEObject.getName() + "' having the same signature " +
238             "(type and parameter list) after inserting a parameter named '" +
239             paramName + "' with type '" + paramType + "'!";
240         if (classInheritsOperation(cl, selectedEObject, paramName, paramType))
241                                 result.addFatalError(msg);
242       }
243     }
244     return result;
245 }
```

Figure F.2: Final Check Implementation of UML class model refactoring *Add Parameter*

5. The owning `Class` of the contextual `Operation` must not inherit a similar `Operation` after inserting the new `Parameter` (lines 236–242).

Figure F.3 shows the implementation of the proper model change of refactoring *Add Parameter* (method `run()`). A new `Parameter` is created, named as specified in refactoring parameter *paramName*, and typed as specified in parameter *paramType* (lines 139–144). Finally, this newly created parameter is inserted into the parameter list of the contextual operation (line 145).

```
129 public void run() {
130   org.eclipse.uml2.uml.Operation selectedEObject =
131     (org.eclipse.uml2.uml.Operation) dataManagement.
132         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
133   String paramName =
134     (String) dataManagement.getInPortByName("paramName").getValue();
135   String paramType =
136     (String) dataManagement.getInPortByName("paramType").getValue();
137   // execute: insert a new parameter with the specified name and type
138   // to the selected operation
139   Parameter param = UMLFactory.eINSTANCE.createParameter();
140   param.setName(paramName);
141   EList<Type> types = UmlUtils.getTypes(selectedEObject.getModel());
142   EList<NamedElement> typesWithName =
143           UmlUtils.getNamedElementsFromList(types, paramType);
144   param.setType((Type) typesWithName.get(0));
145   selectedEObject.getOwnedParameters().add(param);
146 }
```

Figure F.3: Model Change Implementation of UML class model refactoring *Add Parameter*

TEST CASES The following test cases have been performed:

1. The contextual operation *op1* is owned by an interface *Interf1* ⇒ corresponding error message. ✓
2. *paramName* is set to *p1*; the contextual operation *op1* owns a parameter named *p1* ⇒ corresponding error message. ✓
3. *paramType* is set to *Type1*; the model does not contain a type element named *Type1* ⇒ corresponding error message. ✓
4. *paramType* is set to *Type1*; the model contains two type elements named *Type1* ⇒ corresponding error message. ✓
5. *paramName* is set to *p1*; *paramType* is set to *Type1*; the owning class *A* of the contextual operation *op1* owns an operation *op1* with the same signature as the contextual operation *op1* after inserting a new parameter *p1* with type *Type1* ⇒ corresponding error message. ✓
6. *paramName* is set to *p1*; *paramType* is set to *Type1*; the owning class *A* of the contextual operation *op1* inherits an operation *op1* with the same signature as the contextual operation *op1* after inserting a new parameter *p1* with type *Type1* ⇒ corresponding error message. ✓
7. *paramName* is set to *p1*; *paramType* is set to *Type1*; no precondition is violated ⇒ refactoring execution as expected. ✓

F.2 create associated class

DESCRIPTION This refactoring creates an empty class and connects it with a new association to the source class from where it is extracted. The multiplicity of the new association is 1 at both ends. Usually, refactorings *Move Property* and *Move Operation* are the next steps after this refactoring. [107, 150]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS (1) `className` - Name of the new associated class. (2) `associationName` - Name of the new association. (4) `roleName1` and (5) `roleName2` - Names of the association ends of the new association.

IMPLEMENTATION Refactoring *Create Associated Class* has been implemented in Henshin pattern rules (for specifying precondition checks) and a Henshin transformation rule (for specifying the proper model changes) using the abstract syntax of UML.

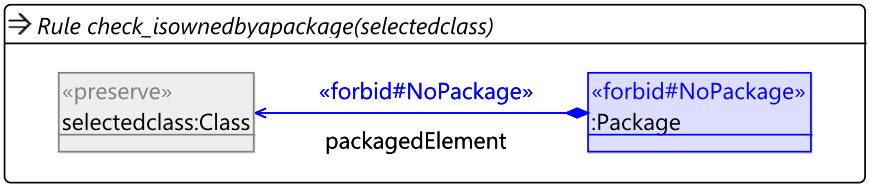


Figure F.4: Initial Check Implementation of UML class model refactoring *Create Associated Class*

Figure F.4 shows the Henshin pattern rule specification of the initial precondition check. It specifies that the contextual `Class` (named *selectedclass*) is **not** owned by a `Package` (see NAC *NoPackage*). Please note that this pattern models the situation of a violated precondition[1]. This means, that refactoring *Create Associated Class* can only be applied on classes which **are** owned by a package!

Figure F.5 shows the Henshin pattern rule specifications of the final precondition checks. Altogether four violated preconditions are modeled:

1. The owning `Package` of the contextual `Class` already owns an element named as specified in parameter *classname*.
2. The owning `Package` of the contextual `Class` already owns an element named as specified in parameter *associationname*.

1 This is done to provide meaningful error messages (see Section 13.4 of this thesis).

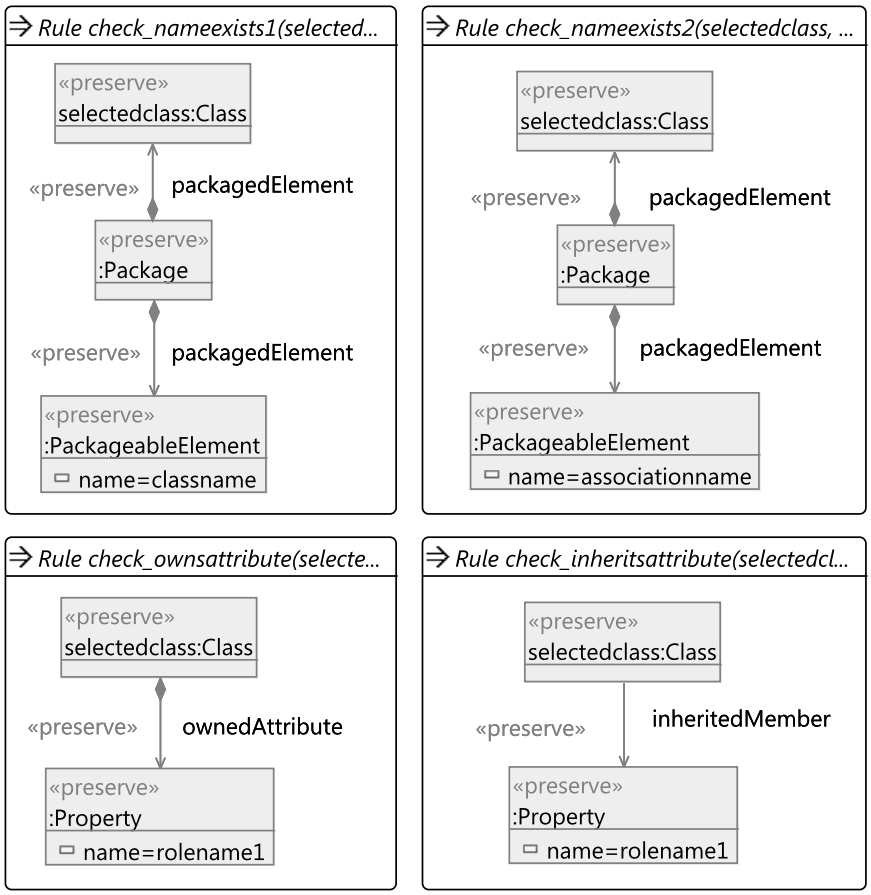


Figure F.5: Final Check Implementation of UML class model refactoring *Create Associated Class*

3. The contextual `Class` already owns an attribute named as specified in parameter *rolename1*.
4. The contextual `Class` already inherits an attribute named as specified in parameter *rolename1*.

Figure F.6 shows the Henshin pattern rule specification of the proper model change of refactoring *Create Associated Class*. This rule inserts new `Class` respectively `Association` objects into the owning `Package` of the contextual `Class` with names as specified in parameters *classname* respectively *associationname*. Furthermore, it creates the corresponding association end `Properties` as attributes of both classes with appropriate names and types.

TEST CASES The following test cases have been performed:

1. The contextual class A is an inner class of class $B \Rightarrow$ corresponding error message. ✓

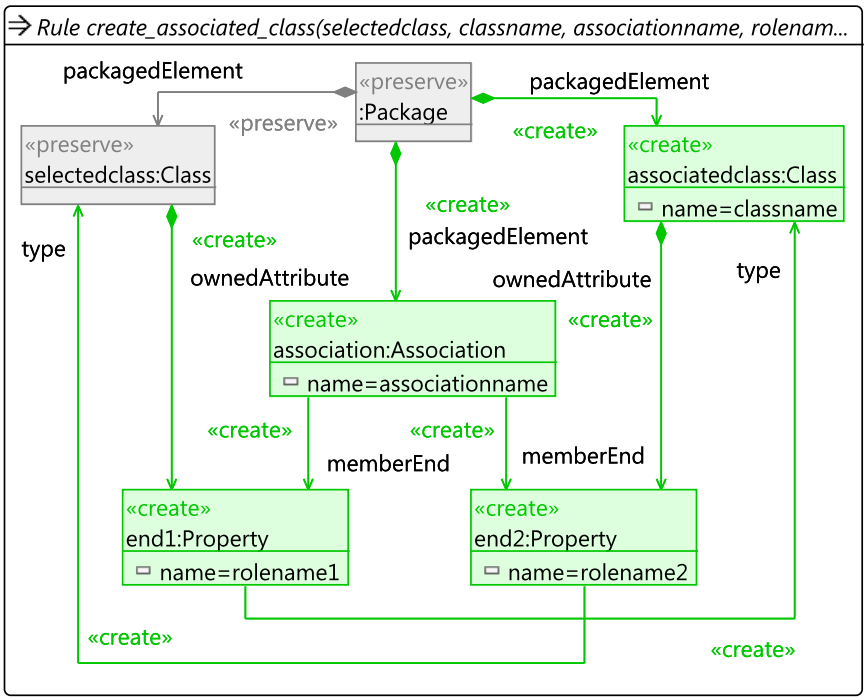


Figure F.6: Model Change Implementation of UML class model refactoring *Create Associated Class*

2. *classname* is set to *B*; the owning package *P1* of the contextual class *A* already owns a class *B* ⇒ corresponding error message. ✓
3. *classname* is set to *Interf1*; the owning package *P1* of the contextual class *A* owns an interface *Interf1* ⇒ corresponding error message. ✓
4. *associationname* is set to *assoc1*; the owning package *P1* of the contextual class *A* already owns an association *assoc1* ⇒ corresponding error message. ✓
5. *associationname* is set to *Interf1*; the owning package *P1* of the contextual class *A* owns an interface *Interf1* ⇒ corresponding error message. ✓
6. *rolename1* is set to *attr1*; the contextual class *A* already owns an attribute *attr1* ⇒ corresponding error message. ✓
7. *rolename1* is set to *attr1*; the contextual class *A* already inherits an attribute *attr1* ⇒ corresponding error message. ✓
8. *classname* is set to *B*; *associationname* is set to *assoc1*; *rolename1* is set to *b1*; *rolename2* is set to *a1*; no precondition is violated ⇒ refactoring execution as expected. ✓

F.3 create subclass

DESCRIPTION A class has features (attributes or operations) that are not used in all instances. This refactoring creates a subclass for that subset of features. However, the new subclass has no features. Usually, refactorings *Push Down Property* and *Push Down Operation* are the next steps after this refactoring. [150]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS `className` - Name of the new subclass.

IMPLEMENTATION Refactoring *Create Subclass* has been implemented in Java code using the UML2EMF API.

```
165 public RefactoringStatus checkInitialConditions() {
166     RefactoringStatus result = new RefactoringStatus();
167     org.eclipse.uml2.uml.Class selectedEObject =
168       (org.eclipse.uml2.uml.Class) dataManagement.
169           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
170     // test: the selected class must be owned by a package
171     String msg = "This refactoring can only be applied" +
172             " on classes which are owned by a package!";
173     if (selectedEObject.getPackage() == null) result.addFatalError(msg);
174     return result;
175 }
```

Figure F.7: Initial Check Implementation of UML class model refactoring *Create Subclass*

Figure F.7 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`). Here, the only check is whether the contextual `Class` (named *selectedEObject*) is owned by a `Package` (line 173), i.e., this refactoring can not be applied on inner classes, for example. If this precondition is violated, an appropriate error message is returned (lines 171–173).

```
184 public RefactoringStatus checkFinalConditions() {
185     RefactoringStatus result = new RefactoringStatus();
186     org.eclipse.uml2.uml.Class selectedEObject =
187       (org.eclipse.uml2.uml.Class) dataManagement.
188           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
189     String className =
190       (String) dataManagement.getInPortByName("className").getValue();
191     // test: the owning package must not own an element with the inserted name
192     String msg = "The owning package already owns " +
193         "an element named '" + className + "'!";
194     Package p = selectedEObject.getPackage();
195     if (p.getPackagedElement(className) != null) result.addFatalError(msg);
196     return result;
197 }
```

Figure F.8: Final Check Implementation of UML class model refactoring *Create Subclass*

Figure F.8 shows the concrete implementation of the final precondition check (Java method `checkFinalConditions()`). Besides the contextual `Class` *selectedEObject*, this check additionally considers the user input parameter *className*. Here, the only check is whether the owning `Package` of the contextual `Class` does not own an element named as specified in refactoring parameter *className* (line 192–195).

```
124 public void run() {
125   org.eclipse.uml2.uml.Class selectedEObject =
126     (org.eclipse.uml2.uml.Class) dataManagement.
127         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
128   String className =
129     (String) dataManagement.getInPortByName("className").getValue();
130   // execute: create new class named 'className'
131   Class subclass = UMLFactory.eINSTANCE.createClass();
132   subclass.setName(className);
133   selectedEObject.getPackage().getPackagedElements().add(subclass);
134   // execute: create generalization from new class to context class
135   Generalization gen = UMLFactory.eINSTANCE.createGeneralization();
136   subclass.getGeneralizations().add(gen);
137   gen.setGeneral(selectedEObject);
138 }
```

Figure F.9: Model Change Implementation of UML class model refactoring *Create Subclass*

Figure F.9 shows the concrete implementation of the proper model change of refactoring *Create Subclass* (method `run()`). First, a new `Class` is created and named as specified in refactoring parameter *className* (lines 131/132). Then, the new class is inserted into the owning `Package` of the contextual `Class` (line 133). Finally, a new `Generalization` relationship from the newly created class to the contextual class is created (lines 135–137).

TEST CASES The following test cases have been performed:

1. The contextual class *A* is an inner class of class *B* ⇒ corresponding error message. ✓
2. *className* is set to *B*; the owning package *P1* of the contextual class *A* already owns a class *B* ⇒ corresponding error message. ✓
3. *className* is set to *Interf1*; the owning package *P1* of the contextual class *A* owns an interface *Interf1* ⇒ corresponding error message. ✓
4. *className* is set to *B*; no precondition is violated ⇒ refactoring execution as expected. ✓

F.4 create superclass

DESCRIPTION This refactoring can be applied when there are at least two classes with similar features (attributes or operations). The refactoring creates a superclass for this set of classes and is normally followed by refactorings *Pull Up Property* and *Pull Up Operation*. So, the refactoring helps to reduce the duplicate common features spread throughout different classes. [141, 150, 107, 160]

CONTEXTUAL ELEMENTS Set of `Classes`

REFACTORING PARAMETERS `className` - Name of the new superclass.

IMPLEMENTATION Refactoring *Create Superclass* has been implemented in Henshin pattern rules (for specifying precondition checks) respectively Henshin transformation rules (for specifying the proper model changes) using the abstract syntax of UML.

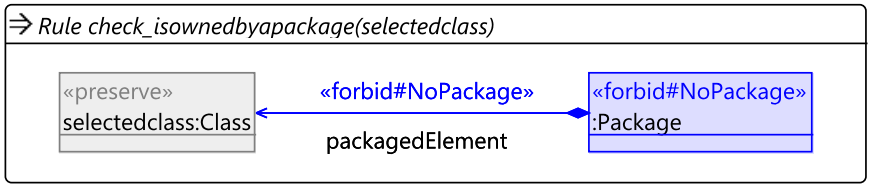


Figure F.10: Initial Check Implementation of UML class model refactoring *Create Superclass*

Figure F.10 shows the Henshin pattern rule specification of the initial precondition check. This rule defines the erroneous situation that the contextual `Class` (named *selectedclass*) is not owned by a `Package`, i.e., this refactoring can not be applied on inner classes, for example. NAC *NoPackage* specifies this violated precondition.

Figures F.11 and F.12 shows the Henshin pattern rule specifications of the final precondition checks. Altogether eleven violated preconditions are modeled. First, it is checked whether the owning `Package` of the contextual `Class` (*selectedclass*) already owns an interface named as specified in parameter *classname* (rule *check_noInterface*). Since the refactoring can also be applied if a class named as specified in parameter *classname* already exists, the remaining rules check whether this class is concrete, public and empty. The corresponding violated conditions are:

1. Rule *check_noAbstractClass*: The existing `Class` (*excl*) is abstract.

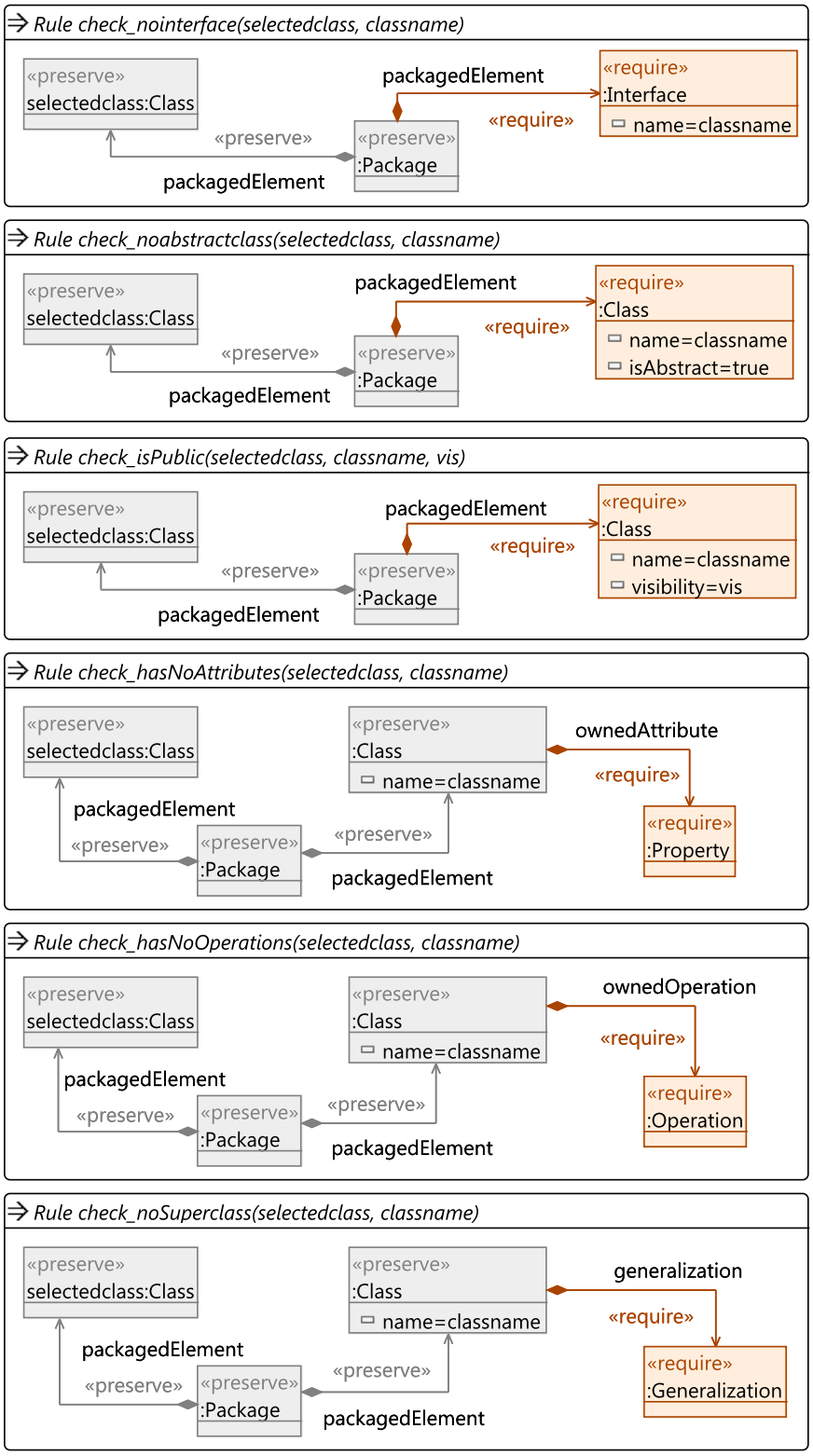


Figure F.11: Final Check Implementation of UML class model refactoring *Create Superclass* (1)

2. Rule *check_isPublic*: The existing `Class` (*excl*) is not public. Here, an additional internal rule parameter *vis* is used in combination with parameter condition `vis != public`.
3. Rule *check_hasNoAttributes*: The existing `Class` (*excl*) owns at least one attribute (`Property`).
4. Rule *check_hasNoOperation*: The existing `Class` (*excl*) owns at least one `Operation`.
5. Rule *check_noSuperclass*: The existing `Class` (*excl*) has a superclass.
6. Rule *check_noAssociation*: The existing `Class` (*excl*) is involved in at least one `Association` (as type of an association end `Property`).
7. Rule *check_noInnerClass*: The existing `Class` (*excl*) has at least one inner `Class`.
8. Rule *check_noImplementingInterface*: The existing `Class` (*excl*) implements at least one `Interface`.
9. Rule *check_noInterfaceUsing*: The existing `Class` (*excl*) uses at least one `Interface`.
10. Rule *check_noType*: The existing `Class` (*excl*) is used as type of at least one `TypedElement` (`Parameter`, for example).

Figure F.13 on page 266 shows the Henshin pattern rule specifications of the proper model changes of refactoring *Create Superclass*. Rule *create_superclass* creates a new `Class` named as specified in parameter *classname* and inserts it into the owning `Package` of the contextual `Class` (*selectedclass*) if such a class does not exist yet (specified by NAC *ClassNotExists*). Then, rule *create_generalization* creates a `Generalization` relation between the contextual class and the (potentially created) class named as specified in parameter *classname*.

TEST CASES The following test cases have been performed:

1. The contextual class A is an inner class of class B $\Rightarrow$ corresponding error message. ✓
2. *classname* is set to *Interf1*; the owning package *P1* of the contextual class A owns an interface *Interf1* $\Rightarrow$ corresponding error message. ✓
3. *classname* is set to B; the owning package *P1* of the contextual class A owns class B; B is abstract $\Rightarrow$ corresponding error message. ✓
4. *classname* is set to B; the owning package *P1* of the contextual class A owns class B; class B has private visibility $\Rightarrow$ corresponding error message. ✓

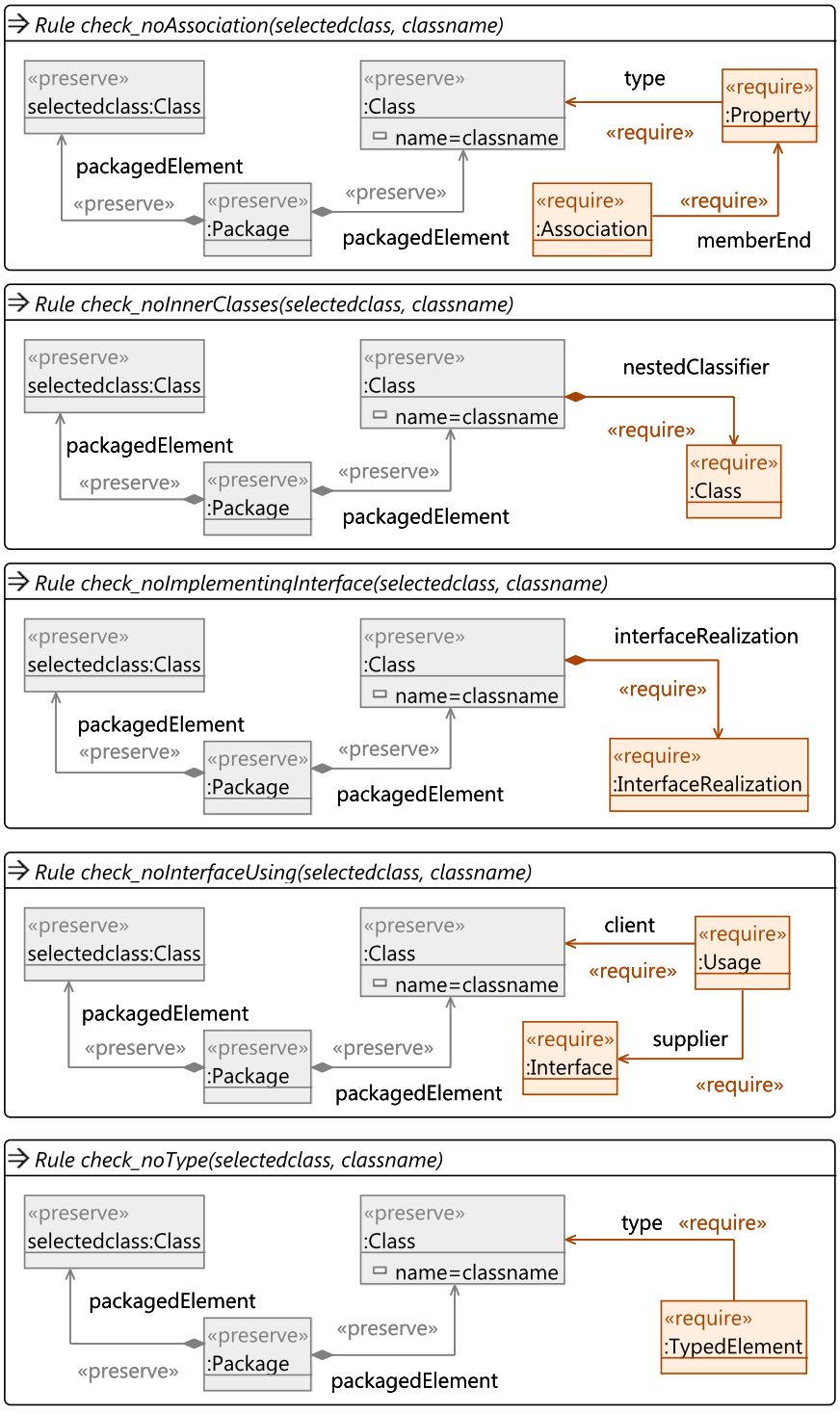


Figure F.12: Final Check Implementation of UML class model refactoring *Create Superclass* (2)

5. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* owns an attribute *att* ⇒ corresponding error message. ✓

6. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* owns an operation *op* ⇒ corresponding error message. ✓
7. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* has an inner class *C* ⇒ corresponding error message. ✓
8. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* has a superclass *C* ⇒ corresponding error message. ✓
9. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* has an incoming association *assoc* ⇒ corresponding error message. ✓
10. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* has an outgoing association *assoc* ⇒ corresponding error message. ✓

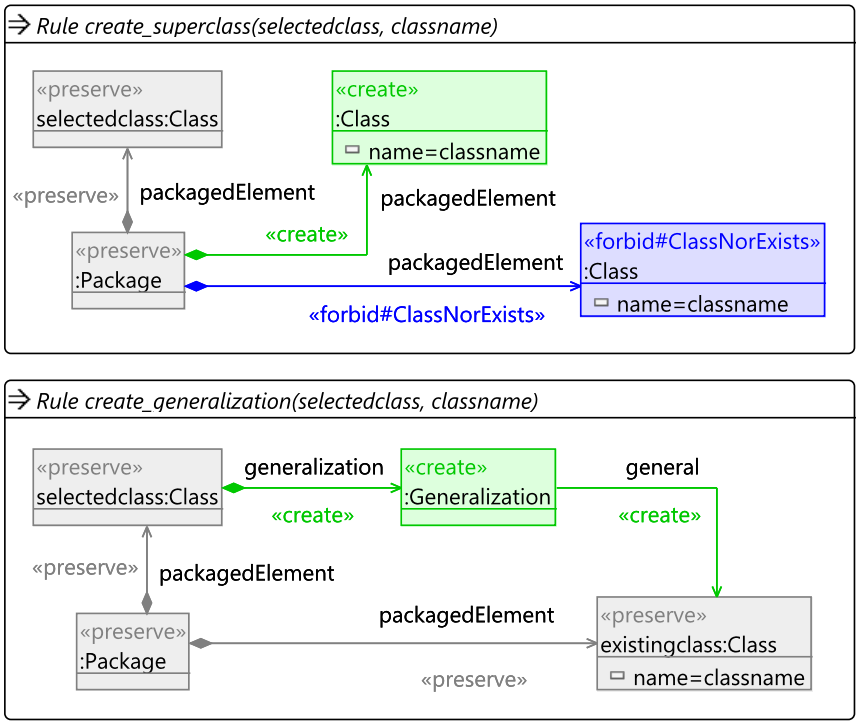


Figure F.13: Model Change Implementation of UML class model refactoring *Create Superclass*

11. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* implements an interface *Interf1* ⇒ corresponding error message. ✓
12. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* uses an interface *Interf1* ⇒ corresponding error message. ✓

13. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* is used as type of class attribute *P1::C::att* $\Rightarrow$ corresponding error message. ✓
14. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; class *B* is used as type of parameter *P1::C::op1::par1* $\Rightarrow$ corresponding error message. ✓
15. *classname* is set to *B*; the owning package *P1* of the contextual class *A* does not own a class *B*; no precondition is violated $\Rightarrow$ refactoring execution as expected. ✓
16. *classname* is set to *B*; the owning package *P1* of the contextual class *A* owns class *B*; no precondition is violated $\Rightarrow$ refactoring execution as expected. ✓

F.5 extract associated class

DESCRIPTION This refactoring extracts interrelated features (attributes and operations) from a class to a new separated class. [107, 150]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS Each parameter of refactoring *Create Associated Class*. Additionally, a list of attributes and operations which have to be moved to the new associated class.

```
260 public RefactoringStatus checkInitialConditions() {
261     RefactoringStatus result = new RefactoringStatus();
262     Class selectedEObject =
263       (Class) dataManagement.getInPortByName
264         (dataManagement.SELECTEDEOBJECT).getValue();
265     ArrayList<Property> attributesList =
266         ((UmlPropertyList) dataManagement.
267             getInPortByName("attributesList").getValue()).getUmlProperties();
268     ArrayList<Operation> operationsList =
269         ((UmlOperationList) dataManagement.
270             getInPortByName("operationsList").getValue()).getUmlOperations();
271     // test: the selected class must be owned by a package
272     String msg = "This refactoring can only be applied" +
273         " on classes which are owned by a package!";
274     if (selectedEObject.getPackage() == null) {
275       result.addFatalError(msg);
276       return result;
277     }
278     // test: each property must be an owned attribute of the selected class
279     msg = "At least one selected property is not an owned attribute of class '"
280                         + selectedEObject.getName() +"'!";
281     for (Property attribute : attributesList) {
282       if (! selectedEObject.getOwnedAttributes().contains(attribute)) {
283         result.addFatalError(msg);
284         return result;
285       }
286     }
287     // test: each operation must be an owned operation of the selected class
288     msg = "At least one selected operation is not an owned operation of class '"
289                         + selectedEObject.getName() +"'!";
290     for (Operation operation : operationsList) {
291       if (! selectedEObject.getOwnedOperations().contains(operation)) {
292         result.addFatalError(msg);
293         return result;
294       }
295     }
296     return result;
297 }
```

Figure F.14: Initial Check Implementation of UML class model refactoring *Extract Associated Class*

IMPLEMENTATION Refactoring *Extract Associated Class* has been implemented in Java code using the UML2EMF API (for specifying precondition checks) respectively an appropriate model of the CoMReL language (for specifying the proper model changes) using predefined UML refactorings.

Figure F.14 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`).

Here, the contextual elements are accessible by variables *selectedEObject* (of type `Class`), *attributesList* (as list of type `Property`), and *operationsList* (as list of type `Operation`). First, it is checked whether the contextual class is owned by a `Package`, i.e., this refactoring can not be applied on inner classes, for example (lines 272–277). Then, it is checked whether each contextual property is an attribute of the contextual class (lines 279–286) respectively whether each contextual operation is owned by the contextual class (lines 288–295).

Figure F.15 shows the concrete CoMReL unit specification of the proper model change of refactoring *Extract Associated Class*[2]. As described in Appendix E, refactoring *Extract Associated Class* relies on three atomic model refactorings. The main refactoring unit *extractclass* is a strict `Sequential Unit` consisting of one `AtomicUnit` and two `SingleQueuedUnits`. The atomic unit creates the new associated class and the queued units move all attributes and operations of the corresponding input lists to this newly created class.

Each `SingleQueuedUnit` contains an `AtomicUnit` calling an already existing refactoring. First, the atomic one *Move Attribute* must be applied on each attribute of the corresponding input list. Analogously, an atomic unit for refactoring *Move Operation* is put into a single queued unit. In both cases, the *strict* attributes are set to *true* since each feature should be moved.

TEST CASES The following test cases have been performed:

1. The contextual class A is an inner class of class B $\Rightarrow$ corresponding error message. ✓
2. Attributes *att1* and *att2* of the contextual attributes list (*att1*, *att2*, *att3*) are owned by the contextual class A; attribute *att3* of the contextual attributes list (*att1*, *att2*, *att3*) is owned by class B; $\Rightarrow$ corresponding error message. ✓
3. Operations *op1* and *op2* of the contextual operations list (*op1*, *op2*, *op3*) are owned by the contextual class A; operation *op3* of the contextual operations list (*op1*, *op2*, *op3*) is owned by class B; $\Rightarrow$ corresponding error message. ✓
4. *className* is set to B; *associationName* is set to *assoc1*; *roleName1* is set to *a*; *roleName2* is set to *b*; the contextual class A owns each attribute of the contextual attributes list (*att1*, *att2*, *att3*) as well as each operation of the contextual operations list (*op1*, *op2*, *op3*); no violated preconditions $\Rightarrow$ refactoring execution as expected. ✓

2 Although this refactoring has additional parameters (*className*, *associationName*, *roleName1* and *roleName2*), no final precondition checks have to be implemented since they are checked by the internal refactorings.

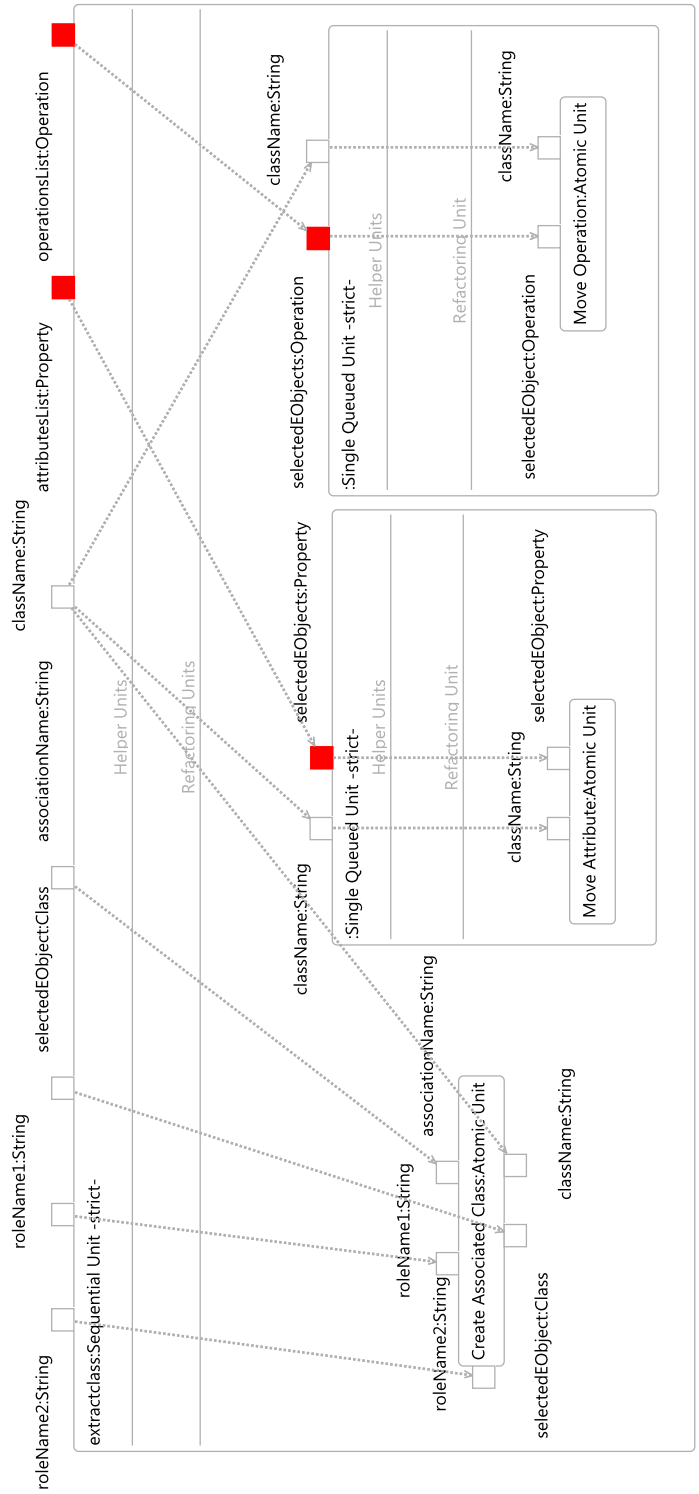


Figure F.15: Model Change Implementation of UML class model refactoring *Extract Associated Class*

F.6 extract subclass

DESCRIPTION There are features (attributes and operations) in a class required for a special case only. This refactoring extracts a subclass containing this features. [150]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS Each parameter of refactoring *Create Associated Class*. Additionally, a list of attributes and operations which have to be pushed to the new subclass.

IMPLEMENTATION Refactoring *Extract Subclass* has been implemented in Java code using the UML2EMF API (for specifying precondition checks) respectively an appropriate model of the CoMReL language (for specifying the proper model changes) using predefined UML refactorings.

```
239 public RefactoringStatus checkInitialConditions() {
240     RefactoringStatus result = new RefactoringStatus();
241     Class selectedEObject =
242         (Class) dataManagement.getInPortByName
243           (dataManagement.SELECTEDEOBJECT).getValue();
244     ArrayList<Property> attributesList = ((UmlPropertyList) dataManagement.
245               getInPortByName("attributesList").getValue()).getUmlProperties();
246     ArrayList<Operation> operationsList = ((UmlOperationList) dataManagement.
247               getInPortByName("operationsList").getValue()).getUmlOperations();
248     // test: the selected class must be owned by a package
249     String msg = "This refactoring can only be applied" +
250           " on classes which are owned by a package!";
251     if (selectedEObject.getPackage() == null) {
252       result.addFatalError(msg);
253       return result;
254     }
255     // test: each property must be an owned attribute of the selected class
256     msg = "At least one selected property is not an owned attribute of class '"
257                           + selectedEObject.getName() +"'!";
258     for (Property attribute : attributesList) {
259       if (! selectedEObject.getOwnedAttributes().contains(attribute)) {
260         result.addFatalError(msg);
261         return result;
262       }
263     }
264     // test: each operation must be an owned operation of the selected class
265     msg = "At least one selected operation is not an owned operation of class '"
266                           + selectedEObject.getName() +"'!";
267     for (Operation operation : operationsList) {
268       if (! selectedEObject.getOwnedOperations().contains(operation)) {
269         result.addFatalError(msg);
270         return result;
271       }
272     }
273     return result;
274 }
```

Figure F.16: Initial Check Implementation of UML class model refactoring *Extract Subclass*

Figure F.16 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`).

Here, the contextual elements are accessible by variables *selectedEObject* (of type `Class`), *attributesList* (as list of type `Property`), and *operationsList* (as list of type `Operation`). First, it is checked whether the contextual class is owned by a `Package`, i.e., this refactoring can not be applied on inner classes, for example (lines 249–254). Then, it is checked whether each contextual property is an attribute of the contextual class (lines 256–263) respectively whether each contextual operation is owned by the contextual class (lines 265–272).

Figure F.17 shows the concrete CoMReL unit specification of the proper model change of refactoring *Extract Subclass*[3]. As described in Appendix E, refactoring *Extract Subclass* relies on three atomic model refactorings. The main refactoring unit *extractsubclass* is a strict `SequentialUnit` consisting of one `AtomicUnit` and two `SingleQueuedUnits`. The atomic unit creates the new subclass and the queued units push down all attributes and operations of the corresponding input lists to each subclass of the contextual `Class` (i.e., also to the newly created one).

Each `SingleQueuedUnit` contains an `AtomicUnit` calling an already existing model refactoring. First, atomic refactoring *Push Down Attribute* must be applied on each attribute of the corresponding input list. Analogously, an atomic unit for refactoring *Push Down Operation* is put into a single queued unit. In both cases, the according *strict* attributes are set to *true* since each contextual feature should be pushed down.

TEST CASES The following test cases have been performed:

1. The contextual class A is an inner class of class B $\Rightarrow$ corresponding error message. ✓
2. Attributes *att1* and *att2* of the contextual attributes list (*att1*, *att2*, *att3*) are owned by the contextual class A; attribute *att3* of the contextual attributes list (*att1*, *att2*, *att3*) is owned by class B; $\Rightarrow$ corresponding error message. ✓
3. Operations *op1* and *op2* of the contextual operations list (*op1*, *op2*, *op3*) are owned by the contextual class A; operation *op3* of the contextual operations list (*op1*, *op2*, *op3*) is owned by class B; $\Rightarrow$ corresponding error message. ✓
4. *className* is set to B; the contextual class A owns each attribute of the contextual attributes list (*att1*, *att2*, *att3*) as well as each operation of the contextual operations list (*op1*, *op2*, *op3*); class A does not have any subclasses; no violated preconditions $\Rightarrow$ refactoring execution as expected. ✓

3 Although this refactoring has an additional parameter (*className*), no final precondition checks have to be implemented since they are checked by the internal refactorings.

5. *className* is set to *D*; the contextual class *A* owns each attribute of the contextual attributes list (*att1*, *att2*, *att3*) as well as each operation of the contextual operations list (*op1*, *op2*, *op3*); class *A* has subclasses *B* and *C*; no violated preconditions ⇒ refactoring execution as expected. ✓

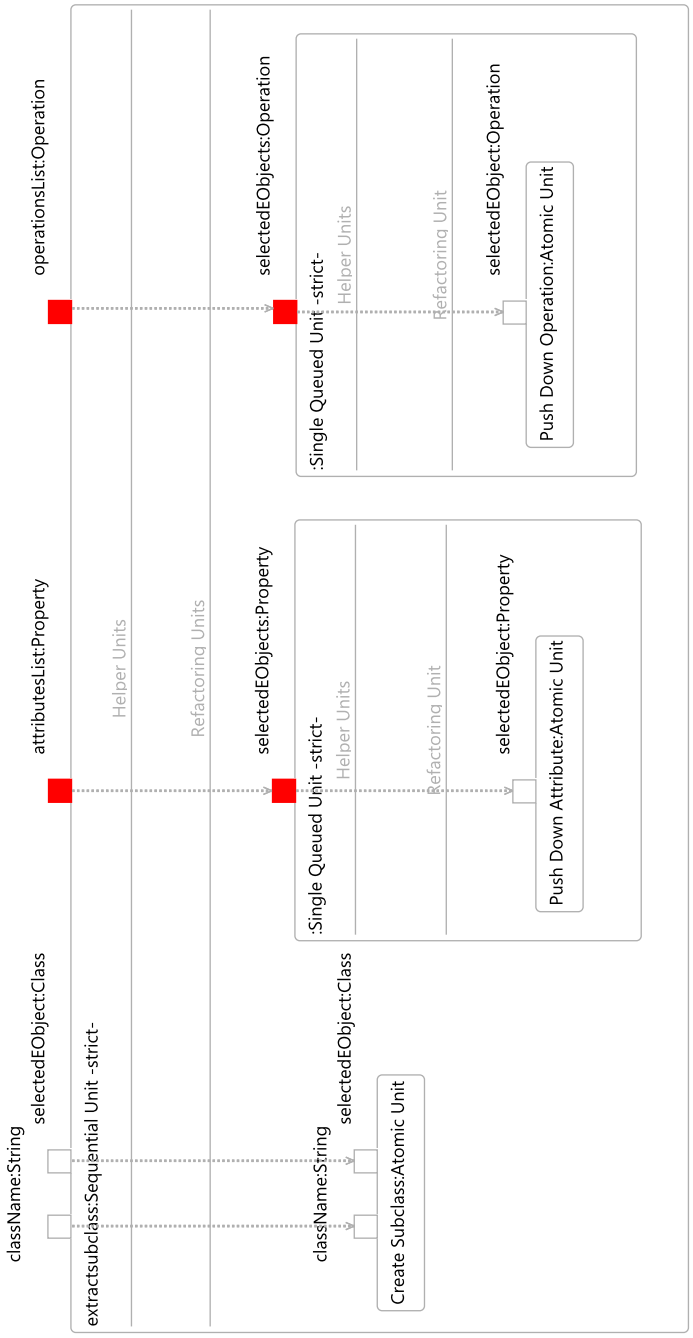


Figure F.17: Model Change Implementation of UML class model refactoring *Extract Subclass*

F.7 extract superclass

DESCRIPTION There are two or more classes with similar features. This refactoring creates a new superclass and moves the common features to the superclass. The refactoring helps to reduce redundancy by assembling common features spread throughout different classes. [141, 106, 150, 107, 160]

CONTEXTUAL ELEMENTS Set of `Classes`

REFACTORING PARAMETERS Each parameter of refactoring *Create Superclass*. Additionally, a list of attributes and operations which have to be pushed to the new subclass is taken from one contextual class.

IMPLEMENTATION Refactoring *Extract Superclass* has been implemented in Java code using the UML2EMF API (for specifying precondition checks) respectively an appropriate model of the CoMReL language (for specifying the proper model changes) using predefined UML refactorings.

```
220 public RefactoringStatus checkInitialConditions() {
221     RefactoringStatus result = new RefactoringStatus();
222     ArrayList<org.eclipse.uml2.uml.Class> selectedEObjects =
223         ((UmlClassList) dataManagement.
224             getInPortByName(dataManagement.SELECTEDEOBJECT).
225               getValue()).getUmlClasses();
226       // test: all classes are owned by different packages
227       String msg1 = "At least one class is not owned by a package!";
228       String msg2 = "The selected classes are not owned by the same package!";
229       ArrayList<Package> owningPackages = new ArrayList<Package>();
230       for (Class cls : selectedEObjects) {
231         if (cls.getPackage() == null) {
232           result.addFatalError(msg1);
233         } else {
234           if (! owningPackages.contains(cls.getPackage())) {
235             owningPackages.add(cls.getPackage());
236           }
237         }
238       }
239       if (owningPackages.size() > 1) result.addFatalError(msg2);
240     return result;
241 }
```

Figure F.18: Initial Check Implementation of UML class model refactoring *Extract Superclass*

Figure F.18 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`). Here, the only check is whether the contextual `Classes` (named *selectedEObjects*) are owned by the same `Package` (lines 229–239), i.e., this refactoring can not be applied on inner classes or classes which are owned by different packages, for example. If this pre-

condition is violated, an appropriate error message is returned (lines 227/228/232/239).

```
250 public RefactoringStatus checkFinalConditions() {
251     RefactoringStatus result = new RefactoringStatus();
252     ArrayList<org.eclipse.uml2.uml.Class> selectedEObjects =
253         ((UmlClassList) dataManagement.
254             getInPortByName(dataManagement.SELECTEDEOBJECT).
255               getValue()).getUmlClasses();
256     String className =
257       (String) dataManagement.getInPortByName("className").getValue();
258     // test: No selected class must be named like specified in 'newName'
259     String msg = "At least one selected class is already named '" + className + "'!";
260     for (Class cls : selectedEObjects) {
261       if (cls.getName().equals(className)) {
262         result.addFatalError(msg);
263         break;
264       }
265     }
266     return result;
267 }
```

Figure F.19: Final Check Implementation of UML class model refactoring *Extract Superclass*

Figure F.19 shows the concrete implementation of the final precondition check (Java method `checkFinalConditions()`). Besides the contextual `Classes` list *selectedEObjects*, this check additionally considers the user input parameter *className*. Here, it is checked whether no contextual class is already named as specified in refactoring parameter *className*, otherwise an appropriate error message is returned (lines 259–265).

Figure F.20 on page 276 shows the concrete CoMReL unit specification of the proper model change of refactoring *Extract Superclass*. As described in Appendix E, refactoring *Extract Superclass* relies on three atomic model refactorings. The main refactoring unit *extractsuperclass* is a strict `Sequential Unit` consisting of three `SingleQueuedUnits` where the first one creates superclasses for all selected classes, the second one pulls up all common attributes, and the third one pulls up all common operations.

Each `SingleQueuedUnit` contains an `AtomicUnit` calling an already existing model refactoring. First, atomic refactoring *Create Superclass* must be applied on each selected class. So, this atomic unit is put into a strict `SingleQueuedUnit` to address the looping execution for each selected class. Analogously, atomic units for refactorings *Pull Up Attribute* and `Pull Up Operation` are put into a single queued unit each. In these cases, the according *strict* attributes are set to *false* since only those features should be pulled up that fulfill the corresponding preconditions.

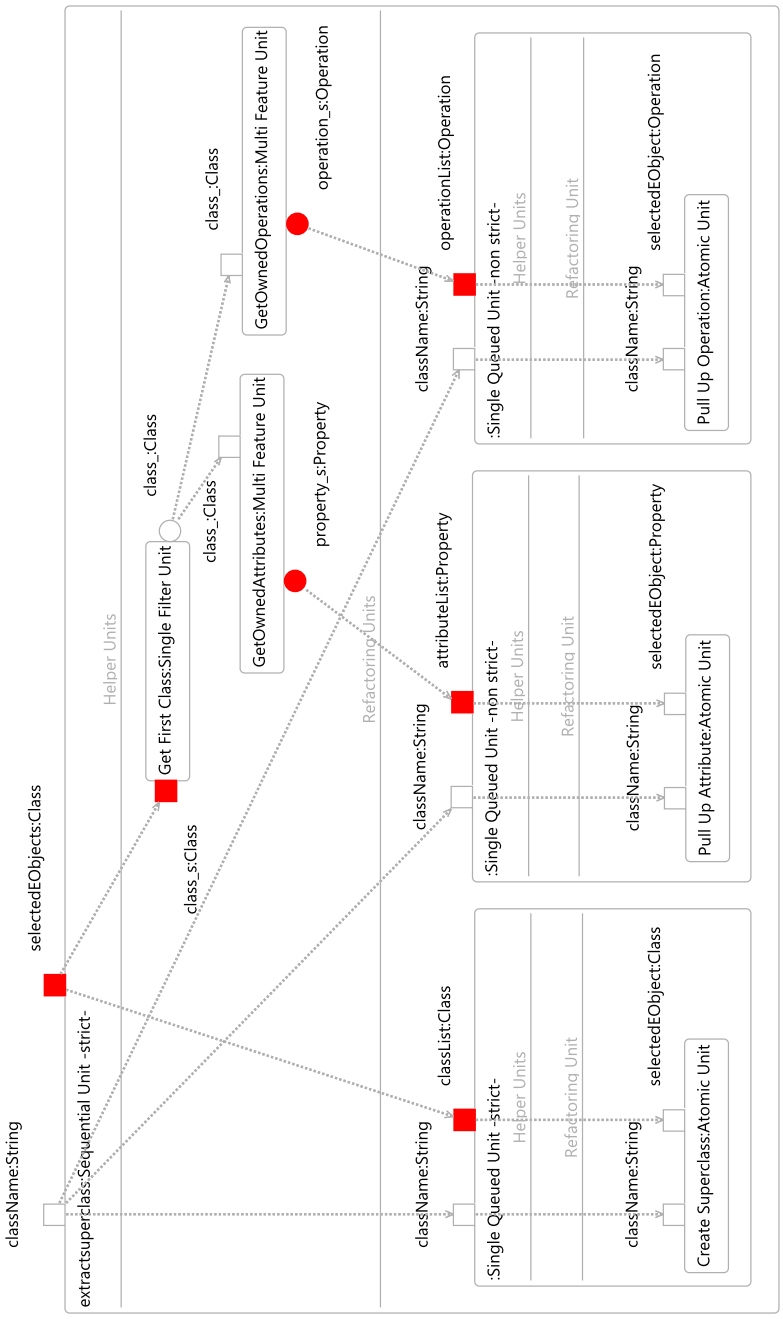


Figure F.20: Model Change Implementation of UML class model refactoring *Extract Superclass*

To ensure conformity with respect to typing and multiplicity of included ports, the main refactoring unit *extractsuperclass* requires altogether three helper units. `FilterUnit` *Get First Class* extracts the first class of a list of classes while `FeatureUnits` *Get Owned Attributes* and *Get Owned Operations* take this extracted class as input and yield all owned attributes and operations of that class.

TEST CASES The following test cases have been performed:

1. Class A of the contextual classes list (A, B, C) is an inner class of class D $\Rightarrow$ corresponding error message. ✓
2. Classes A and B of the contextual classes list (A, B, C) are owned by package *p1*; Class C of the contextual classes list (A, B, C) is owned by package *p2* $\Rightarrow$ corresponding error message. ✓
3. *className* is set to A; the contextual classes list is (A, B, C) $\Rightarrow$ corresponding error message. ✓
4. *className* is set to D; the contextual classes list is (A, B, C); each class contains equal attributes *att1* and *att2* as well as equal operations *op1* and *op2*; no violated internal preconditions $\Rightarrow$ refactoring execution as expected. ✓
5. *className* is set to D; the contextual classes list is (A, B, C); the classes do not have equal attributes nor operations; no violated internal preconditions $\Rightarrow$ refactoring execution as expected. ✓

F.8 inline class

DESCRIPTION There are two classes connected by a 1:1 association. One of them has no further use. This refactoring merges these classes. [150, 93]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS Each parameter of refactoring *Remove Empty Associated Class*. Additionally, a list of attributes and operations which have to be moved to the associated class.

IMPLEMENTATION Refactoring *Inline Class* has been implemented in Java code using the UML2EMF API (for specifying precondition checks) respectively an appropriate model of the CoMReL language (for specifying the proper model changes) using predefined UML refactorings.

```
210 public RefactoringStatus checkInitialConditions() {
211     RefactoringStatus result = new RefactoringStatus();
212     org.eclipse.uml2.uml.Class selectedEObject =
213         (org.eclipse.uml2.uml.Class) dataManagement.
214             getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
215     // test: the selected class must be associated to at least one class
216     List<Class> associatedclasses =
217                 UmlUtils.getOtherAssociatedClasses(selectedEObject);
218     String msg = "Class '" + selectedEObject.getName() +
219                  "' is not associated to any classes!";
220     if (associatedclasses.isEmpty()) {
221       result.addFatalError(msg);
222       return result;
223     }
224     return result;
225 }
```

Figure F.21: Initial Check Implementation of UML class model refactoring *Inline Class*

Figure F.21 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`). Here, the only check is whether the contextual `Class` (named *selectedEObject*) is associated to at least one other class[4], otherwise this refactoring does not make sense and an appropriate error message is returned (lines 216–223).

Figure F.22 shows the concrete CoMReL unit specification of the proper model change of refactoring *Inline Class*[5].

4 Please note that the concrete checks are implemented as static methods of a utility class in order to avoid redundant code and to reuse it as often as possible.

5 Since this refactoring does not have any additional parameters, no final precondition checks have to be implemented.

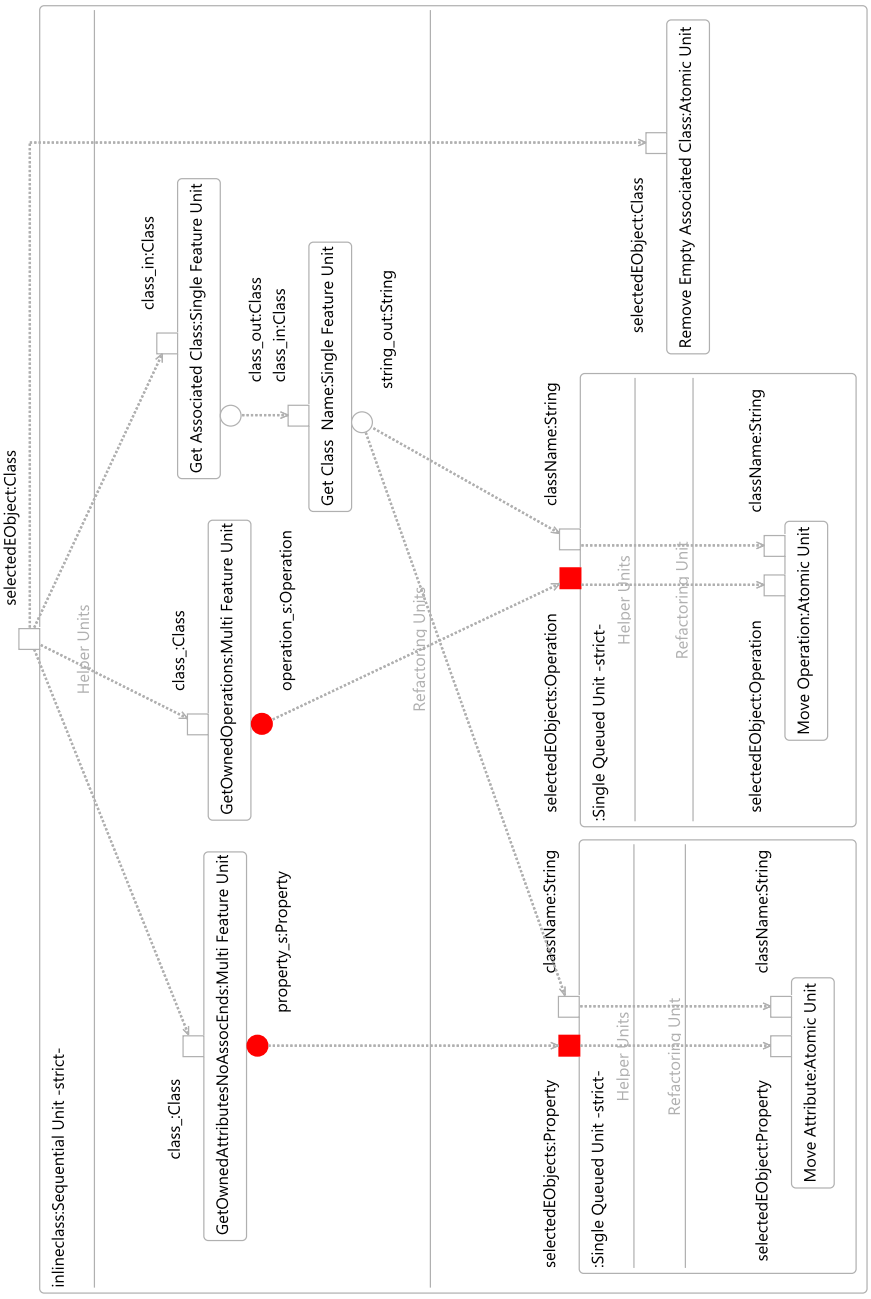


Figure F.22: Model Change Implementation of UML class model refactoring *Inline Class*

As described in Appendix E, refactoring *Inline Class* relies on three atomic model refactorings. The main refactoring unit *inlineclass* is a strict `Sequential Unit` consisting of two separate `SingleQueuedUnits` and one `AtomicUnit`. The queued units move all attributes and operations from the contextual class to a spe-

cific associated class of the contextual `Class`. Then, the contextual class is removed by the corresponding atomic unit.

Each `SingleQueuedUnit` contains an `AtomicUnit` calling an already existing model refactoring. First, atomic refactoring *Move Attribute* must be applied on each attribute of the selected class. Analogously, an atomic unit for refactoring *Move Operation* is put into a single queued unit. In both cases, the according *strict* attributes are set to *true* since each feature should be moved to ensure that the contextual class is empty afterwards.

To ensure conformity with respect to typing and multiplicity of included ports, the main refactoring unit *inlineclass* requires altogether four helper units. `MultiFeatureUnits` *Get Owned Attributes* and *Get Owned Operations* take the contextual class as input and yield all owned attributes and operations of that class. `SingleFeatureUnit` *Get Associated Class* returns an arbitrary associated class of the corresponding class. Here, the initial check of refactoring *Inline Class* guarantees that at least one associated class exists. Please note that if the contextual class has more than one associated classes refactoring *Inline Class* is not executed since a corresponding precondition of the internal refactoring *Remove Empty Associated Class* fails! Finally, `SingleFeatureUnit` *Get Class Name* returns the name of the class given by helper *Get Associated Class*.

TEST CASES The following test cases have been performed:

1. The contextual class A is not associated to any classes in the model $\Rightarrow$ corresponding error message. ✓
2. The contextual class A is associated to class B; no further associations to other classes; A owns operation *op*; B inherits an operation *op* with an equal parameter list as *A::op* $\Rightarrow$ no changes. ✓
3. The contextual class A is associated to class B; no further associations to other classes; A owns operation *op* and attribute *att*; class A is used as type of attribute *C::att1* $\Rightarrow$ no changes. ✓
4. The contextual class A is associated to classes B and C; A owns operation *op* and attribute *att* $\Rightarrow$ no changes. ✓
5. The contextual class A is associated to class B; no further associations to other classes; A owns operation *op* and attribute *att*; no violated internal preconditions $\Rightarrow$ no changes. ✓

F.9 introduce parameter object

DESCRIPTION There is a group of parameters that naturally go together. This refactoring replaces a list of parameters with one object. This parameter object is created for that purpose. [11, 131, 161, 104]

CONTEXTUAL ELEMENTS List of `Parameters`

REFACTORING PARAMETERS (1) `className` - Name of the new parameterclass. (2) `parameterName` - Name of the new parameter.

```
229 public RefactoringStatus checkInitialConditions() {
230     RefactoringStatus result = new RefactoringStatus();
231     ArrayList<org.eclipse.uml2.uml.Parameter> selectedEObjects =
232       ((UmlParameterList) dataManagement.
233           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue()).
234                               getUmlParameters();
235     // test: the selected parameters must be owned by one single operation
236     String msg = "The selected parameters do not belong to one operation!";
237     ArrayList<Operation> operations = new ArrayList<Operation>();
238     for (Parameter param : selectedEObjects) {
239       Operation op = param.getOperation();
240       if (! operations.contains(op)) operations.add(op);
241     }
242     if (operations.size() > 1) {
243       result.addFatalError(msg);
244       return result;
245     }
246     // test: each selected parameter must be an input parameter
247     msg = "This refactoring can only be applied on input parameters!";
248     for (Parameter param : selectedEObjects) {
249       if (! UmlUtils.isInputParameter(param)) {
250         result.addFatalError(msg);
251         return result;
252       }
253     }
254     // test: the owning operation must be owned by a class
255     msg = "This refactoring can only be applied on parameters whose " +
256         "owning operation is owned by a class!";
257     Operation owningOperation = operations.get(0);
258     if (owningOperation.getClass_() == null) {
259       result.addFatalError(msg);
260       return result;
261     }
262     // test: the owning class must be owned by a package
263     msg = "This refactoring can only be applied on parameters whose " +
264         "owning class is owned by a package!";
265     Class owningClass = owningOperation.getClass_();
266     if (owningClass.getPackage() == null)  result.addFatalError(msg);
267     return result;
268 }
```

Figure F.23: Initial Check Implementation of UML class model refactoring *Introduce Parameter Object*

IMPLEMENTATION Refactoring *Introduce Parameter Object* has been implemented in Java code using the UML2EMF API (for specifying precondition checks) respectively an appropriate model of

the CoMReL language (for specifying the proper model changes) using predefined UML refactorings.

Figure F.23 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`). First, it is checked whether the contextual parameters (named *selectedEObjects*) are owned by the same `Operation` (lines 236–245). Then, lines 247–253 check whether each contextual parameter is an input parameter of the owning operation. Finally, it is checked whether the owning operation of the contextual parameters is owned by a class (lines 255–261) which is in turn owned by a package (lines 263–266). This means, that refactoring *Introduce Parameter Object* can not be applied on parameters of an interface operation, or on operation parameters of an inner class, for example.

```
277 public RefactoringStatus checkFinalConditions() {
278     RefactoringStatus result = new RefactoringStatus();
279     ArrayList<org.eclipse.uml2.uml.Parameter> selectedEObjects =
280         ((UmlParameterList) dataManagement.
281             getInPortByName(dataManagement.SELECTEDEOBJECT).getValue()).
282                                 getUmlParameters();
283     String className =
284       (String) dataManagement.getInPortByName("className").getValue();
285     String parameterName =
286       (String) dataManagement.getInPortByName("parameterName").getValue();
287     // test: there must be no class in the model named as specified in 'className'
288     String msg = "There is already a class in the model named '" + className + "'!";
289     ArrayList<Class> allClasses =
290         UmlUtils.getAllClasses(selectedEObjects.get(0).getModel());
291     for (Class cls : allClasses) {
292       if ((cls.getName() != null) && (cls.getName().equals(className))) {
293         result.addFatalError(msg);
294         break;
295       }
296     }
297     // test: no selected parameter must be named as specified in 'parameterName'
298     msg = "One selected parameter is already named '" + parameterName + "'!";
299     for (Parameter param : selectedEObjects) {
300       if ((param.getName() != null)
301           && (param.getName().equals(parameterName))) {
302         result.addFatalError(msg);
303         break;
304       }
305     }
306     return result;
307 }
```

Figure F.24: Final Check Implementation of UML class model refactoring *Introduce Parameter Object*

Figure F.24 shows the concrete implementation of the final precondition check (Java method `checkFinalConditions()`). Besides the contextual `Parameter` list *selectedEObjects*, this check additionally considers the user input parameters *className* and *parameterName*. The first check ensures whether the model does

not contain a `Class` named as specified in refactoring parameter *className* (lines 288–296). Finally, it is checked whether no parameter of the owning operation of the contextual parameters is named as specified in *parameterName* (lines 298–305).

Figure F.25 shows the concrete CoMReL unit specification of the proper model change of refactoring *Introduce Parameter Object*. This refactoring relies on three atomic model refactorings. The main refactoring unit *introduceparameterobject* is a strict `SequentialUnit` consisting of one `AtomicUnit` and two `SingleQueuedUnits`. The atomic unit creates a new class with contained attributes which correspond to the given parameter list[6]. The queued units replace the contextual parameters with one single parameter in each affected operation of the containing class.

Each `SingleQueuedUnit` contains an `AtomicUnit` calling an already existing refactoring. First, atomic refactoring *Add Parameter* must be applied on each affected operation of the containing class. Then, refactoring *Remove Parameter* must be applied on each corresponding parameter in the affected operations. In both cases, the according *strict* attributes are set to *true*.

To ensure conformity with respect to typing and multiplicity of included ports, the main refactoring unit *introduceparameterobject* requires two helper units. `MultiFeatureUnit` *Get Operations Having Parameters* returns all operations of the owning class of the contextual parameter list which have these parameters in their corresponding parameter lists. Instead of returning the affected operations, `MultiFeatureUnit` *Get Parameters Equal To Parameters* returns the affected parameters of these operations.

TEST CASES The following test cases have been performed:

1. Parameters *p1* and *p2* of the contextual parameter list (*p1*, *p2*, *p3*) are owned by operation *A::op1*; parameter *p3* of the contextual parameter list (*p1*, *p2*, *p3*) is owned by operation *A::op2* ⇒ corresponding error message. ✓
2. Each parameter of the contextual parameter list (*p1*, *p2*, *p3*) is owned by operation *A::op1*; parameter *p1* is a return parameter ⇒ corresponding error message. ✓
3. Each parameter of the contextual parameter list (*p1*, *p2*, *p3*) is owned by operation *Interf1::op1*; *Interf* is an interface ⇒ corresponding error message. ✓
4. Each parameter of the contextual parameter list (*p1*, *p2*, *p3*) is owned by operation *A::op1*; *A* is an inner class of class *B* ⇒ corresponding error message. ✓

6 The atomic refactoring *Create Class with Attributes from Parameter List* is neither described in this document nor in Appendix E. It can be seen as a kind of *helper* refactoring. Nevertheless, it has been implemented in our tool support.

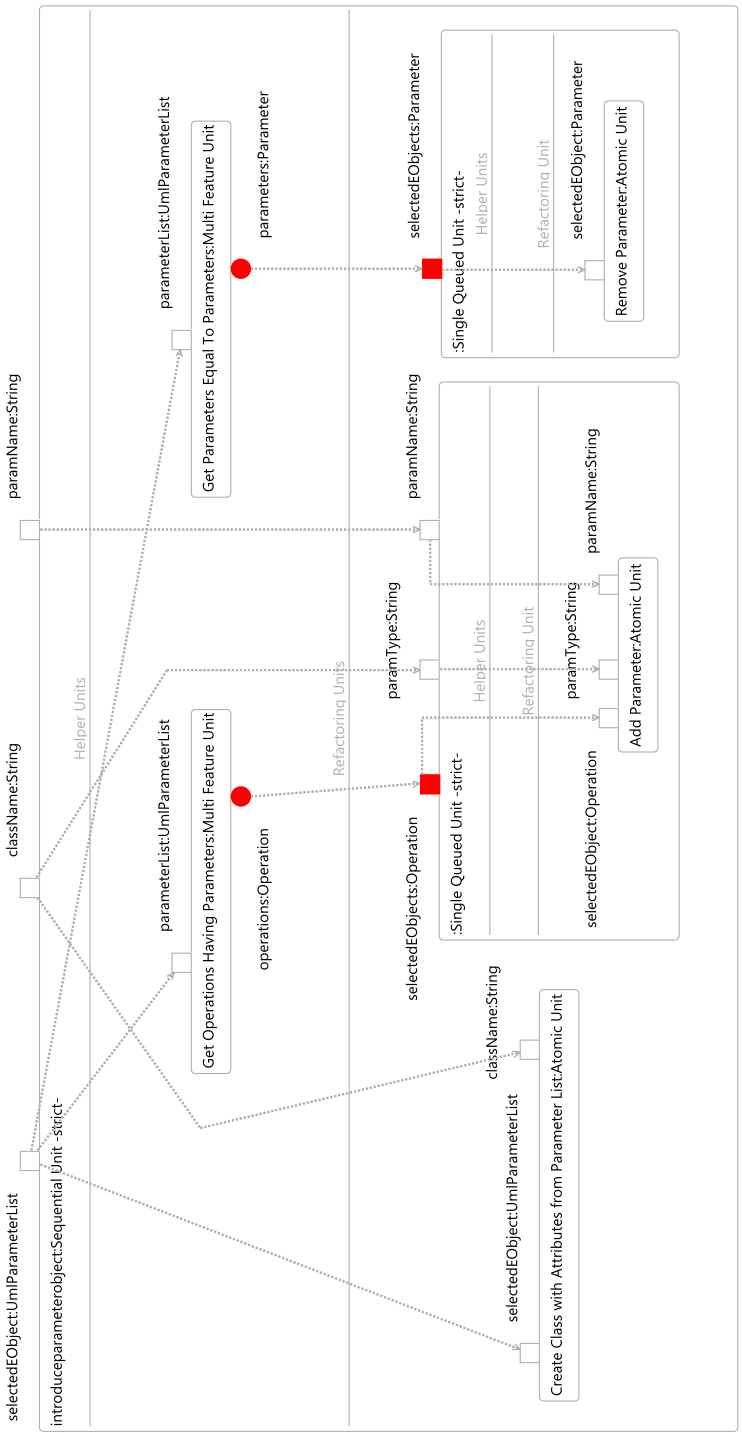


Figure F.25: Model Change Implementation of UML class model refactoring *Introduce Parameter Object*

5. *className* is set to *B*; the model contains a class named *B* ⇒ corresponding error message. ✓
6. *parameterName* is set to *p*; each parameter of the contextual parameter list (*p1*, *p2*, *p3*) is owned by operation *A::op1*; op-

eration *A::op1* has a parameter named $p \Rightarrow$ corresponding error message. ✓

7. *className* is set to *B*; *parameterName* is set to *p*; each parameter of the contextual parameter list (*p1*, *p2*, *p3*) is owned by operation *A::op1*; class *A* has no further operation with the contextual parameter list as sublist; no violated internal precondition $\Rightarrow$ refactoring execution as expected. ✓
8. *className* is set to *B*; *parameterName* is set to *p*; each parameter of the contextual parameter list (*p1*, *p2*, *p3*) is owned by operations *A::op1*, *A::op2*, and *A::op3*; no violated internal precondition $\Rightarrow$ refactoring execution as expected. ✓

F.10 move operation

DESCRIPTION This refactoring moves an operation of a class to an associated class. It is often applied when some class has too much behavior or when classes collaborate too much. In most cases, the visibility of the operation should be the same as before. In some cases, when the operation is *private* or it is moved between classes belonging to different packages, this is not enough. [107, 150]

CONTEXTUAL ELEMENT `Operation`

REFACTORING PARAMETERS `className` - Name of the target class.

IMPLEMENTATION Refactoring *Add Parameter* has been implemented in Java code using the UML2EMF API.

```
166 public RefactoringStatus checkInitialConditions() {
167     RefactoringStatus result = new RefactoringStatus();
168     org.eclipse.uml2.uml.Operation selectedEObject =
169       (org.eclipse.uml2.uml.Operation) dataManagement.
170           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
171     // test: the selected operation must be owned by a class
172     String msg = "This refactoring can only be applied" +
173             " on operations which are owned by a class!";
174     if (selectedEObject.getClass_() == null) {
175       result.addFatalError(msg);
176     } else {
177       // test: the selected operation must be public
178       msg = "This refactoring can only be applied on public operations!";
179       if (! selectedEObject.getVisibility().equals(VisibilityKind.PUBLIC_LITERAL))
180                         result.addFatalError(msg);
181       // test: the owning class must be related to at least one other class
182       // using a bidirectional association with multiplicity 1:1
183       msg = "The owning class is not related to at least one other class " +
184           "using a bidirectional association with multiplicity 1:1 " +
185           "and public association ends!";
186       List<Class> associatedClasses =
187           UmlUtils.getOne2OneAssociatedClasses(selectedEObject.getClass_());
188       if (associatedClasses.isEmpty()) result.addFatalError(msg);
189     }
190     return result;
191 }
```

Figure F.26: Initial Check Implementation of UML class model refactoring *Move Operation*

Figure F.26 shows the concrete implementation of the initial precondition checks (Java method `checkInitialConditions()`). First, it is checked whether the contextual `Operation` (named *selectedEObject*) is owned by a `Class` since this refactoring should not be applied on interface operations. If this precondition is violated, an appropriate error message is returned (lines 172–175). Then, it is checked whether the contextual operation has *public* visibility since only public operations should be moved (lines

178–180). Finally, lines 183–188 check whether the owning class of the contextual operation is related to at least one other class using a bidirectional association with multiplicity 1:1[7].

```
200 public RefactoringStatus checkFinalConditions() {
201     RefactoringStatus result = new RefactoringStatus();
202     org.eclipse.uml2.uml.Operation selectedEObject =
203       (org.eclipse.uml2.uml.Operation) dataManagement.
204           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
205     String className =
206       (String) dataManagement.getInPortByName("className").getValue();
207     // test: the owning class must be related to a class with the specified
208     // name using a bidirectional association with multiplicity 1:1
209     String msg = "The owning class is not related to a class named '" +
210             className + "' using a bidirectional association with " +
211             "multiplicity 1:1 and public association ends!";
212     List<Class> associatedClasses =
213         UmlUtils.getOne2OneAssociatedClasses(selectedEObject.getClass_());
214     if (! UmlUtils.isAssociatedClass(associatedClasses, className)) {
215       result.addFatalError(msg);
216     } else {
217       // test: the associated class must not own an operation with the
218       // same name as the selected operation and a similar parameter list
219       msg = "The associated class already owns an operation named '" +
220           selectedEObject.getName() + "' having the same signature " +
221           "(type and parameter list)!";
222       Class cl = UmlUtils.getAssociatedClass(associatedClasses, className);
223       if (classOwnsOperation(cl, selectedEObject)) result.addFatalError(msg);
224       // test: the associated class must already inherit an operation with the
225       // same name as the selected operation and a similar parameter list
226       msg = "The associated class already inherits an an operation named '" +
227           selectedEObject.getName() + "' having the same signature " +
228           "(type and parameter list)!";
229       if (classInheritsOperation(cl, selectedEObject)) result.addFatalError(msg);
230     }
231     return result;
232 }
```

Figure F.27: Final Check Implementation of UML class model refactoring *Move Operation*

Figure F.27 shows the concrete implementation of the final precondition check (Java method `checkFinalConditions()`). Besides the contextual `Operation` *selectedEObject*, this check additionally considers the user input parameter *className*. First, it is checked whether the owning class of the contextual operation is related to a `Class` named as specified in parameter *className* by a bidirectional `Association` with 1:1 multiplicity (lines 209–215). Furthermore, it is checked whether this associated class does not already own (lines 219–223) or inherit (lines 226–229) an operation with the same name and a similar parameter list as the contextual operation.

Figure F.28 shows the concrete implementation of the proper model change of refactoring *Move Operation* (method `run()`).

7 Please note that the concrete checks are implemented as static methods of a utility class in order to avoid redundant code and to reuse it as often as possible.

```
126 public void run() {
127   org.eclipse.uml2.uml.Operation selectedEObject =
128     (org.eclipse.uml2.uml.Operation) dataManagement.
129         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
130   String className =
131     (String) dataManagement.getInPortByName("className").getValue();
132   // execute: move selected operation to specified class
133   List<Class> associatedClasses =
134       UmlUtils.getOne2OneAssociatedClasses(selectedEObject.getClass_());
135   Class newClass = UmlUtils.getAssociatedClass(associatedClasses, className);
136   Class oldClass = selectedEObject.getClass_();
137   oldClass.getOwnedOperations().remove(selectedEObject);
138   newClass.getOwnedOperations().add(selectedEObject);
139 }
```

Figure F.28: Model Change Implementation of UML class model refactoring *Move Operation*

First, the appropriate associated `Class` is determined by using input parameter *className*. Then, the contextual attribute (*selectedEObject*) is removed from its containing class (line 137) and inserted to the corresponding attributes list of the associated class (line 138).

TEST CASES The following test cases have been performed:

1. The contextual operation *op* has *private* visibility ⇒ corresponding error message. ✓
2. The contextual operation *op* is owned by an interface *Interf1* ⇒ corresponding error message. ✓
3. The owning class *A* of the contextual operation *op* is not associated to other classes ⇒ corresponding error message. ✓
4. The owning class *A* of the contextual operation *op* has one outgoing association *assoc* but no incoming associations ⇒ corresponding error message. ✓
5. The owning class *A* of the contextual operation *op* has one incoming association *assoc* but no outgoing associations ⇒ corresponding error message. ✓
6. The owning class *A* of the contextual operation *op* has one outgoing association *assoc1* to class *B* and one outgoing association *assoc2* to class *C* but no further associations ⇒ corresponding error message. ✓
7. The owning class *A* of the contextual operation *op* has a bidirectional association *assoc* to class *B* with multiplicity 1:0..* and no further associations ⇒ corresponding error message. ✓
8. The owning class *A* of the contextual operation *op* has a bidirectional association *assoc* to class *B* with multiplicity

1..*:1 and no further associations ⇒ corresponding error message. ✓

9. The owning class *A* of the contextual operation *op* has a bidirectional association *assoc* to class *B* with multiplicity 1..*:0..* and no further associations ⇒ corresponding error message. ✓
10. The owning class *A* of the contextual operation *op* has a bidirectional 1:1 association *assoc* to class *B*; the opposite association end has *private* visibility ⇒ corresponding error message. ✓
11. *className* is set to *B*; the owning class *A* of the contextual operation *op* has no associations to a class named *B* ⇒ corresponding error message. ✓
12. *className* is set to *B*; the owning class *A* of the contextual operation *op* has an incoming association *assoc* from a class named *B* but no outgoing association to it ⇒ corresponding error message. ✓
13. *className* is set to *B*; the owning class *A* of the contextual operation *op* has an outgoing association *assoc* to a class named *B* but no incoming associations from it ⇒ corresponding error message. ✓
14. *className* is set to *B*; the owning class *A* of the contextual operation *op* has a bidirectional association *assoc* to class *B* with multiplicity 1:0..* ⇒ corresponding error message. ✓
15. *className* is set to *B*; the owning class *A* of the contextual operation *op* has a bidirectional association *assoc* to class *B* with multiplicity 1..*:1 ⇒ corresponding error message. ✓
16. *className* is set to *B*; the owning class *A* of the contextual operation *op* has a bidirectional association *assoc* to class *B* with multiplicity 1..*:0..* ⇒ corresponding error message. ✓
17. *className* is set to *B*; the owning class *A* of the contextual operation *op* has a bidirectional 1:1 association *assoc* to class *B*; the opposite association end has *private* visibility ⇒ corresponding error message. ✓
18. *className* is set to *B*; the owning class *A* of the contextual operation *op* has a bidirectional 1:1 association *assoc* to class *B*; the opposite association end has *public* visibility; *B* owns an operation named *op* with a similar parameter list as the contextual operation *op* ⇒ corresponding error message. ✓
19. *className* is set to *B*; the owning class *A* of the contextual operation *op* has a bidirectional 1:1 association *assoc* to class

B; the opposite association end has *public* visibility; *B* inherits an operation named *op* with a similar parameter list as the contextual operation *op* ⇒ corresponding error message. ✓

20. *className* is set to *B*; the owning class *A* of the contextual operation *op* has a bidirectional 1:1 association *assoc* to class *B*; the opposite association end has *public* visibility; *B* does not own an operation named *op* with a similar parameter list as the contextual operation *op* ⇒ refactoring execution as expected. ✓
21. *className* is set to *B*; the owning class *A* of the contextual operation *op* has a bidirectional 1:1 association *assoc1* to class *B* as well as a bidirectional 1:1 association *assoc2* to class *C*; the opposite association end has *public* visibility; *B* does not own an operation named *op* with a similar parameter list as the contextual operation *op* ⇒ refactoring execution as expected. ✓

F.11 move property

DESCRIPTION A property (attribute) is better placed in another class which is associated to this class. This refactoring moves this property to the associated class. In most cases, the visibility of the property should be the same as before. In some cases, when the property is *private* or it is moved between classes belonging to different packages, this is not enough [107, 150, 93]

CONTEXTUAL ELEMENT `Property`

REFACTORING PARAMETERS `className` - Name of the target class.

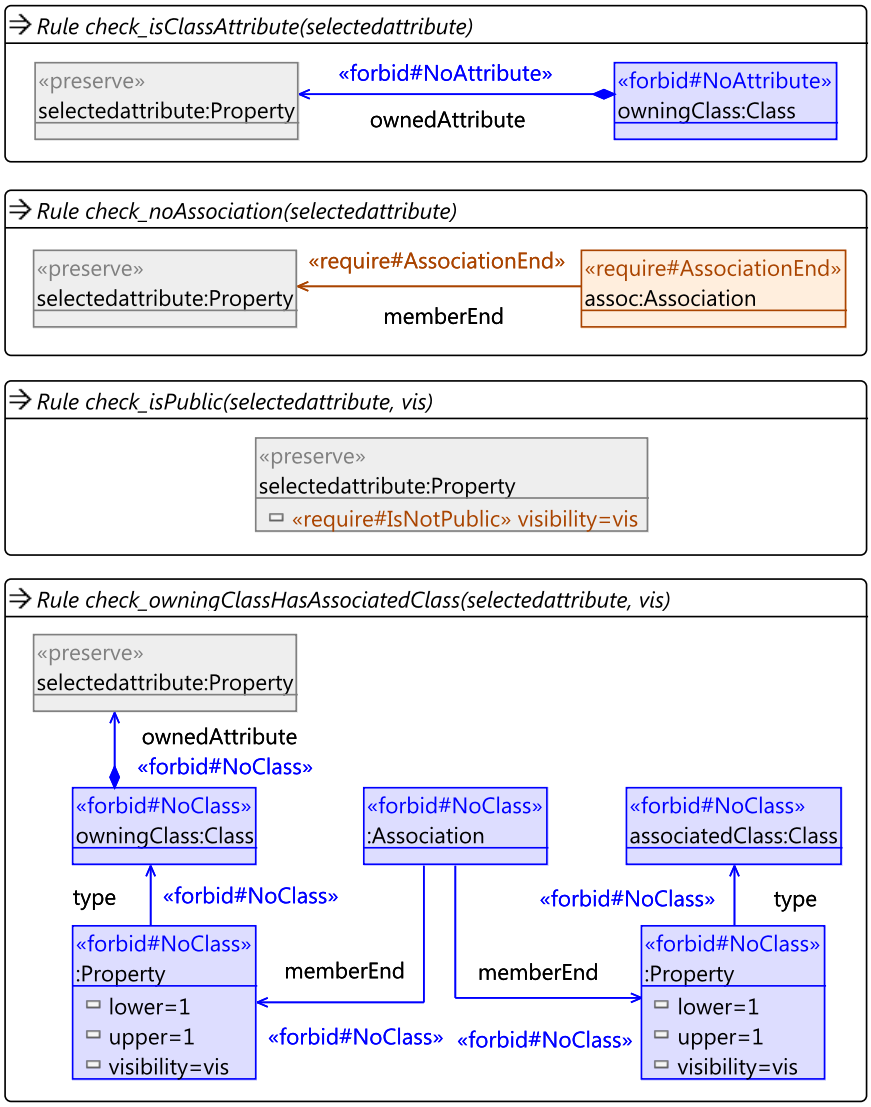


Figure F.29: Initial Check Implementation of UML class model refactoring *Move Property*

IMPLEMENTATION Refactoring *Move Property* has been implemented in Henshin pattern rules (for specifying precondition checks)

respectively a Henshin transformation rule (for specifying the proper model changes) using the abstract syntax of UML.

Figure F.29 shows the Henshin pattern rule specifications of the initial precondition check. Rule *check_isClassAttribute* specifies the violated precondition that the contextual `Property` (*selectedattribute*) is no class attribute (NAC *NoAttribute*). Rule *check_no Association* defines the situation that the contextual attribute represents an association end (specified by PAC *AssociationEnd*), i.e., this refactoring can be applied on attributes only which are not an end of an appropriate association. Rule *check_isPublic* defines the violated precondition that the contextual attribute has a visibility different from *public*. Here, an additional internal rule parameter *vis* is used in combination with parameter condition `vis != public`. Finally, rule *check_owningClassHasAssociatedClass* ensures that the owning class of the contextual attribute is related to at least one other `Class` (node *associatedclass* in Figure F.29) by an `Association` with 1:1 multiplicity and a public opposite association end.

Figure F.30 shows the Henshin pattern rule specifications of the final precondition check. Besides the contextual `Property` *selectedattribute*, this check additionally considers the user input parameter *classname*. The upper rule ensures that the owning class of the contextual attribute is related to a `Class` named as specified in parameter *classname* (node *associatedclass* in Figure F.30) by an `Association` with 1:1 multiplicity and a public opposite association end. The other rules model the erroneous situations that this associated class already owns (rule *check_ownsNoAttribute*) or inherits (henshin rule *check_inheritsNo Attribute*) an attribute with the same name as the contextual one already. Both rules use an internal parameter *attributename* each to specify the equality of the corresponding attribute names.

Figure F.31 on page 294 shows the Henshin pattern rule specification of the proper model change of refactoring *Move Property*. This rule determines the appropriate associated `Class` by using input parameter *classname*, removes the contextual attribute (*selectedattribute*) from its containing class and inserts it to the corresponding attributes list of the associated class.

TEST CASES The following test cases have been performed:

1. The contextual attribute *att* has *private* visibility ⇒ corresponding error message. ✓
2. The contextual attribute *att* is an end of association *assoc* ⇒ corresponding error message. ✓
3. The contextual attribute *att* is owned by an interface *Interf1* ⇒ corresponding error message. ✓

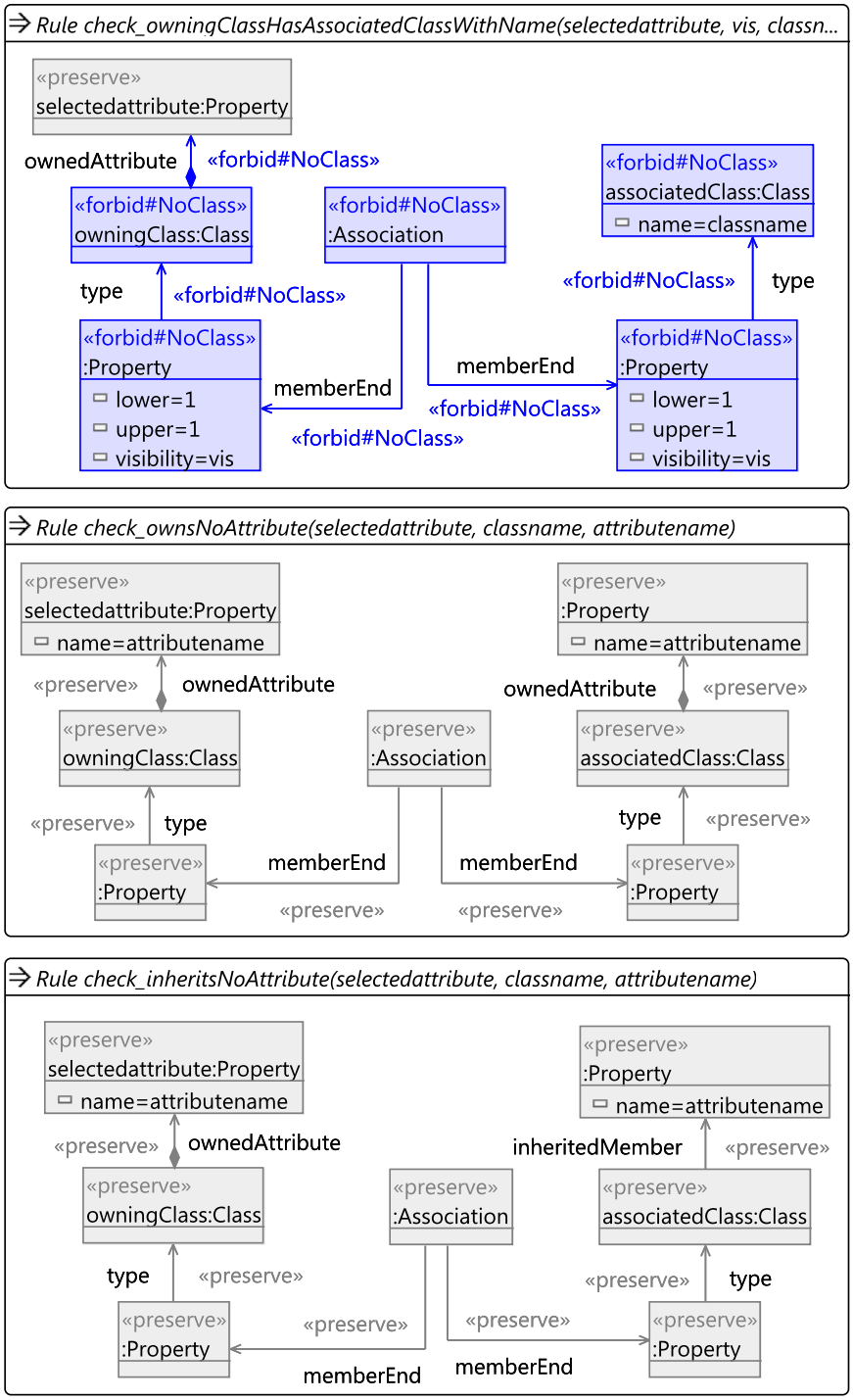


Figure F.30: Final Check Implementation of UML class model refactoring *Move Property*

4. The owning class A of the contextual attribute att is not associated to other classes $\Rightarrow$ corresponding error message. ✓

5. The owning class A of the contextual attribute *att* has one outgoing association *assoc* but no incoming associations ⇒ corresponding error message. ✓
6. The owning class A of the contextual attribute *att* has one incoming association *assoc* but no outgoing associations ⇒ corresponding error message. ✓

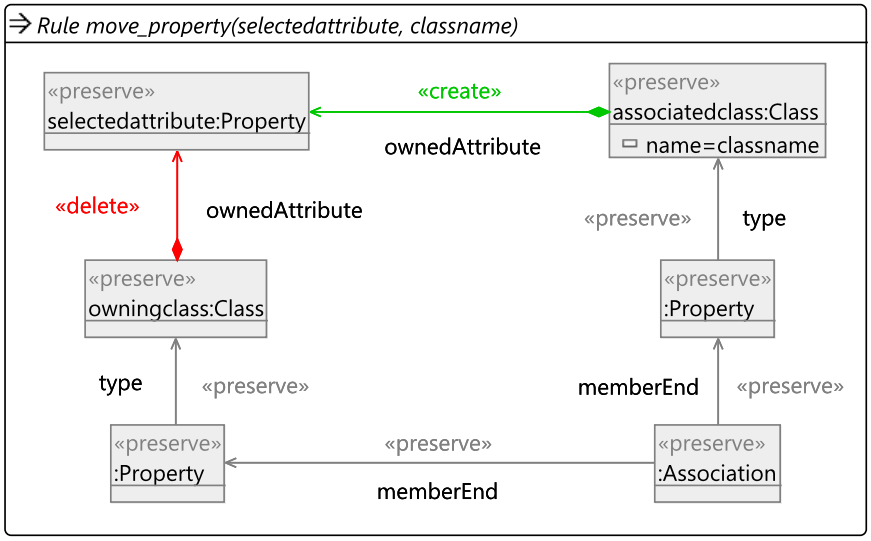


Figure F.31: Model Change Implementation of UML class model refactoring *Move Property*

7. The owning class A of the contextual attribute *att* has one outgoing association *assoc1* to class B and one outgoing association *assoc2* to class C but no further associations ⇒ corresponding error message. ✓
8. The owning class A of the contextual attribute *att* has a bidirectional association *assoc* to class B with multiplicity 1:0..* and no further associations ⇒ corresponding error message. ✓
9. The owning class A of the contextual attribute *att* has a bidirectional association *assoc* to class B with multiplicity 1..*:1 and no further associations ⇒ corresponding error message. ✓
10. The owning class A of the contextual attribute *att* has a bidirectional association *assoc* to class B with multiplicity 1..*:0..* and no further associations ⇒ corresponding error message. ✓
11. The owning class A of the contextual attribute *att* has a bidirectional 1:1 association *assoc* to class B; the opposite

association end has *private* visibility ⇒ corresponding error message. ✓

12. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has no associations to a class named *B* ⇒ corresponding error message. ✓
13. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has an incoming association *assoc* from a class named *B* but no outgoing association to it ⇒ corresponding error message. ✓
14. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has an outgoing association *assoc* to a class named *B* but no incoming associations from it ⇒ corresponding error message. ✓
15. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a bidirectional association *assoc* to class *B* with multiplicity 1:0..* ⇒ corresponding error message. ✓
16. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a bidirectional association *assoc* to class *B* with multiplicity 1..*:1 ⇒ corresponding error message. ✓
17. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a bidirectional association *assoc* to class *B* with multiplicity 1..*:0..* ⇒ corresponding error message. ✓
18. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a bidirectional 1:1 association *assoc* to class *B*; the opposite association end has *private* visibility ⇒ corresponding error message. ✓
19. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a bidirectional 1:1 association *assoc* to class *B*; the opposite association end has *public* visibility; *B* owns an attribute named *att* ⇒ corresponding error message. ✓
20. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a bidirectional 1:1 association *assoc* to class *B*; the opposite association end has *public* visibility; *B* inherits an attribute named *att* ⇒ corresponding error message. ✓
21. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a bidirectional 1:1 association *assoc* to class *B*; the opposite association end has *public* visibility; *B* does not own an attribute named *att* ⇒ refactoring execution as expected. ✓
22. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a bidirectional 1:1 association *assoc1* to class *B* as well as a bidirectional 1:1 association *assoc2* to class

C; the opposite association end has *public* visibility; *B* does not own an attribute named *att* $\Rightarrow$ refactoring execution as expected. ✓

F.12 pull up operation

DESCRIPTION This refactoring pulls an operation of a class to its superclass. Usually it is used simultaneously on several classes which inherit from the same superclass. The aim of this refactoring is often to extract identical operations. In some cases, when the property is *private* or it is moved between classes belonging to different packages, this is not enough. [137, 107, 150]

CONTEXTUAL ELEMENT `Operation`

REFACTORING PARAMETERS `className` - Name of the superclass the contextual operation has to be pulled up to.

IMPLEMENTATION Refactoring *Pull Up Operation* has been implemented in Java code using the UML2EMF API.

```
181 public RefactoringStatus checkInitialConditions() {
182     RefactoringStatus result = new RefactoringStatus();
183     org.eclipse.uml2.uml.Operation selectedEObject =
184       (org.eclipse.uml2.uml.Operation) dataManagement.
185           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
186     // test: the selected operation must be owned by a class
187     String msg = "This refactoring can only be applied" +
188             " on operations which are owned by a class!";
189     if (selectedEObject.getClass_() == null) {
190       result.addFatalError(msg);
191     } else {
192       // test: the selected operation must be public
193       msg = "This refactoring can only be applied on public operations!";
194       if (! selectedEObject.getVisibility().equals(VisibilityKind.PUBLIC_LITERAL))
195                               result.addFatalError(msg);
196       // test: the owning class must have at least one superclass
197       msg = "This refactoring can not be applied because the owning class " +
198           "of the selected operation does not have any superclasses!";
199       if (selectedEObject.getClass_().getSuperClasses().isEmpty())
200                               result.addFatalError(msg);
201     }
202     return result;
203 }
```

Figure F.32: Initial Check Implementation of UML class model refactoring *Pull Up Operation*

Figure F.32 shows the concrete implementation of the initial precondition checks (Java method `checkInitialConditions()`). First, it is checked whether the contextual `Operation` (named *selectedEObject*) is owned by a `Class` since this refactoring should not be applied on interface operations. If this precondition is violated, an appropriate error message is returned (lines 187–190). Then, it is checked whether the contextual operation has *public* visibility since only public operations should be pulled up to the specified superclass (lines 193–195). Finally, lines 197–201 check whether the owning class of the contextual operation has at least one superclass.

```
212 public RefactoringStatus checkFinalConditions() {
213     RefactoringStatus result = new RefactoringStatus();
214     org.eclipse.uml2.uml.Operation selectedEObject =
215       (org.eclipse.uml2.uml.Operation) dataManagement.
216           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
217     String className =
218       (String) dataManagement.getInPortByName("className").getValue();
219     // test: the owning class must have a superclass with the specified name
220     String msg = "The owning class does not have a superclass named '" +
221                                         className + "'!";
222     Class superClass = selectedEObject.getClass_().getSuperClass(className);
223     if (superClass == null) {
224       result.addFatalError(msg);
225     } else {
226       // test: each subclass of the specified superclass must have an
227       // operation equal to the selected operation
228       if (! UmlUtils.subClassesHaveOperation(superClass, selectedEObject)) {
229         ArrayList<String> msgs =
230             UmlUtils.getReasonsWhySubClassesDoNotHaveOperation
231                                                 (superClass, selectedEObject);
232         for (String str : msgs) {
233           result.addFatalError(str);
234         }
235       }
236       // test: the superclass must not own an operation with the
237       // same name as the selected operation and a similar parameter list
238       msg = "Class '" + className + "' already owns an operation named '" +
239           selectedEObject.getName() + "' having the same signature " +
240           "(type and parameter list)!";
241       if (classOwnsOperation(superClass, selectedEObject))
242         result.addFatalError(msg);
243       // test: the superclass must not inherit an operation with the
244       // same name as the selected operation and a similar parameter list
245       msg = "Class '" + className + "' already inherits an operation named '" +
246           selectedEObject.getName() + "' having the same signature " +
247           "(type and parameter list)!";
248       if (classInheritsOperation(superClass, selectedEObject))
249         result.addFatalError(msg);
250     }
251     return result;
252 }
```

Figure F.33: Final Check Implementation of UML class model refactoring *Pull Up Operation*

Figure F.33 shows the concrete implementation of the final precondition check (Java method `checkFinalConditions()`). Besides the contextual `Operation` *selectedEObject*, this check additionally considers the user input parameter *className*. First, it is checked whether the owning class of the contextual operation has a superclass named as specified in parameter *className* (lines 220–224). Then, it is checked whether each subclass of this superclass has an operation that is equivalent to the contextual one (line 228–233)[8]. Finally, it is checked whether the specified superclass does not already own (lines 238–242) or inherit (lines 245–249)

8 Please note that the concrete checks are implemented as static methods of a utility class in order to avoid redundant code and to reuse it as often as possible.

an operation with the same name and a similar parameter list as the contextual operation.

```
126  public void run() {
127    org.eclipse.uml2.uml.Operation selectedEObject =
128      (org.eclipse.uml2.uml.Operation) dataManagement.
129          getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
130    String className =
131      (String) dataManagement.getInPortByName("className").getValue();
132    // execute: move selected attribute to specified superclass
133    Class oldClass = selectedEObject.getClass_();
134    Class newClass = oldClass.getSuperClass(className);
135    oldClass.getOwnedOperations().remove(selectedEObject);
136    newClass.getOwnedOperations().add(selectedEObject);
137    // execute: remove equivalent operations from subclasses
138    ArrayList<Class> classes = UmlUtils.getAllSubClasses(newClass);
139    classes.remove(oldClass);
140    for (Class cls : classes) {
141      Operation operationToRemove = null;
142      for (Operation op : cls.getOwnedOperations()) {
143        if (UmlUtils.haveSameNames(op, selectedEObject)
144            && UmlUtils.haveSameType(op, selectedEObject)
145            && UmlUtils.haveSameSignatures(op, selectedEObject)) {
146          operationToRemove = op;
147          break;
148        }
149      }
150      // remove equivalent operation from subclass
151      Class owningSubClass = operationToRemove.getClass_();
152      owningSubClass.getOwnedOperations().remove(operationToRemove);
153    }
154  }
```

Figure F.34: Model Change Implementation of UML class model refactoring *Pull Up Operation*

Figure F.34 shows the concrete implementation of the proper model change of refactoring *Pull Up Operation* (method `run()`). First, the contextual operation (*selectedEObject*) is removed from its containing class (line 135) and inserted to the corresponding operations list of the superclass class (line 136) named as specified in parameter *className*. Then, the corresponding operations which are equal to the contextual one are removed from each subclass of the specified superclass (lines 138–152).

TEST CASES The following test cases have been performed:

1. The contextual operation *op* is owned by an interface *Interf1* ⇒ corresponding error message. ✓
2. The contextual operation *op* has *protected* visibility ⇒ corresponding error message. ✓
3. The owning class *A* of the contextual operation *op* has no superclass ⇒ corresponding error message. ✓
4. *classname* is set to *B*; the owning class *A* of the contextual operation *op* has no superclass named *B* ⇒ corresponding error message. ✓

5. *classname* is set to *B*; the owning class *A* of the contextual operation *op* has a superclass named *B*; subclass *C* of class *B* does not own an operation *op* ⇒ corresponding error message. ✓
6. *classname* is set to *B*; the owning class *A* of the contextual operation *op* (type *Integer*, arbitrary input parameter list) has a superclass named *B*; subclass *C* of class *B* owns an operation *op* with the same parameter list as the contextual operation *op* and with type *String*; *C* owns no further operations ⇒ corresponding error message. ✓
7. *classname* is set to *B*; the owning class *A* of the contextual operation *op* (type *Integer*, arbitrary input parameter list) has a superclass named *B*; subclass *C* of class *B* owns an operation *op* with a different parameter list as the contextual operation *op* and with type *Integer*; *C* owns no further operations ⇒ corresponding error message. ✓
8. *classname* is set to *B*; the owning class *A* of the contextual operation *op* (type *Integer*, arbitrary input parameter list; multiplicity 1) has a superclass named *B*; subclass *C* of class *B* owns an operation *op* with the same parameter list as the contextual operation *op* and with type *Integer* and multiplicity 0..*; *C* owns no further operations ⇒ corresponding error message. ✓
9. *classname* is set to *B*; the owning class *A* of the contextual operation *op* (type *Integer*, arbitrary input parameter list; multiplicity 1, visibility *public*) has a superclass named *B*; subclass *C* of class *B* owns an operation *op* with the same parameter list as the contextual operation *op* and with type *Integer*, multiplicity 0..*, and visibility *private*; *C* owns no further operations ⇒ corresponding error message. ✓
10. *classname* is set to *B*; the owning class *A* of the contextual operation *op* has a superclass named *B*; *B* has an operation *op* with an equal parameter list as the contextual operation *op* ⇒ corresponding error message. ✓
11. *classname* is set to *B*; the owning class *A* of the contextual operation *op* has a superclass named *B*; *B* inherits an operation *op* with an equal parameter list as the contextual operation *op* ⇒ corresponding error message. ✓
12. *classname* is set to *B*; the owning class *A* of the contextual operation *op* has a superclass named *B*; subclasses *C* and *D* of class *B* own equal operations *op* ⇒ refactoring execution as expected. ✓

F.13 pull up property

DESCRIPTION This refactoring removes one property (attribute) from a class or a set of classes and inserts it to one of its superclasses. In most cases, the visibility of the property should be the same as before. In some cases, when the property is *private* or it is moved between classes belonging to different packages, this is not enough. [14, 107, 30, 150]

CONTEXTUAL ELEMENT `Property`

REFACTORING PARAMETERS `className` - Name of the superclass the contextual property has to be pulled up to.

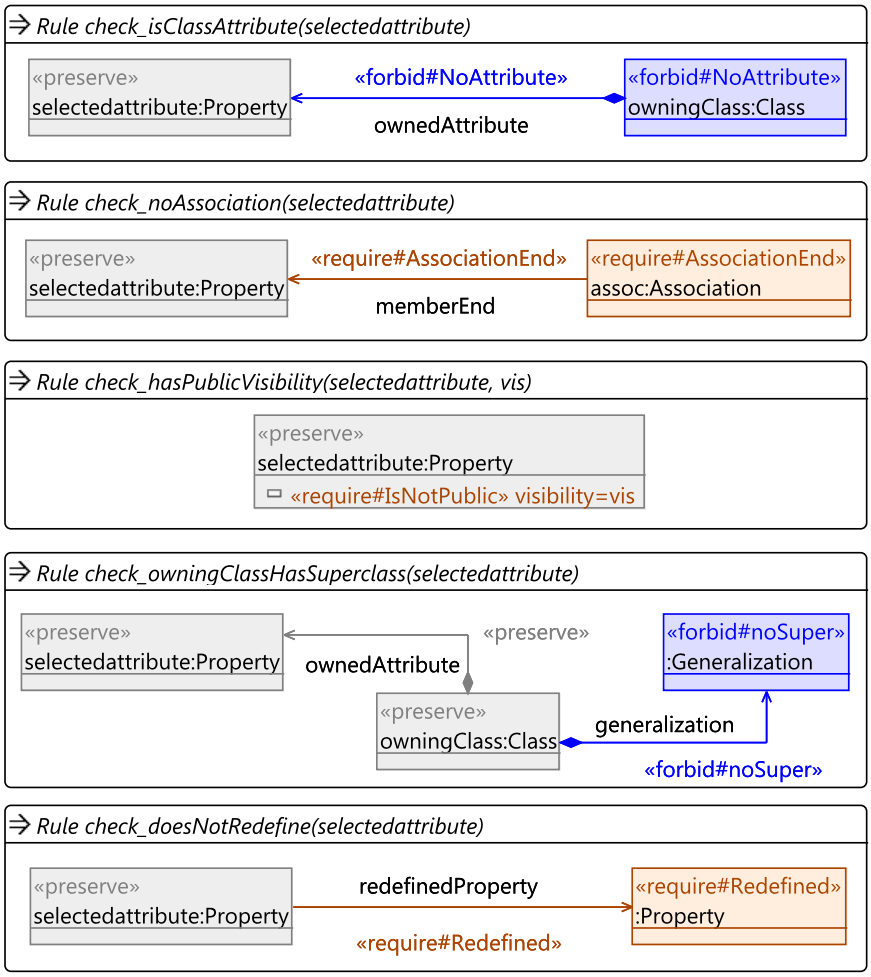


Figure F.35: Initial Check Implementation of UML class model refactoring *Pull Up Property*

IMPLEMENTATION UML class model refactoring *Pull Up Property* has been implemented in Henshin pattern rules (for specifying precondition checks) respectively Henshin transformation rules

(for specifying the proper model changes) using the abstract syntax of UML.

Figure F.35 shows the Henshin pattern rule specifications of the initial precondition checks. Rule *check_isClassAttribute* specifies the violated precondition that the contextual `Property` (*selectedattribute*) is no class attribute (NAC *NoAttribute*). Rule *check_is NoAssociationEnd* defines the situation that the contextual attribute represents an association end (specified by PAC *AssociationEnd*), i.e., this refactoring can be applied on attributes only which are not an end of an appropriate association. Rule *check_ hasPublicVisibility* defines the violated precondition that the contextual attribute has a visibility different from *public*. Here, an additional internal rule parameter *vis* is used in combination with parameter condition `vis!=public` rule *check_ owningClass HasSuperclass* ensures that the owning class of the contextual attribute has a `Generalization` relationship to at least one other `Class`. Finally, pattern rule *check_doesNotRedefine* checks whether the contextual attribute redefines another one in the inheritance hierarchy of the corresponding classes what also represents a violated precondition of this refactoring.

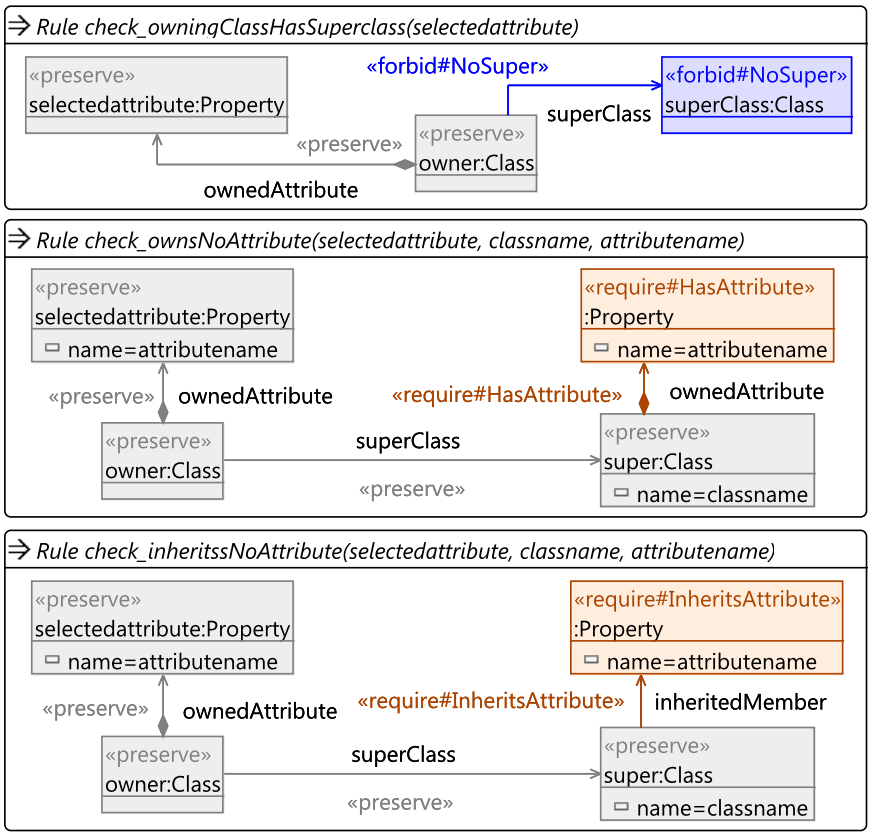


Figure F.36: Final Check Implementation of UML class model refactoring *Pull Up Property* (1)

Figures F.36 to F.38 shows the Henshin pattern rule specifications of the final precondition checks. Besides the contextual `Property` *selectedattribute*, these checks additionally consider the user input parameter *classname*. Altogether nine violated preconditions are modeled. The upper rule in Figure F.36 ensures that the owning class of the contextual attribute has a superclass named as specified in parameter *classname* (node *superclass*). The following two rules model the erroneous situations that this superclass already owns (rule *check_ownsNoAttribute*) or inherits (rule *check_inheritsNoAttribute*) an attribute with the same name as the contextual one already. Here, the rules use an internal parameter *attributename* each to specify the equality of the corresponding attribute names.

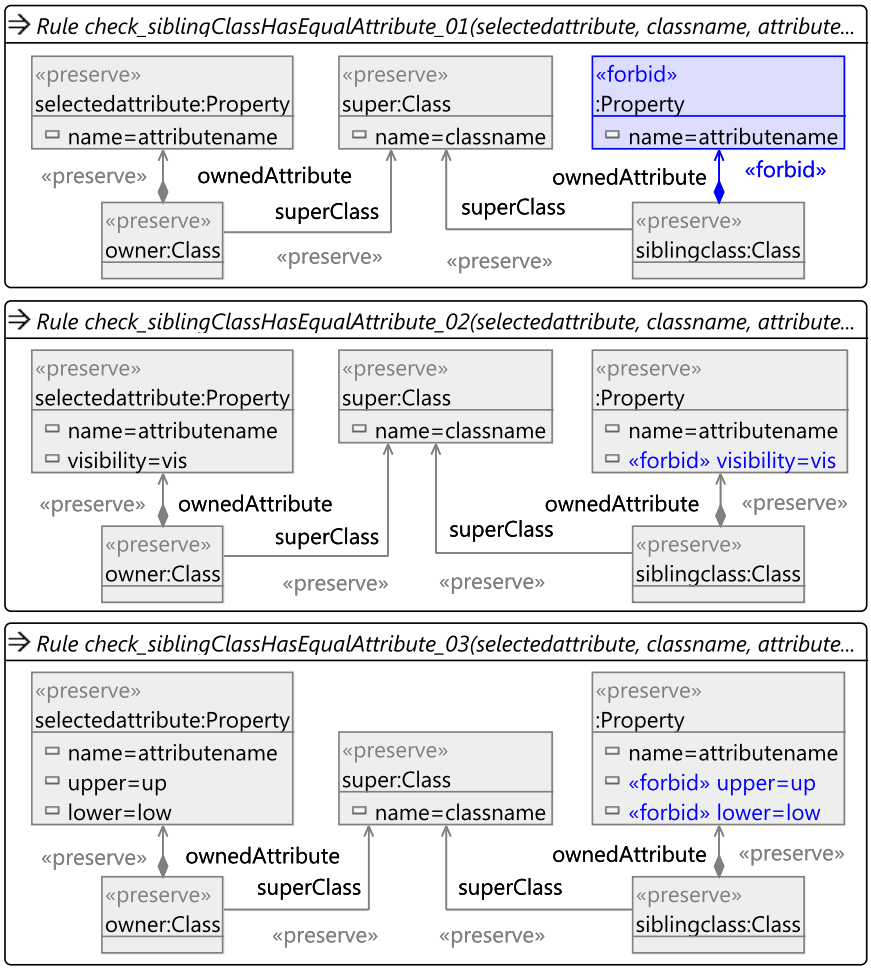


Figure F.37: Final Check Implementation of UML class model refactoring *Pull Up Property* (2)

The remaining rules check whether each subclass of the specified superclass has an attribute that is equivalent to the contextual one. Again, each rule uses an internal parameter *attribute-*

name to specify the equality of the attribute names. The corresponding violated preconditions are:

1. Rule *check_siblingClassHasEqualAttribute_01*: At least one subclass of the superclass named as specified in parameter *classname* does not have an attribute with the same name as the contextual attribute.
2. Rule *check_siblingClassHasEqualAttribute_02*: At least one subclass of the superclass named as specified in parameter *classname* has an attribute with the same name as the contextual attribute but not of equal visibility. Here, rule parameter *vis* is used in the corresponding NAC to specify this unequality.

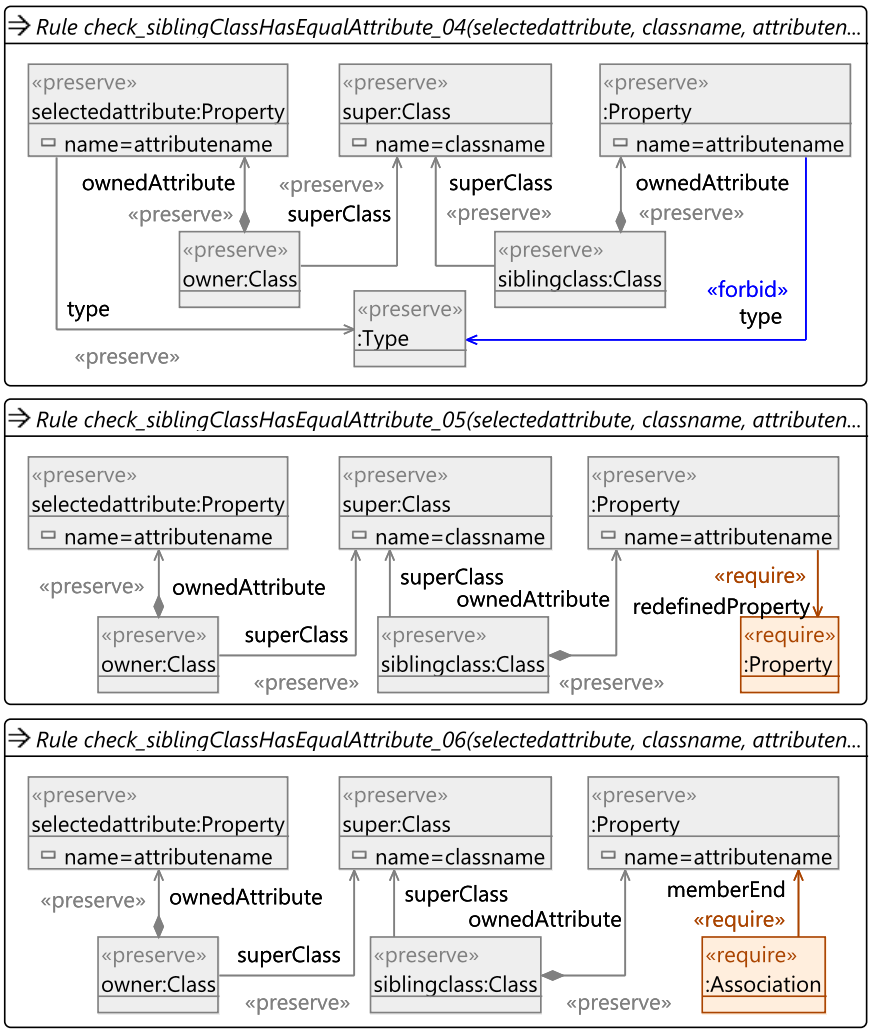


Figure F.38: Final Check Implementation of UML class model refactoring *Pull Up Property* (3)

3. Rule *check_siblingClassHasEqualAttribute_03*: At least one subclass of the superclass named as specified in parameter

classname has an attribute with the same name as the contextual attribute but not of equal multiplicity. Here, rule parameters *upp* and *low* are used in the corresponding NAC to specify this inequality.

4. Rule *check_siblingClassHasEqualAttribute_04*: At least one subclass of the superclass named as specified in parameter *classname* has an attribute with the same name as the contextual attribute but not of equal `Type`.
5. Rule *check_siblingClassHasEqualAttribute_05*: At least one subclass of the superclass named as specified in parameter *classname* has an attribute with the same name as the contextual attribute but redefining another `Property`.
6. Rule *check_siblingClassHasEqualAttribute_06*: At least one subclass of the superclass named as specified in parameter *classname* has an attribute with the same name as the contextual attribute but involved in an `Association` as association end.

Figure F.39 shows the Henshin pattern rule specifications of the proper model changes of refactoring *Pull Up Property*. Rule *pullup* determines the appropriate superclass by using input parameter *classname*, removes the contextual attribute (*selectedattribute*) from its owning class and inserts it to the corresponding attributes list of the superclass. Rule *delete* determines the sibling classes of the contextual class with respect to the specified superclass as well as the corresponding attribute (using the internal rule parameter *attributename*) and removes this attribute from its owning class. To remove these attribute from each sibling class, we used the Henshin control structures as described in [4].

TEST CASES The following test cases have been performed:

1. The contextual attribute *att* is owned by an interface *Interf1* ⇒ corresponding error message. ✓
2. The contextual attribute *att* has *private* visibility ⇒ corresponding error message. ✓
3. The contextual attribute *att* is an end of association *assoc* ⇒ corresponding error message. ✓
4. The contextual attribute *att* redefines another attribute *superatt* ⇒ corresponding error message. ✓
5. The owning class *A* of the contextual attribute *att* has no superclass ⇒ corresponding error message. ✓
6. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has no superclass named *B* ⇒ corresponding error message. ✓

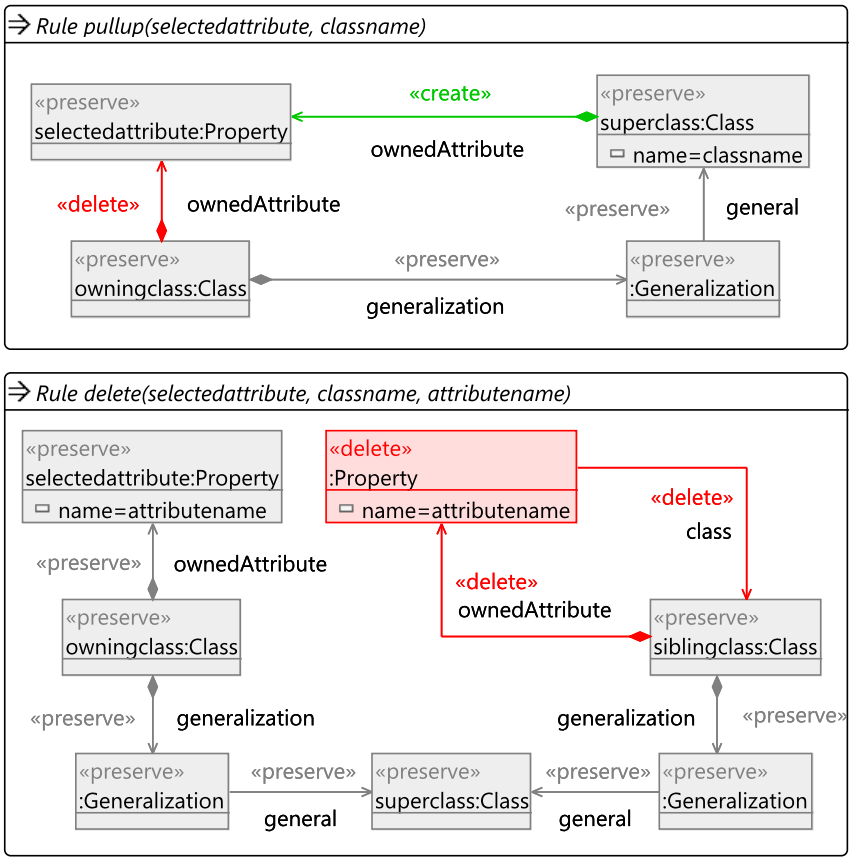


Figure F.39: Model Change Implementation of UML class model refactoring *Pull Up Property*

7. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a superclass named *B*; subclass *C* of class *B* does not own an attribute *att* ⇒ corresponding error message. ✓
8. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a superclass named *B*; subclass *C* of class *B* owns an attribute *att* with *private* visibility ⇒ corresponding error message. ✓
9. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* (multiplicity 1) has a superclass named *B*; subclass *C* of class *B* owns an attribute *att* with multiplicity 0..* ⇒ corresponding error message. ✓
10. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* (type *Integer*) has a superclass named *B*; subclass *C* of class *B* owns an attribute *att* with type *String* ⇒ corresponding error message. ✓

11. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a superclass named *B*; subclass *C* of class *B* owns an attribute *att* that redefines another attribute *superatt*. ⇒ corresponding error message. ✓
12. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a superclass named *B*; subclass *C* of class *B* owns an attribute *att* that is an end of association *assoc* ⇒ corresponding error message. ✓
13. *classname* is set to *B*; the owning class *A* of the contextual attribute *att* has a superclass named *B*; subclasses *C* and *D* of class *B* own equal attributes *att* ⇒ refactoring execution as expected. ✓

F.14 push down operation

DESCRIPTION This refactoring pushes an operation from the owning class down to all its subclasses. If it makes sense, the operation can be removed from some of these afterwards. Sometimes, it also makes sense to keep an operation in all subclasses to hide it from the superclass. [107, 150, 93]

CONTEXTUAL ELEMENT `Operation`

REFACTORING PARAMETERS This refactoring does not have any more parameters.

```
public RefactoringStatus checkInitialConditions() {
    RefactoringStatus result = new RefactoringStatus();
    org.eclipse.uml2.uml.Operation selectedEObject =
      (org.eclipse.uml2.uml.Operation) dataManagement.
          getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
    // test: the selected operation must be owned by a class
    String msg = "This refactoring can only be applied" +
            " on operations which are owned by a class!";
    if (selectedEObject.getClass_() == null) {
      result.addFatalError(msg);
    } else {
      // test: the selected operation must be public
      msg = "This refactoring can only be applied on public operations!";
      if (! selectedEObject.getVisibility().equals(VisibilityKind.PUBLIC_LITERAL))
                              result.addFatalError(msg);
      // test: the owning class must have at least one subclass
      msg = "This refactoring can not be applied because the owning class " +
          "of the selected operation does not have any subclasses!";
      if (! UmlUtils.hasSubclasses(selectedEObject.getClass_()))
                              result.addFatalError(msg);
      // test: the owning class must not be used as attribute type
      msg = "This refactoring can not be applied because the owning class " +
          "of the selected operation is used as attribute type!";
      if (UmlUtils.isUsedAsAttributeType(selectedEObject.getClass_()))
                              result.addFatalError(msg);
      // test: the owning class must not have any subclasses which own
      // operations with the same name, type and parameter list
      msg = "This refactoring can not be applied because at least one subclass " +
          "already owns an operation named '" + selectedEObject.getName() +"' " +
              "and with the same type and parameter list!";
      if (UmlUtils.oneSubClassHasEquallyOperation
        (selectedEObject.getClass_(), selectedEObject)) result.addFatalError(msg);
      // test: the owning class must not have any subclasses which inherit
      // operations with the same name, type and parameter list
      msg = "This refactoring can not be applied because at least one subclass " +
          "already inherits an operation named '" + selectedEObject.getName()
            +"' and with the same type and parameter list!";
      if (UmlUtils.oneSubClassInheritsEquallyOperation
        (selectedEObject.getClass_(), selectedEObject)) result.addFatalError(msg);
    }
    return result;
}
```

Figure F.40: Initial Check Implementation of UML class model refactoring *Push Down Operation*

IMPLEMENTATION Refactoring *Push Down Operation* has been implemented in Java code using the UML2EMF API.

Figure F.40 shows the concrete implementation of the initial precondition checks (Java method `checkInitialConditions()`). First, it is checked whether the contextual `Operation` (named *selectedEObject*) is owned by a `Class` since this refactoring should not be applied on interface operations. If this precondition is violated, an appropriate error message is returned (lines 176–179). Lines 182–184 check whether the contextual operation has *public* visibility since only public operations should be pushed down to the existing subclasses. Finally, the following checks are performed with respect to the owning class of the contextual operation[9]:

1. The owning class must have at least one subclass (lines 186–189).
2. The owning class must not be used as attribute type (lines 191–194).
3. The owning class must not have any subclasses that own operations which are equal to the contextual operation (lines 197–201).
4. The owning class must not have any subclasses that inherit operations which are equal to the contextual operation (lines 204–208).

```
126 public void run() {
127   org.eclipse.uml2.uml.Operation selectedEObject =
128     (org.eclipse.uml2.uml.Operation) dataManagement.
129         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
130   // execute: copy selected operation to each subclass
131   Class owningClass = selectedEObject.getClass_();
132   ArrayList<Class> subClasses = UmlUtils.getAllSubClasses(owningClass);
133   for (Class subClass : subClasses) {
134     // create copy of the selected operation
135     Copier attCopier = new Copier();
136     Operation newOperation = (Operation) attCopier.copy(selectedEObject);
137     attCopier.copyReferences();
138     // add copied operation to subclass
139     subClass.getOwnedOperations().add(newOperation);
140   }
141   // execute: remove selected operation from owning class
142   owningClass.getOwnedOperations().remove(selectedEObject);
143 }
```

Figure F.41: Model Change Implementation of UML class model refactoring *Push Down Operation*

Figure F.41 shows the concrete implementation of the proper model change of refactoring *Push Down Operation* (method `run()`)[10].

9 Please note that the concrete checks are implemented as static methods of a utility class in order to avoid redundant code and to reuse it as often as possible.

10 Since this refactoring does not have any additional parameters, no final precondition checks have to be implemented.

First, the contextual operation (*selectedEObject*) is copied to the corresponding operations list of each subclass of the owning class of the contextual operation (lines 131–139). Then, the contextual operation is removed from its containing class (line 142).

TEST CASES The following test cases have been performed:

1. The contextual operation *op* is owned by an interface *Interf1* ⇒ corresponding error message. ✓
2. The contextual operation *op* has *protected* visibility ⇒ corresponding error message. ✓
3. The owning class *A* of the contextual operation *op* has no subclass ⇒ corresponding error message. ✓
4. The owning class *A* of the contextual operation *op* is used as type of attribute *B::attB* ⇒ corresponding error message. ✓
5. The owning class *A* of the contextual operation *op* has a subclass *B* that owns an operation *op* with an equivalent parameter list like the contextual operation *op* ⇒ corresponding error message. ✓
6. The owning class *A* of the contextual operation *op* has a subclass *B* that inherits an operation *op* with an equivalent parameter list like the contextual operation *op* ⇒ corresponding error message. ✓
7. The owning class *A* of the contextual operation *op* has subclasses *B*, *C* and *D*; no violated preconditions ⇒ refactoring execution as expected. ✓

F.15 push down property

DESCRIPTION An attribute (property) is used only by some subclasses. Move the attribute to only these subclasses. More generally, this refactoring moves the attribute to all subclasses. If it makes sense, the attribute can be removed from some of these afterwards. Sometimes, it also makes sense to keep an attribute in all subclasses to hide it from the superclass. [107, 30, 150]

CONTEXTUAL ELEMENT `Property`

REFACTORING PARAMETERS This refactoring does not have any more parameters.

```
199 public RefactoringStatus checkInitialConditions() {
200     RefactoringStatus result = new RefactoringStatus();
201     org.eclipse.uml2.uml.Property selectedEObject =
202       (org.eclipse.uml2.uml.Property) dataManagement.
203           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
204     // test: the selected property must be an attribute (owned by a class)
205     String msg = "This refactoring can only be applied" +
206             " on properties which are owned attributes of a class!";
207     if (selectedEObject.getClass_() == null) {
208       result.addFatalError(msg);
209     } else {
210       // test: the selected attribute must be public
211       msg = "This refactoring can only be applied on public class attributes!";
212       if (! selectedEObject.getVisibility().equals(VisibilityKind.PUBLIC_LITERAL))
213                               result.addFatalError(msg);
214       // test: the owning class must have at least one subclass
215       msg = "This refactoring can not be applied because the owning class " +
216           "of the selected attribute does not have any subclasses!";
217       if (! UmlUtils.hasSubclasses(selectedEObject.getClass_()))
218                               result.addFatalError(msg);
219       // test: the owning class must not be used as attribute type
220       msg = "This refactoring can not be applied because the owning class " +
221           "of the selected attribute is used as attribute type!";
222       if (UmlUtils.isUsedAsAttributeType(selectedEObject.getClass_()))
223                               result.addFatalError(msg);
224       // test: the owning class must not have any subclasses which own
225       // attributes with the same name as the selected attribute
226       msg = "This refactoring can not be applied because at least one subclass " +
227           "already owns an attribute named '" + selectedEObject.getName() +"'!";
228       if (UmlUtils.oneSubClassHasEquallyNamedAttribute
229         (selectedEObject.getClass_(), selectedEObject)) result.addFatalError(msg);
230       // test: the owning class must not have any subclasses which inherit
231       // elements with the same name as the selected attribute
232       msg = "This refactoring can not be applied because at least one subclass " +
233           "already inherits an element named '" + selectedEObject.getName() +"'!";
234       if (UmlUtils.oneSubClassInheritsEquallyNamedAttribute
235         (selectedEObject.getClass_(), selectedEObject)) result.addFatalError(msg);
236     }
237     return result;
238 }
```

Figure F.42: Initial Check Implementation of UML class model refactoring *Push Down Property*

IMPLEMENTATION Refactoring *Push Down Property* has been implemented in Java code using the UML2EMF API.

Figure F.42 shows the concrete implementation of the initial precondition checks (Java method `checkInitialConditions()`). First, it is checked whether the contextual `Property` (named *selectedEObject*) is owned by a `Class` since this refactoring should be applied on class attributes only. If this precondition is violated, an appropriate error message is returned (lines 205–208). Lines 211–213 check whether the contextual attribute has *public* visibility since only public attributes should be pushed down to the existing subclasses. Finally, the following checks are performed with respect to the owning class of the contextual attribute[11]:

1. The owning class must have at least one subclass (lines 215–218).
2. The owning class must not be used as attribute type (lines 220–223).
3. The owning class must not have any subclasses that own attributes which are equal to the contextual attribute (lines 226–229).
4. The owning class must not have any subclasses that inherit attributes which are equal to the contextual attribute (lines 232–235).

Figure F.43 shows the concrete implementation of the proper model change of refactoring *Push Down Property* (method `run()`)[12]. First, the contextual attribute (*selectedEObject*) is copied to the corresponding attributes list of each subclass of the owning class of the contextual attribute (lines 138–142). If the contextual attribute is modeled as association end, lines 145–160 additionally create new associations according to the copied attributes. Finally, the potential association of the contextual attribute is deleted (lines 167–169) and the contextual attribute is removed from its containing class (line 171).

TEST CASES The following test cases have been performed:

1. The contextual attribute *att* is owned by an interface *Interf1* ⇒ corresponding error message. ✓
2. The contextual attribute *att* has *package* visibility ⇒ corresponding error message. ✓
3. The owning class *A* of the contextual attribute *att* has no subclass ⇒ corresponding error message. ✓

11 Please note that the concrete checks are implemented as static methods of a utility class in order to avoid redundant code and to reuse it as often as possible.

12 Since this refactoring does not have any additional parameters, no final precondition checks have to be implemented.

```
129 public void run() {
130   org.eclipse.uml2.uml.Property selectedEObject =
131     (org.eclipse.uml2.uml.Property) dataManagement.
132         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
133   // execute: copy selected attribute to each subclass
134   Class owningClass = selectedEObject.getClass_();
135   ArrayList<Class> subClasses = UmlUtils.getAllSubClasses(owningClass);
136   for (Class subClass : subClasses) {
137     // create copy of the selected attribute
138     Copier attCopier = new Copier();
139     Property newAttribute = (Property) attCopier.copy(selectedEObject);
140     attCopier.copyReferences();
141     // add copied attribute to subclass
142     subClass.getOwnedAttributes().add(newAttribute);
143     if (selectedEObject.getAssociation() != null) {
144       // selected attribute is member end of an association
145       Association assoc = selectedEObject.getAssociation();
146       Property oldOppositeEnd = assoc.getOwnedEnds().get(0);
147       // create copy of the opposite association end
148       Copier aeCopier = new Copier();
149       Property newAssocEnd = (Property) aeCopier.copy(oldOppositeEnd);
150       aeCopier.copyReferences();
151       newAssocEnd.setType(subClass);
152       // create new association and add association ends
153       Package owningPackage = assoc.getPackage();
154       Association newAssociation = UMLFactory.eINSTANCE.createAssociation();
155       String assocName = "a_" + subClass.getName().toLowerCase() + "_" +
156                 newAttribute.getType().getName().toLowerCase();
157       newAssociation.setName(assocName);
158       owningPackage.getPackagedElements().add(newAssociation);
159       newAssociation.getOwnedEnds().add(newAssocEnd);
160       newAssociation.getMemberEnds().add(newAttribute);
161     }
162   }
163   // execute: remove selected attribute from owning class
164   if (selectedEObject.getAssociation() != null) {
165     // selected attribute is member end of an association
166     // delete association from owning package
167     Association assoc = selectedEObject.getAssociation();
168     Package owningPackage = assoc.getPackage();
169     owningPackage.getPackagedElements().remove(assoc);
170   }
171   owningClass.getOwnedAttributes().remove(selectedEObject);
172 }
```

Figure F.43: Model Change Implementation of UML class model refactoring *Push Down Property*

4. The owning class A of the contextual attribute att is used as type of attribute $B::attB \Rightarrow$ corresponding error message. ✓
5. The owning class A of the contextual attribute att has a subclass B that owns an attribute $att \Rightarrow$ corresponding error message. ✓
6. The owning class A of the contextual attribute att has a subclass B that inherits an attribute $att \Rightarrow$ corresponding error message. ✓

7. The owning class A of the contextual attribute *att* has subclasses B, C and D; *att* is an end of association *assoc*; no violated preconditions $\Rightarrow$ refactoring execution as expected. ✓
8. The owning class A of the contextual attribute *att* has subclasses B, C and D; no violated preconditions $\Rightarrow$ refactoring execution as expected. ✓

F.16 remove empty associated class

DESCRIPTION There is an empty class that is associated to another class. An associated class is empty if it has no features except for possible getter and setter operations for the corresponding association end. Furthermore, it has no inner classes, subclasses, or superclasses, it does not implement any interfaces, and it is not referred to as type of an attribute, operation or parameter. [150, 93]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS This refactoring does not have any more parameters.

IMPLEMENTATION Refactoring *Remove Empty Associated Class* has been implemented in Java code using the UML2EMF API.

Figure F.44 shows the implementation of the initial precondition check (Java method `checkInitialConditions()`). First, it is checked whether the contextual `Class` (*selectedEObject*) is owned by a `Package`, i.e., this refactoring can not be applied on inner classes for example (lines 190–194). Then, it is checked whether the contextual class is associated to at least one other class (lines 197–204) and to at most one other class (lines 206–211). The remaining checks ensure that the contextual `Class` is empty except for the corresponding association end attribute. The following checks are performed and appropriate error messages are returned[13]:

1. The contextual `Class` must not have any owned attributes except for the corresponding association end attribute (lines 214–218).
2. The contextual `Class` must not have any owned operations (lines 220/221).
3. The contextual `Class` must not have any inner classes (lines 223/224).
4. The contextual `Class` must not have any subclasses (lines 226/227).
5. The contextual `Class` must not have any superclasses (lines 229/230).
6. The contextual `Class` must not implement any interfaces (lines 232–234).
7. The contextual `Class` must not use any interfaces (lines 236/237).

13 Please note that the concrete checks are implemented as static methods of a utility class in order to avoid redundant code and to reuse it as often as possible.

```
184 public RefactoringStatus checkInitialConditions(){
185     RefactoringStatus result = new RefactoringStatus();
186     org.eclipse.uml2.uml.Class selectedEObject =
187       (org.eclipse.uml2.uml.Class) dataManagement.
188           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
189     // test: the selected class must be owned by a package
190     String msg = "This refactoring can only be applied" +
191             " on classes which are owned by a package!";
192     if (selectedEObject.getPackage() == null) {
193       result.addFatalError(msg);
194       return result;
195     }
196     // test: the selected class must be associated to at least one class
197     List<Class> associatedclasses =
198               UmlUtils.getOtherAssociatedClasses(selectedEObject);
199     msg = "Class '" + selectedEObject.getName() +
200                 "' is not associated to any classes!";
201     if (associatedclasses.isEmpty()) {
202       result.addFatalError(msg);
203       return result;
204     }
205     // test: the selected class must be associated to at most one class
206     msg = "Class '" + selectedEObject.getName() +
207         "' is associated to more than one class!";
208     if (associatedclasses.size() > 1) {
209       result.addFatalError(msg);
210       return result;
211     }
212     // test: the selected class must not have any attributes
213     // except for those that are owned by associations
214     msg = "Class '" + selectedEObject.getName() + "' has further " +
215         "attributes except for those that are owned by associations!";
216     for (Property attr : selectedEObject.getOwnedAttributes()) {
217       if (attr.getAssociation() == null) result.addFatalError(msg);
218     }
219     // test: the selected class  must not own any operations
220     msg = "Class '" + selectedEObject.getName() + "' owns at least one operation!";
221     if (UmlUtils.hasOperations(selectedEObject)) result.addFatalError(msg);
222     // test: the class must not have any inner classes
223     msg = "Class '" + selectedEObject.getName() + "' has at least one inner class!";
224     if (UmlUtils.hasInnerClasses(selectedEObject)) result.addFatalError(msg);
225     // test: the selected class must not have any subclasses
226     msg = "Class '" + selectedEObject.getName() + "' has at least one subclass!";
227     if (UmlUtils.hasSubclasses(selectedEObject)) result.addFatalError(msg);
228     // test: the selected class must not have any superclasses
229     msg = "Class '" + selectedEObject.getName() + "' has at least one superclass!";
230     if (UmlUtils.hasSuperclasses(selectedEObject)) result.addFatalError(msg);
231     // test: the class must not implement any interfaces
232     msg = "Class '" + selectedEObject.getName()
233         + "' implements at least one interface!";
234     if (UmlUtils.implementsInterfaces(selectedEObject)) result.addFatalError(msg);
235     // test: the class must not use any interfaces
236     msg = "Class '" + selectedEObject.getName() + "' uses at least one interface!";
237     if (UmlUtils.usesInterfaces(selectedEObject)) result.addFatalError(msg);
238     // test: the class must not be used as attribute type
239     msg = "Class '" + selectedEObject.getName() +  "' is used as attribute type!";
240     if (UmlUtils.isUsedAsFurtherAttributeType(selectedEObject))
241       result.addFatalError(msg);
242     // test: the class must not be used as operation/parameter type
243     msg = "Class '" + selectedEObject.getName()
244         + "' is used as operation/parameter type!";
245     if (UmlUtils.isUsedAsParameterType(selectedEObject))
246       result.addFatalError(msg);
247     return result;
248 }
```

Figure F.44: Initial Check Implementation of UML class model refactoring *Remove Empty Associated Class*

8. The contextual `Class` must not be used as attribute type (lines 239–241).
9. The contextual `Class` must not be used as parameter type (lines 243–246).

```
126 public void run() {
127   org.eclipse.uml2.uml.Class selectedEObject =
128     (org.eclipse.uml2.uml.Class) dataManagement.
129         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
130   // execute: remove all associations from class
131   List<Association> associations = UmlUtils.getAssociations(selectedEObject);
132   List<Class> owningClasses = new ArrayList<Class>();
133   List<Property> attributesToDelete = new ArrayList<Property>();
134   List<Package> owningPackages = new ArrayList<Package>();
135   List<Association> associationsToDelete = new ArrayList<Association>();
136   for (Association assoc : associations) {
137     for (Property ae : assoc.getMemberEnds()) {
138       if (! assoc.getOwnedEnds().contains(ae)) {
139         owningClasses.add(ae.getClass_());
140         attributesToDelete.add(ae);
141       }
142     }
143     owningPackages.add(assoc.getPackage());
144     associationsToDelete.add(assoc);
145   }
146   // remove association end as class attributes
147   for (int i=0; i < owningClasses.size(); i++) {
148     owningClasses.get(i).getOwnedAttributes().remove(attributesToDelete.get(i));
149   }
150   // remove associations from owning packages
151   for (int i=0; i < owningPackages.size(); i++) {
152     owningPackages.get(i).getPackagedElements()
153                           .remove(associationsToDelete.get(i));
154   }
155   // execute: remove selected class from owning package
156   Package owningPackage = selectedEObject.getPackage();
157   owningPackage.getPackagedElements().remove(selectedEObject);
158 }
```

Figure F.45: Model Change Implementation of UML class model refactoring *Remove Empty Associated Class*

Figure F.45 shows the concrete implementation of the proper model change[14] of refactoring *Remove Empty Associated Class* (method `run()`). After collecting the corresponding elements (lines 131–144), the appropriate association ends are deleted from their containing classes if they are modeled as owned attributes of these classes (lines 147–149). Then, the corresponding `Associations` are removed from their owning `Packages` (lines 151–153). Finally, the contextual class is simply removed from the element list of its owning `Package` (lines 156/157).

TEST CASES The following test cases have been performed:

1. The contextual class A is an inner class of class $B \Rightarrow$ corresponding error message. ✓

14 Since this refactoring does not have any additional parameters, no final precondition checks have to be implemented.

2. The contextual class A has no incoming association $\Rightarrow$ corresponding error message. ✓
3. The contextual class A has no outgoing association $\Rightarrow$ corresponding error message. ✓
4. The contextual class A is associated to class B and to C $\Rightarrow$ corresponding error message. ✓
5. The contextual class A owns an attribute *attr* that is not an association end $\Rightarrow$ corresponding error message. ✓
6. The contextual class A owns an operation *op* $\Rightarrow$ corresponding error message. ✓
7. The contextual class A has an inner class B $\Rightarrow$ corresponding error message. ✓
8. The contextual class A has a superclass B $\Rightarrow$ corresponding error message. ✓
9. The contextual class A has a subclass B $\Rightarrow$ corresponding error message. ✓
10. The contextual class A implements interface *Interf1* $\Rightarrow$ corresponding error message. ✓
11. The contextual class A uses interface *Interf1* $\Rightarrow$ corresponding error message. ✓
12. The contextual class A is used as type of class attribute *P1::B::att* $\Rightarrow$ corresponding error message. ✓
13. The contextual class A is used as type of parameter *P1::C::op1::par1* $\Rightarrow$ corresponding error message. ✓
14. The contextual class A has an outgoing association to class B; no precondition is violated $\Rightarrow$ refactoring execution as expected. ✓
15. The contextual class A has an incoming association from class B; no precondition is violated $\Rightarrow$ refactoring execution as expected. ✓
16. The contextual class A has an outgoing association to class B as well as an incoming association from class B; no precondition is violated $\Rightarrow$ refactoring execution as expected. ✓

F.17 remove empty subclass

DESCRIPTION A superclass has an empty subclass which shall be removed. This class is not associated to another class. It has no features, no inner classes, no further subclasses, and is not associated to other classes. It does not implement any interfaces, and it is not referred to as type of an attribute, operation or parameter. [150]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS This refactoring does not have any more parameters.

IMPLEMENTATION Refactoring *Remove Empty Subclass* has been implemented in Java code using the UML2EMF API.

Figure F.46 shows the implementation of the initial precondition check (Java method `checkInitialConditions()`). First, it is checked whether the contextual `Class` (*selectedEObject*) is owned by a `Package`, i.e., this refactoring can not be applied on inner classes for example (lines 165–167). The remaining checks ensure that the contextual `Class` is empty. The following checks are performed and appropriate error messages are returned[15]:

1. The contextual `Class` must have at least one superclass (lines 169–171).
2. The contextual `Class` must not have any owned attributes (lines 173/174).
3. The contextual `Class` must not have any owned operations (lines 176/177).
4. The contextual `Class` must not have any subclasses (lines 179/180).
5. The contextual `Class` must not have any inner classes (lines 182/183).
6. The contextual `Class` must not have any outgoing associations (lines 185–188).
7. The contextual `Class` must not have any incoming associations (lines 190–193).
8. The contextual `Class` must not implement any interfaces (lines 195–197).
9. The contextual `Class` must not use any interfaces (lines 199/200).

15 Please note that the concrete checks are implemented as static methods of a utility class in order to avoid redundant code and to reuse it as often as possible.

```
159 public RefactoringStatus checkInitialConditions(){
160     RefactoringStatus result = new RefactoringStatus();
161     org.eclipse.uml2.uml.Class selectedEObject =
162       (org.eclipse.uml2.uml.Class) dataManagement.
163           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
164     // test: the selected class must be owned by a package
165     String msg = "This refactoring can only be applied" +
166             " on classes which are owned by a package!";
167     if (selectedEObject.getPackage() == null) result.addFatalError(msg);
168     // test: the selected class must have at least one superclass
169     msg = "Class '" + selectedEObject.getName() +
170         "' does not have any superclasses!";
171     if (! UmlUtils.hasSuperclasses(selectedEObject)) result.addFatalError(msg);
172     // test: the selected class  must not own any attributes
173     msg = "Class '" + selectedEObject.getName() + "' owns at least one attribute!";
174     if (UmlUtils.hasAttributes(selectedEObject)) result.addFatalError(msg);
175     // test: the selected class  must not own any operations
176     msg = "Class '" + selectedEObject.getName() + "' owns at least one operation!";
177     if (UmlUtils.hasOperations(selectedEObject)) result.addFatalError(msg);
178     // test: the selected class must not have any subclasses
179     msg = "Class '" + selectedEObject.getName() + "' has at least one subclass!";
180     if (UmlUtils.hasSubclasses(selectedEObject)) result.addFatalError(msg);
181     // test: the class must not have any inner classes
182     msg = "Class '" + selectedEObject.getName() + "' has at least one inner class!";
183     if (UmlUtils.hasInnerClasses(selectedEObject)) result.addFatalError(msg);
184     // test: the class must not have any outgoing associations
185     msg = "Class '" + selectedEObject.getName()
186         + "' has at least one outgoing association!";
187     if (UmlUtils.hasOutgoingAssociations(selectedEObject))
188       result.addFatalError(msg);
189     // test: the class must not have any incoming associations
190     msg = "Class '" + selectedEObject.getName()
191         + "' has at least one incoming association!";
192     if (UmlUtils.hasIncomingAssociations(selectedEObject))
193       result.addFatalError(msg);
194     // test: the class must not implement any interfaces
195     msg = "Class '" + selectedEObject.getName()
196         + "' implements at least one interface!";
197     if (UmlUtils.implementsInterfaces(selectedEObject)) result.addFatalError(msg);
198     // test: the class must not use any interfaces
199     msg = "Class '" + selectedEObject.getName() + "' uses at least one interface!";
200     if (UmlUtils.usesInterfaces(selectedEObject)) result.addFatalError(msg);
201     // test: the class must not be used as attribute type
202     msg = "Class '" + selectedEObject.getName() +  "' is used as attribute type!";
203     if (UmlUtils.isUsedAsAttributeType(selectedEObject))
204       result.addFatalError(msg);
205     // test: the class must not be used as operation/parameter type
206     msg = "Class '" + selectedEObject.getName()
207         + "' is used as operation/parameter type!";
208     if (UmlUtils.isUsedAsParameterType(selectedEObject))
209       result.addFatalError(msg);
210     return result;
211 }
```

Figure F.46: Initial Check Implementation of UML class model refactoring *Remove Empty Subclass*

10. The contextual `Class` must not be used as attribute type (lines 202–204).
11. The contextual `Class` must not be used as parameter type (lines 206–209).

Figure F.47 shows the concrete implementation of the proper model change[16] of refactoring *Remove Empty Subclass* (method `run()`). First, all `Generalization` relationships from the contextual `Class` (*selectedEObject*) to other classes are deleted (line 128). Finally, the contextual class is simply removed from the element list of its owning `Package` (lines 130/131).

```
123 public void run() {
124   org.eclipse.uml2.uml.Class selectedEObject =
125     (org.eclipse.uml2.uml.Class) dataManagement.
126         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
127   // execute: delete generalizations from selected class
128   selectedEObject.getGeneralizations().clear();
129   // execute: delete selected class from owning package
130   Package p = selectedEObject.getPackage();
131   p.getPackagedElements().remove(selectedEObject);
132 }
```

Figure F.47: Model Change Implementation of UML class model refactoring *Remove Empty Subclass*

TEST CASES The following test cases have been performed:

1. The contextual class A has no superclass $\Rightarrow$ corresponding error message. ✓
2. The contextual class A is an inner class of class B $\Rightarrow$ corresponding error message. ✓
3. The contextual class A owns an attribute *attr* $\Rightarrow$ corresponding error message. ✓
4. The contextual class A owns an operation *op* $\Rightarrow$ corresponding error message. ✓
5. The contextual class A has an inner class B $\Rightarrow$ corresponding error message. ✓
6. The contextual class A has a subclass B $\Rightarrow$ corresponding error message. ✓
7. The contextual class A has an incoming association *assoc* $\Rightarrow$ corresponding error message. ✓
8. The contextual class A has an outgoing association *assoc* $\Rightarrow$ corresponding error message. ✓
9. The contextual class A implements interface *Interf1* $\Rightarrow$ corresponding error message. ✓
10. The contextual class A uses interface *Interf1* $\Rightarrow$ corresponding error message. ✓
11. The contextual class A is used as type of class attribute *P1::B::att* $\Rightarrow$ corresponding error message. ✓

16 Since this refactoring does not have any additional parameters, no final precondition checks have to be implemented.

12. The contextual class *A* is used as type of parameter *P1::C::op1::par1* ⇒ corresponding error message. ✓
13. The contextual class *A* has one superclass *B*; no precondition is violated ⇒ refactoring execution as expected. ✓
14. The contextual class *A* has superclasses *B* and *C*; no precondition is violated ⇒ refactoring execution as expected. ✓

F.18 remove empty superclass

DESCRIPTION A set of classes has an empty superclass which shall be removed. This class is not associated to another class. It has no features, no inner classes, and is not associated to other classes. It does not implement any interfaces, and it is not referred to as type of an attribute, operation or parameter. [150]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS This refactoring does not have any more parameters.

IMPLEMENTATION Refactoring *Remove Empty Superclass* has been implemented in Java code using the UML2EMF API.

Figure F.48 shows the implementation of the initial precondition check (Java method `checkInitialConditions()`). First, it is checked whether the contextual `Class` (*selectedEObject*) is owned by a `Package`, i.e., this refactoring can not be applied on inner classes for example (lines 173–175). The remaining checks ensure that the contextual `Class` is empty. The following checks are performed and appropriate error messages are returned[17]:

1. The contextual `Class` must have at least one subclass (lines 177/178).
2. The contextual `Class` must not have any owned attributes (lines 180/181).
3. The contextual `Class` must not have any owned operations (lines 183/184).
4. The contextual `Class` must not have any superclasses (lines 186/187).
5. The contextual `Class` must not have any inner classes (lines 189/190).
6. The contextual `Class` must not have any outgoing associations (lines 192–195).
7. The contextual `Class` must not have any incoming associations (lines 197-200).
8. The contextual `Class` must not implement any interfaces (lines 202–205).
9. The contextual `Class` must not use any interfaces (lines 207/208).
10. The contextual `Class` must not be used as attribute type (lines 210/211).

17 Please note that the concrete checks are implemented as static methods of a utility class in order to avoid redundant code and to reuse it as often as possible.

```
167 public RefactoringStatus checkInitialConditions(){
168     RefactoringStatus result = new RefactoringStatus();
169     org.eclipse.uml2.uml.Class selectedEObject =
170       (org.eclipse.uml2.uml.Class) dataManagement.
171           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
172     // test: the selected class must be owned by a package
173     String msg = "This refactoring can only be applied" +
174             " on classes which are owned by a package!";
175     if (selectedEObject.getPackage() == null) result.addFatalError(msg);
176     // test: the selected class must have at least one superclass
177     msg = "Class '" + selectedEObject.getName() + "' does not have any subclasses!";
178     if (! UmlUtils.hasSubclasses(selectedEObject)) result.addFatalError(msg);
179     // test: the selected class  must not own any attributes
180     msg = "Class '" + selectedEObject.getName() + "' owns at least one attribute!";
181     if (UmlUtils.hasAttributes(selectedEObject)) result.addFatalError(msg);
182     // test: the selected class  must not own any operations
183     msg = "Class '" + selectedEObject.getName() + "' owns at least one operation!";
184     if (UmlUtils.hasOperations(selectedEObject)) result.addFatalError(msg);
185     // test: the selected class must not have any superclasses
186     msg = "Class '" + selectedEObject.getName() + "' has at least one superclass!";
187     if (UmlUtils.hasSuperclasses(selectedEObject)) result.addFatalError(msg);
188     // test: the class must not have any inner classes
189     msg = "Class '" + selectedEObject.getName() + "' has at least one inner class!";
190     if (UmlUtils.hasInnerClasses(selectedEObject)) result.addFatalError(msg);
191     // test: the class must not have any outgoing associations
192     msg = "Class '" + selectedEObject.getName()
193         + "' has at least one outgoing association!";
194     if (UmlUtils.hasOutgoingAssociations(selectedEObject))
195       result.addFatalError(msg);
196     // test: the class must not have any incoming associations
197     msg = "Class '" + selectedEObject.getName()
198         + "' has at least one incoming association!";
199     if (UmlUtils.hasIncomingAssociations(selectedEObject))
200       result.addFatalError(msg);
201     // test: the class must not implement any interfaces
202     msg = "Class '" + selectedEObject.getName()
203         + "' implements at least one interface!";
204     if (UmlUtils.implementsInterfaces(selectedEObject))
205       result.addFatalError(msg);
206     // test: the class must not use any interfaces
207     msg = "Class '" + selectedEObject.getName() + "' uses at least one interface!";
208     if (UmlUtils.usesInterfaces(selectedEObject)) result.addFatalError(msg);
209     // test: the class must not be used as attribute type
210     msg = "Class '" + selectedEObject.getName() +  "' is used as attribute type!";
211     if (UmlUtils.isUsedAsAttributeType(selectedEObject)) result.addFatalError(msg);
212     // test: the class must not be used as operation/parameter type
213     msg = "Class '" + selectedEObject.getName()
214         + "' is used as operation/parameter type!";
215     if (UmlUtils.isUsedAsParameterType(selectedEObject))
216       result.addFatalError(msg);
217     return result;
218 }
```

Figure F.48: Initial Check Implementation of UML class model refactoring *Remove Empty Superclass*

11. The contextual `Class` must not be used as parameter type (lines 213–216).

Figure F.49 shows the concrete implementation of the proper model change[18] of refactoring *Remove Empty Superclass* (method

18 Since this refactoring does not have any additional parameters, no final precondition checks have to be implemented.

`run()`). First, all `Generalization` relationships from the other classes to the contextual `Class` (*selectedEObject*) are deleted (lines 130—136). Finally, the contextual class is simply removed from the element list of its owning `Package` (lines 138/139).

```
125 public void run() {
126   org.eclipse.uml2.uml.Class selectedEObject =
127     (org.eclipse.uml2.uml.Class) dataManagement.
128         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
129   // execute: delete generalizations to selected class
130   List<Class> allClasses = UmlUtils.getAllClasses(selectedEObject.getModel());
131   for (Class cl : allClasses) {
132     if (cl.getSuperClasses().contains(selectedEObject)){
133       Generalization gen = cl.getGeneralization(selectedEObject);
134       cl.getGeneralizations().remove(gen);
135     }
136   }
137   // execute: delete selected class from owning package
138   Package p = selectedEObject.getPackage();
139   p.getPackagedElements().remove(selectedEObject);
140 }
```

Figure F.49: Model Change Implementation of UML class model refactoring *Remove Empty Superclass*

TEST CASES The following test cases have been performed:

1. The contextual class A has no subclass $\Rightarrow$ corresponding error message. ✓
2. The contextual class A is an inner class of class B $\Rightarrow$ corresponding error message. ✓
3. The contextual class A owns an attribute *attr* $\Rightarrow$ corresponding error message. ✓
4. The contextual class A owns an operation *op* $\Rightarrow$ corresponding error message. ✓
5. The contextual class A has an inner class B $\Rightarrow$ corresponding error message. ✓
6. The contextual class A has a superclass B $\Rightarrow$ corresponding error message. ✓
7. The contextual class A has an incoming association *assoc* $\Rightarrow$ corresponding error message. ✓
8. The contextual class A has an outgoing association *assoc* $\Rightarrow$ corresponding error message. ✓
9. The contextual class A implements interface *Interf1* $\Rightarrow$ corresponding error message. ✓
10. The contextual class A uses interface *Interf1* $\Rightarrow$ corresponding error message. ✓
11. The contextual class A is used as type of class attribute *P1::B::att* $\Rightarrow$ corresponding error message. ✓

12. The contextual class *A* is used as type of parameter *P1::C::op1::par1* ⇒ corresponding error message. ✓
13. The contextual class *A* has one subclass *B*; no precondition is violated ⇒ refactoring execution as expected. ✓
14. The contextual class *A* has subclasses *B* and *C*; no precondition is violated ⇒ refactoring execution as expected. ✓

F.19 remove parameter

DESCRIPTION A parameter is no longer needed by the implementation of an operation. Therefore, this refactoring removes this parameter from the parameter list of the corresponding operation. [30, 150]

CONTEXTUAL ELEMENT `Parameter`

REFACTORING PARAMETERS This refactoring does not have any more parameters.

IMPLEMENTATION Refactoring *Add Parameter* has been implemented in Java code using the UML2EMF API.

```
161 public RefactoringStatus checkInitialConditions() {
162     RefactoringStatus result = new RefactoringStatus();
163     org.eclipse.uml2.uml.Parameter selectedEObject =
164       (org.eclipse.uml2.uml.Parameter) dataManagement.
165           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
166     // test: the selected parameter must be an input parameter
167     String msg = "This refactoring can only be applied on input parameters!";
168     if ((selectedEObject.getDirection().getValue()
169             == ParameterDirectionKind.RETURN)
170         || (selectedEObject.getDirection().getValue()
171               == ParameterDirectionKind.OUT)) {
172       result.addFatalError(msg);
173       return result;
174     }
175     // test: the selected operation must be owned by a class
176     msg = "This refactoring can only be applied on parameters" +
177               " whose operation is owned by a class!";
178     if (selectedEObject.getOperation().getClass_() == null) {
179       result.addFatalError(msg);
180       return result;
181     }
182     // test: the owning class must not own an operation with the name
183     // of the contextual operation and a similar parameter list after
184     // inserting a parameter with the given name and type
185     msg = "The owning class already owns an operation named '" +
186         selectedEObject.getOperation().getName() + "' having the same signature " +
187         "(type and parameter list) after removing the selected parameter!";
188     Class cl = selectedEObject.getOperation().getClass_();
189     if (classOwnsOperation(cl, selectedEObject)) result.addFatalError(msg);
190     // test: the owning class must not own an operation with the name
191     // of the contextual operation and a similar parameter list after
192     // inserting a parameter with the given name and type
193     msg = "The owning class already inherits an operation named '" +
194         selectedEObject.getOperation().getName() + "' having the same signature " +
195         "(type and parameter list) after removing the selected parameter!";
196     if (classInheritsOperation(cl, selectedEObject)) result.addFatalError(msg);
197     return result;
198 }
```

Figure F.50: Initial Check Implementation of UML class model refactoring *Remove Parameter*

Figure F.50 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`). Here, the following checks are performed:

1. The contextual `Parameter` must be an input parameter, i.e., its direction must not be of kind `ParameterDirectionKind::RETURN` or `ParameterDirectionKind::OUT` (lines 167–173).
2. The owning `Operation` of the contextual `Parameter` must be owned by a `Class`, i.e., this refactoring can not be applied on interface operation parameters (lines 176–181).
3. The owning `Class` of the owning `Operation` of the contextual `Parameter` must not own a similar `Operation` after removing the contextual `Parameter` (lines 185–189).
4. The owning `Class` of the owning `Operation` of the contextual `Parameter` must not inherit a similar `Operation` after removing the contextual `Parameter` (lines 193–197).

```
127 public void run() {
128   org.eclipse.uml2.uml.Parameter selectedEObject =
129     (org.eclipse.uml2.uml.Parameter) dataManagement.
130         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
131   // execute: remove selected parameter from operation
132   Operation owningOperation = selectedEObject.getOperation();
133   owningOperation.getOwnedParameters().remove(selectedEObject);
134 }
```

Figure F.51: Model Change Implementation of UML class model refactoring *Remove Parameter*

Figure F.51 shows the concrete implementation of the proper model change[19] of refactoring *Remove Parameter* (method `run()`). Here, the contextual `Parameter` (*selectedEObject*) is simply removed from the parameter list of its owning operation (lines 132/133).

TEST CASES The following test cases have been performed:

1. The contextual parameter *par1* has direction *RETURN* ⇒ corresponding error message. ✓
2. The owning operation *op1* of the contextual parameter *par1* is owned by an interface *Interf1* ⇒ corresponding error message. ✓
3. The owning class *A* of the owning operation *op1* of the contextual parameter *par1* owns an operation *op1* with the same signature as the owning operation *op1* after removing the contextual parameter *par1* ⇒ corresponding error message. ✓
4. The owning class *A* of the owning operation *op1* of the contextual parameter *par1* inherits an operation *op1* with the

19 Since this refactoring does not have any additional parameters, no final precondition checks have to be implemented

same signature as the owning operation *op1* after removing the contextual parameter *par1* ⇒ corresponding error message. ✓

5. No precondition is violated; the contextual parameter *par1* is the first parameter in the list of altogether three input parameters ⇒ refactoring execution as expected. ✓
6. No precondition is violated; the contextual parameter *par1* is the middle parameter in the list of altogether three input parameters ⇒ refactoring execution as expected. ✓
7. No precondition is violated; the contextual parameter *par1* is the last parameter in the list of altogether three input parameters ⇒ refactoring execution as expected. ✓

F.20 remove superclass

DESCRIPTION There is a set of classes having a superclass that does not make sense anymore. Remove this superclass after pushing remaining features down. [141, 150, 107, 160]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS This refactoring has no additional parameter. A list of attributes and operations which have to be pushed to the existing subclasses is taken from the contextual class.

IMPLEMENTATION Refactoring *Remove Superclass* has been implemented in Java code using the UML2EMF API (for specifying precondition checks) respectively an appropriate model of the CoMReL language (for specifying the proper model changes) using predefined UML refactorings.

```
212 public RefactoringStatus checkInitialConditions(){
213     RefactoringStatus result = new RefactoringStatus();
214     org.eclipse.uml2.uml.Class selectedEObject =
215         (org.eclipse.uml2.uml.Class) dataManagement.
216             getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
217     // Test: the class must have sub classes
218     ArrayList<org.eclipse.uml2.uml.Class> subclasses =
219                                 UmlUtils.getAllSubClasses(selectedEObject);
220     String msg = "Class '" + selectedEObject.getName()
221         + "' does not have any subclasses!";
222     if (subclasses.isEmpty()) result.addFatalError(msg);
223     return result;
224 }
```

Figure F.52: Initial Check Implementation of UML class model refactoring *Remove Superclass*

Figure F.52 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`). Here, the only check is whether the contextual `Class` (named *selectedEObject*) has some subclasses, otherwise this refactoring does not make sense and an appropriate error message is returned (lines 218–222).

Figure F.53 shows the concrete CoMReL unit specification of the proper model change of refactoring *Remove Superclass*[20]. As described in Appendix E of this thesis, refactoring *Remove Superclass* relies on three atomic model refactorings.

20 Since this refactoring does not have any additional parameters, no final precondition checks have to be implemented.

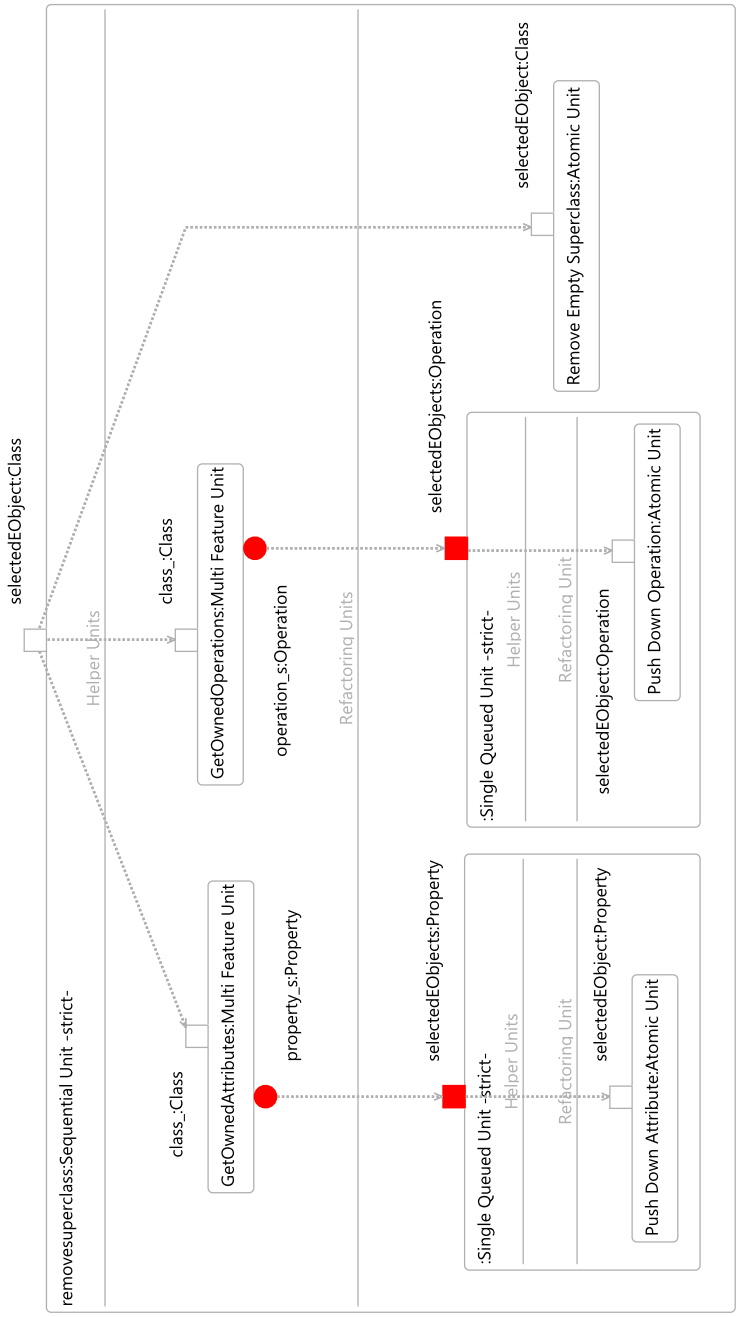


Figure F.53: Model Change Implementation of UML class model refactoring *Remove Superclass*

The main refactoring unit *removesuperclass* is a strict `Sequential Unit` consisting of two `SingleQueuedUnits` and one `AtomicUnit`. The queued units push down all attributes and operations to each subclass of the contextual `Class`. Then, this class is removed by the corresponding atomic unit.

Each `SingleQueuedUnit` contains an `AtomicUnit` calling an already existing model refactoring. First, atomic refactoring *Push Down Attribute* must be applied on each attribute of the selected class. Analogously, an atomic unit for refactoring *Push Down Operation* is put into a single queued unit. In both cases, the according *strict* attributes are set to *true* since each feature should be pushed down to ensure that the contextual class is empty afterwards.

To ensure conformity with respect to typing and multiplicity of included ports, the main refactoring unit *removesuperclass* requires two helper units, more precisely `FeatureUnits`. These feature units, *Get Owned Attributes* and *Get Owned Operations*, take the contextual class as input and yield all owned attributes and operations of that class.

TEST CASES The following test cases have been performed:

1. The contextual class A is an inner class of class B $\Rightarrow$ no changes. ✓
2. The contextual class A has no subclasses $\Rightarrow$ corresponding error message. ✓
3. The contextual class A has a superclass B $\Rightarrow$ no changes. ✓
4. The contextual class A has subclasses B, C and D; class A owns attribute *att* with *private* visibility $\Rightarrow$ no changes. ✓
5. The contextual class A has subclasses B, C and D; class A owns attribute *att*; class B owns attribute *att* $\Rightarrow$ no changes. ✓
6. The contextual class A has subclasses B, C and D; class A owns attribute *att*; class B inherits attribute *att* from superclass E $\Rightarrow$ no changes. ✓
7. The contextual class A has subclasses B, C and D; class A owns operation *op* with *protected* visibility $\Rightarrow$ no changes. ✓
8. The contextual class A has subclasses B, C and D; class A owns operation *op*; class B owns operation *op* $\Rightarrow$ no changes. ✓
9. The contextual class A has subclasses B, C and D; class A owns operation *op*; class B inherits operation *op* from superclass E $\Rightarrow$ no changes. ✓
10. The contextual class A has an inner class B $\Rightarrow$ no changes. ✓
11. The contextual class A has an incoming association *assoc* $\Rightarrow$ no changes. ✓
12. The contextual class A has an outgoing association *assoc* $\Rightarrow$ no changes. ✓
13. The contextual class A implements interface *Interf1* $\Rightarrow$ no changes. ✓

14. The contextual class *A* uses interface *Interf1* ⇒ no changes. ✓
15. The contextual class *A* is used as type of class attribute *P1::B::att* ⇒ no changes. ✓
16. The contextual class *A* is used as type of operation parameter *P1::C::op1::par1* ⇒ no changes. ✓
17. The contextual class *A* has subclasses *B*, *C* and *D*; class *A* contains attributes *att1* and *att2* as well as equal operations *op1* and *op2*; no violated internal preconditions ⇒ refactoring execution as expected. ✓

F.21 rename class

DESCRIPTION The current name of a class does not reflect its purpose. This refactoring changes the name of the class to a new name. [30]

CONTEXTUAL ELEMENT `Class`

REFACTORING PARAMETERS `newName` - New name of the contextual class.

IMPLEMENTATION Refactoring *Rename Class* has been implemented in Henshin pattern rules (for specifying precondition checks) respectively a Henshin transformation rule (for specifying the proper model change) using the abstract syntax of UML.

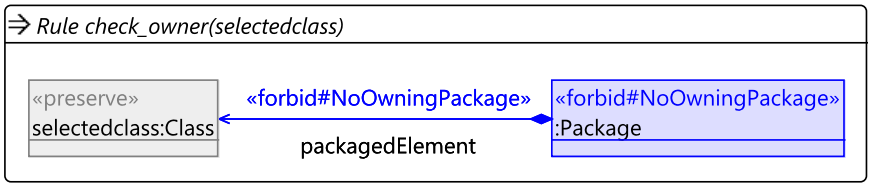


Figure F.54: Initial Check Implementation of UML class model refactoring *Rename Class*

Figure F.54 shows the Henshin pattern rule specification of the initial precondition check. This rule defines the erroneous situation that the contextual `Class` (named *selectedclass*) is not owned by a `Package`, i.e., this refactoring can not be applied on inner classes for example. NAC *NoOwningPackage* specifies this violated precondition.

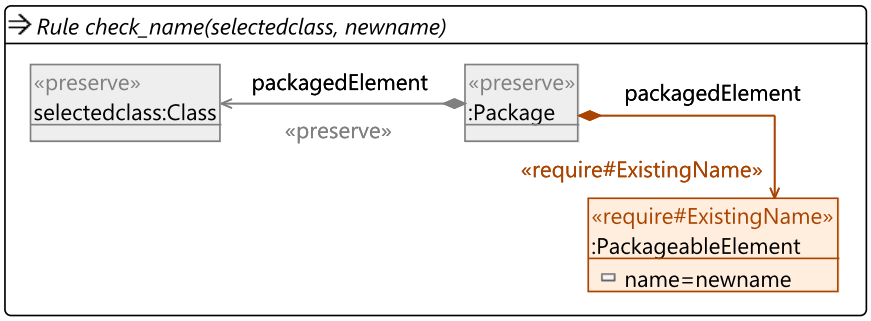


Figure F.55: Final Check Implementation of UML class model refactoring *Rename Class*

Figure F.55 shows the Henshin pattern rule that defines the final precondition check of refactoring *Rename Class*. It specifies the violated precondition that the owning `Package` of the contextual `Class` already owns an element with the same name as

specified in rule parameter *newname* (see PAC *ExistingName* in Figure F.55).

Figure F.56 shows the Henshin transformation rule of the proper model change specification of refactoring *Rename Class*. Here, attribute *name* of the contextual `Class` (*selectedclass*) is set to the value of rule parameter *newname*.

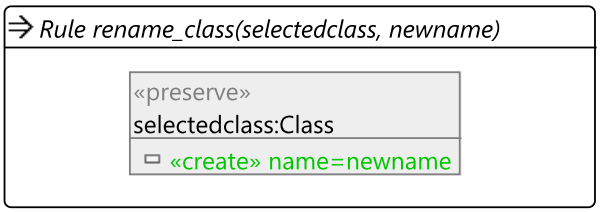


Figure F.56: Model Change Implementation of UML class model refactoring *Rename Class*

TEST CASES The following test cases have been performed:

1. The contextual class *A* is an inner class of class *B* ⇒ corresponding error message. ✓
2. *newname* is set to *B*; the owning package *P1* of the contextual class *A* already owns a class *B* ⇒ corresponding error message. ✓
3. *newname* is set to *Interf1*; the owning package *P1* of the contextual class *A* owns an interface *Interf1* ⇒ corresponding error message. ✓
4. No precondition is violated ⇒ refactoring execution as expected. ✓

F.22 rename operation

DESCRIPTION The current name of an operation does not reflect its purpose. This refactoring changes the name of the operation. [107, 30]

CONTEXTUAL ELEMENT `Operation`

REFACTORING PARAMETERS `newName` - New name of the contextual operation.

IMPLEMENTATION Refactoring *Rename Operation* has been implemented in Java code using the UML2EMF API.

```
160 public RefactoringStatus checkInitialConditions() {
161   RefactoringStatus result = new RefactoringStatus();
162   org.eclipse.uml2.uml.Operation selectedEObject =
163     (org.eclipse.uml2.uml.Operation) dataManagement.
164         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
165   // test: the selected operation must be owned by a class
166   String msg = "This refactoring can only be applied" +
167           " on operations which are owned by a class!";
168   if (selectedEObject.getClass_() == null) result.addFatalError(msg);
169   return result;
170 }
```

Figure F.57: Initial Check Implementation of UML class model refactoring *Rename Operation*

Figure F.57 shows the concrete implementation of the initial precondition check (Java method `checkInitialConditions()`). Here, the only check is whether the contextual UML `Operation` (named *selectedEObject*) is owned by a `Class` (line 168) since this refactoring should not be applied on interface operations. If this precondition is violated, an appropriate error message is returned (lines 166–168).

Figure F.58 shows the concrete implementation of the final precondition check (Java method `checkInitialConditions()`). Besides the contextual `Operation` *selectedEObject*, this check additionally considers the user input parameter *newName*. Here, the following checks are performed:

1. The specified name in parameter *newName* must be different from the current name of the contextual `Operation` (line 187–189).
2. The owning `Class` of the contextual `Operation` must not own an `Operation` named as specified in parameter *newName* and having an equivalent parameter list (lines 192–196).
3. The owning `Class` of the contextual `Operation` must not inherit an `Operation` named as specified in parameter *new-*

```
179 public RefactoringStatus checkFinalConditions() {
180     RefactoringStatus result = new RefactoringStatus();
181     org.eclipse.uml2.uml.Operation selectedEObject =
182       (org.eclipse.uml2.uml.Operation) dataManagement.
183           getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
184     String newName =
185       (String) dataManagement.getInPortByName("newName").getValue();
186     // test: the inserted name must be different from the old operation name
187     String msg = "The selected operation is already named '" + newName + "'!";
188     if (selectedEObject.getName().equals(newName))
189       result.addFatalError(msg);
190     // the owning class must not own an operation with the inserted name
191     // and a similar parameter list
192     msg = "The owning class already owns an operation named '" + newName +
193         "' having the same signature (type and parameter list)!";
194     Class cl = selectedEObject.getClass_();
195     if (classOwnsOperation(cl, selectedEObject, newName))
196       result.addFatalError(msg);
197     // the owning class must not inherit an operation with the inserted name
198     // and a similar parameter list
199     msg = "The owning class already inherits an operation named '" + newName +
200         "' having the same signature (type and parameter list)!";
201     if (classInheritsOperation(cl, selectedEObject, newName))
202       result.addFatalError(msg);
203     return result;
204 }
```

Figure F.58: Final Check Implementation of UML class model refactoring *Rename Operation*

Name and having an equivalent parameter list (lines 199–202).

Figure F.59 shows the concrete implementation of the proper model change of refactoring *Rename Operation* (method `run()`). Here, attribute *name* of the contextual `Operation` (*selectedEObject*) is set to the value of rule parameter *newName* (line 132).

```
125 public void run() {
126   org.eclipse.uml2.uml.Operation selectedEObject =
127     (org.eclipse.uml2.uml.Operation) dataManagement.
128         getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
129   String newName =
130     (String) dataManagement.getInPortByName("newName").getValue();
131   // execute: rename selected operation to the inserted name
132   selectedEObject.setName(newName);
133 }
```

Figure F.59: Model Change Implementation of UML class model refactoring *Rename Operation*

TEST CASES The following test cases have been performed:

1. The contextual operation *op1* is owned by an interface *Interf1* ⇒ corresponding error message. ✓
2. *newName* is set to *op1*; the contextual operation is names *op1* ⇒ corresponding error message. ✓
3. *newName* is set to *op2*; the owning class *A* of the contextual operation *op1* ownes an operation *op2* with the same sig-

nature as the contextual operation *op1* ⇒ corresponding error message. ✓

4. *newName* is set to *op2;* the owning class *A* of the contextual operation *op1* inherits an operation *op2* with the same signature as the contextual operation *op1* ⇒ corresponding error message. ✓
5. No precondition is violated ⇒ refactoring execution as expected. ✓

F.23 rename property

DESCRIPTION The current name of an attribute or association end does not reflect its purpose. This refactoring changes the name of the property. [107, 30]

CONTEXTUAL ELEMENT `Property`

REFACTORING PARAMETERS `newName` - New name of the contextual attribute or association end.

IMPLEMENTATION The UML class model refactoring *Rename Property* has been implemented in Henshin pattern rules (for specifying precondition checks) respectively a Henshin transformation rule (for specifying the proper model change) using the abstract syntax of UML.

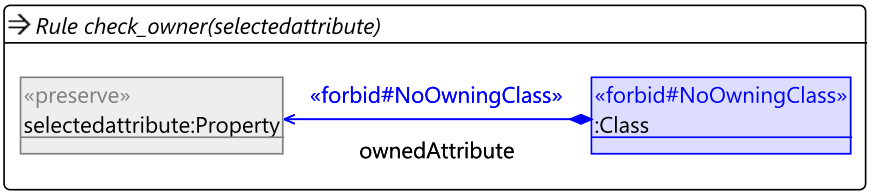


Figure F.60: Initial Check Implementation of UML class model refactoring *Rename Property*

Figure F.60 shows the Henshin pattern rule specification of the initial precondition check. This rule defines the erroneous situation that the contextual `Property` (named *selectedattribute*) is not owned by a `Class`, i.e., this refactoring can be applied on class attributes only. NAC *NoOwningClass* specifies this violated precondition.

Figure F.61 shows the Henshin pattern rules that define the final precondition checks of refactoring *Rename Property*. The first rule specifies the violated precondition that the owning `Class` of the contextual attribute (meta model element `Property`) already owns an attribute with the same name as specified in rule parameter *newname* (PAC *ExistingName*). The second rule specifies the violated precondition that the owning `Class` of the contextual attribute already inherits an attribute with the same name as specified in rule parameter *newname* (PAC *InheritedName*).

Figure F.62 shows the Henshin transformation rule of the proper model change specification of refactoring *Rename Property*. Here, attribute *name* of the contextual `Property` (*selectedattribute*) is set to the value of rule parameter *newname*.

TEST CASES The following test cases have been performed:

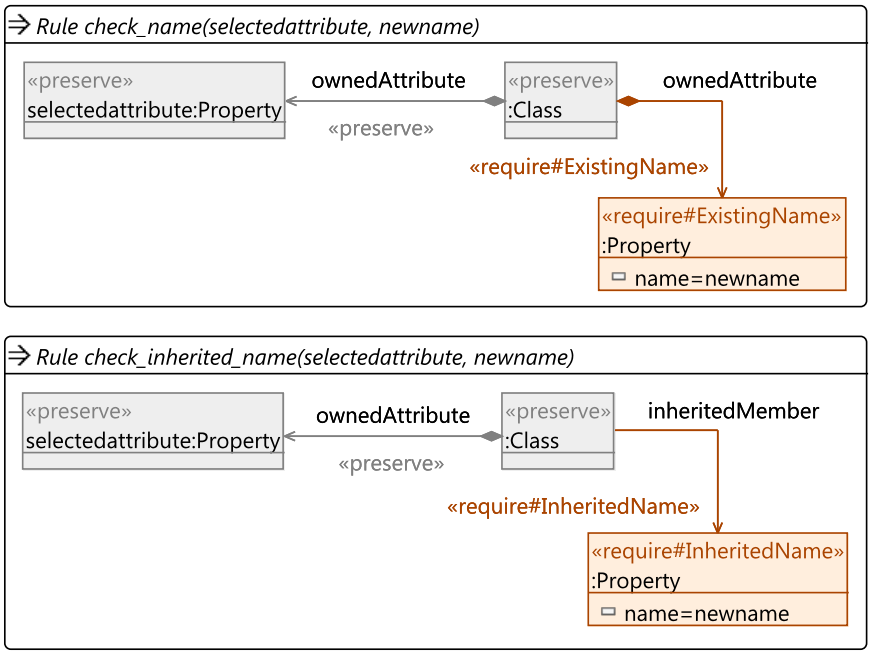


Figure F.61: Final Check Implementation of UML class model refactoring *Rename Property*

1. The contextual attribute *attr1* is an owned end of an association *assoc1* ⇒ corresponding error message. ✓
2. *newname* is set to *attr2*; the owning class *A* of the contextual attribute *attr1* already owns an attribute *attr2* ⇒ corresponding error message. ✓
3. *newname* is set to *attr2*; the owning class *A* of the contextual attribute *attr1* already inherits an attribute *attr2* ⇒ corresponding error message. ✓
4. No precondition is violated ⇒ refactoring execution as expected. ✓

Figure F.62: Model Change Implementation of UML class model refactoring *Rename Property*

G

STUDY MATERIAL EXPERIMENT EX_APP

This appendix contains material of the experiment `Ex_App` used in Chapter 14 of this thesis.

Philipps Universität Marburg

Prof. Dr. Gabriele Taentzer
Thorsten Arendt

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik

Einführung in die Softwaretechnik

Tutorium 11

Wintersemester 13/14
Mittwoch, 22.01.2014
14.15Uhr - 16.00Uhr

Im Mittelpunkt dieses Tutoriums steht ein UML Klassenmodell, das im Rahmen der Analysephase eines Softwareentwicklungsprojekts erstellt wurde. In diesem Projekt soll eine Anwendung für die Abrechnung von Mietvorgängen eines Fahrzeugverleihs (z.B. Europcar, Sixt etc.) entwickelt werden. Abbildung 1 auf der folgenden Seite zeigt ein erstes Analysemodell der entsprechenden Domäne.

Im Rahmen dieses Tutoriums sollen Techniken für die Qualitätssicherung von Softwaremodellen angewendet werden. Diese Techniken sind Modellmetriken, Smells sowie Refactorings und werden einerseits zur Analyse (Metriken und Smells) und andererseits zur Verbesserung der Qualität (Refactorings) der Modelle verwendet.

Das Tutorium wird dabei zeitlich strukturiert unterteilt. Für jede Aufgabe wird eine Bearbeitungszeit von **20 Minuten** eingeplant. **Bitte bearbeiten Sie so viele Aufgaben wie möglich und beantworten Sie bitte die Frage bezüglich der empfundenen Schwierigkeit der jeweiligen Aufgabe!** Die Aufgabenblätter werden nach Ablauf der Bearbeitungszeit wieder eingesammelt. Für die Bearbeitung der Aufgaben können die verteilten Handouts verwenden werden. In Ausnahmefällen können Sie auch den Tutor um Rat fragen. Am Ende des Tutoriums wird ein Fragebogen mit allgemeinen Fragen zu Ihren Beobachtungen bzw. Erfahrungen, die Sie bei der Bearbeitung der Aufgaben gemacht haben, verteilt. Beantworten Sie bitte die aufgeführten Fragen und geben Sie den Fragenbogen ausgefüllt an den Tutor zurück.

Verwenden Sie heute bitte die bereits auf den Fachbereichsrechnern vorinstallierte Version der **Eclipse Modeling Tools**. Nach der Anmeldung mit Ihrem **Fachbereich-Account** finden Sie diese unter

D:\est1314\eclipse.

Starten Sie Eclipse mit einem Doppelklick auf die Datei **eclipse.exe**. Stellen Sie sicher, dass Sie den auf

D:\est1314\ws

befindlichen Workspace verwenden. Anderenfalls wechseln Sie zu diesem (***File → Switch Workspace → D:\est1314\ws***). In diesem Workspace befindet sich das Projekt

de.unimarburg.swt.est

mit dem bereits vorgestellten UML-Klassenmodell aus Abbildung 1.

Bitte beachten Sie, dass die Ergebnisse dieses Tutoriums NICHT in die Benotung dieses Moduls eingehen! Sie dienen lediglich einer Studie im Rahmen der Forschungstätigkeiten des Tutors.

Vielen Dank für Ihre Teilnahme!!!

Thorsten Arendt

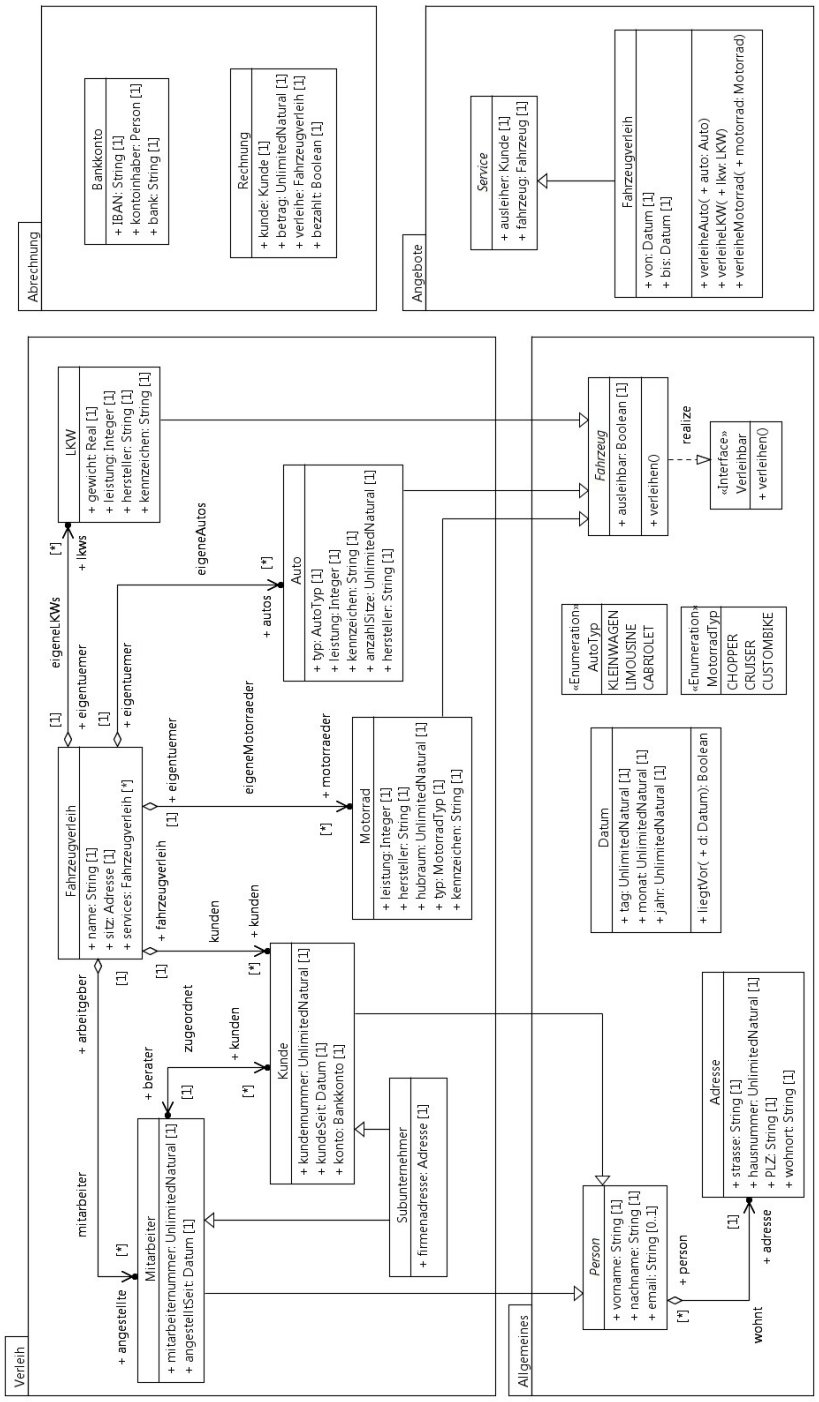


Abb. 1: Erstes Domänenmodell eines Fahrzeugverleihs als UML Klassendiagramm

1 Berechnen von Metriken für UML-Klassenmodelle

Ziel dieser Aufgabe ist es, vorgegebene Metriken für das in Abbildung 1 gezeigte Domänenmodell eines Fahrzeugverleihs in Form eines UML-Klassendiagramms zu berechnen.

Für die Bearbeitung dieser Aufgaben haben Sie insgesamt **20 Minuten** Zeit.

Bitte bearbeiten Sie so viele Aufgaben wie möglich und beantworten Sie bitte die Fragen bezüglich der empfundenen Schwierigkeit der jeweiligen Teilaufgabe sowie bezüglich der verfügbaren Zeit!

Aufgabe 1.1

Berechnen Sie bitte die folgenden Metriken auf Modellebene und tragen Sie die Ergebnisse bitte in die vorgesehenen Zellen der folgenden Tabelle ein.

Metrik	Beschreibung	Wert
AKLM	Anzahl aller Klassen im Modell	
AVBM	Anzahl aller Vererbungs-Beziehungen zwischen Klassen	
VBvsKL	Durchschnittliche Anzahl an Vererbungs-Beziehungen pro Klasse	
AVHM	Anzahl aller Vererbungs-Hierarchien im Modell	
MaxDIT	Maximale Tiefe der Vererbungsbäume im Modell	
AATM	Gesamtanzahl der Attribute aller Klassen im Modell (eigene und geerbte)	
ATvsKL	Durchschnittliche Anzahl an Attributen pro Klasse (eigene und geerbte)	
AOPM	Gesamtanzahl der Operationen aller Klassen im Modell (eigene und geerbte)	
OPvsKL	Durchschnittliche Anzahl an Operationen pro Klasse (eigene und geerbte)	
AELM	Gesamtanzahl der Elemente im Modell (Pakete, Klassen, Interfaces, Attribute, Operationen, Parameter, Assoziationen, Assoziationsenden, Vererbungs-Beziehungen, Interface-Realisierungen, Aufzählungen, und Aufzählungs-Literale)	

Frage 4.1.1

Wie empfanden Sie den Schwierigkeitsgrad dieser Aufgabe (Metrikenberechnung auf Modellebene)?

1	2	3	4	5
sehr einfach	eher einfach	angemessen	eher schwer	sehr schwer

Aufgabe 1.2

Berechnen Sie bitte die folgenden Metriken auf Paketebene für die Pakete ***Verleih*** und ***Allgemeines*** und tragen Sie die Ergebnisse bitte in die vorgesehenen Zellen der folgenden Tabelle ein.

Metrik	Beschreibung	Verleih	Allgemeines
AKLP	Anzahl der Klassen im Paket		
AASP	Anzahl der Assoziationen im Paket		
ASvsKL	Verhältnis zwischen der Anzahl der Assoziationen und der Anzahl der Klassen im Paket		
AP	Anteil der abstrakten Klassen an der Gesamtanzahl aller Klassen im Paket (Abstraktheitsgrad)		
AATKLP	Anzahl aller Attribute in Klassen im Paket (auch geerbte)		
AOPKLP	Anzahl aller Operationen in Klassen im Paket (auch geerbte)		
Ca	Afferent coupling: Anzahl der Klassen in anderen Paketen, die von Klassen innerhalb des Paketes abhängen (Abhängigkeit bedeutet: Typ eines Attributes, einer Operation oder eines Parameters; Superklasse)		
Ce	Efferent coupling: Anzahl der Klassen in anderen Paketen, von denen die Klassen innerhalb des Paketes abhängen (Abhängigkeit wie oben)		
TC	Total coupling: afferent coupling + efferent coupling		
I	Verhältnis zwischen efferent coupling und total coupling (Instabilität)		

Frage 4.1.2

Wie empfanden Sie den Schwierigkeitsgrad dieser Aufgabe (Metrikenberechnung auf Paketebene)?

1	2	3	4	5
sehr einfach	eher einfach	angemessen	eher schwer	sehr schwer

Aufgabe 1.3

Berechnen Sie bitte die folgenden Metriken auf Klassenebene für die Klassen ***Verleih::Fahrzeugverleih***, ***Allgemeines::Datum*** und ***Verleih::Motorrad*** und tragen Sie die Ergebnisse bitte in die vorgesehenen Zellen der folgenden Tabelle ein.

Metrik	Beschreibung	Fahrzeugverleih	Datum	Motorrad
ASUPKL	Anzahl aller Superklassen der Klasse (transitive Hülle)			
MaxDITK	Tiefe in der Vererbungs-Hierarchie (Maximum aufgrund Mehrfachvererbung)			
AAEATKL	Anzahl der eigenen Attribute der Klasse, die in allen Geschwisterklassen äquivalente Attribute haben			
AATKL	Anzahl der Attribute der Klasse (eigene und geerbte)			
AATPTKL	Anzahl der Attribute der Klasse mit primitivem Datentyp (eigene und geerbte)			
AOPKL	Anzahl der Operationen der Klasse (eigene und geerbte)			
AASKL	Anzahl der Assoziationen mit anderen Klassen oder mit sich selbst (auch geerbte Assoziationen)			
AFEKL	Anzahl der Features (Attribute und Operationen) der Klasse (eigene und geerbte)			
CBC	Coupling between classes: Anzahl der Attribute der Klasse sowie der navigierbaren Assoziationsenden, die eine andere Klasse als Typ haben (eigene und geerbte)			
AEBKL	Anzahl der externen Benutzungen der Klasse als Typ von Attributen, Operationen und Parametern (auch ihrer Superklassen)			

Frage 4.1.3

Wie empfanden Sie den Schwierigkeitsgrad dieser Aufgabe (Metrikenberechnung auf Klassenebene)?

1	2	3	4	5
sehr einfach	eher einfach	angemessen	eher schwer	sehr schwer

Frage 4.1.4

Wie empfanden Sie die für diesen Aufgabenteil zur Verfügung stehende Zeit?

1	2	3	4	5
viel zu kurz	eher zu kurz	angemessen	eher zu lang	viel zu lang

2 Auffinden von Model Smells in UML-Klassenmodellen

Ziel dieser Aufgabe ist es, Model Smells in dem in Abbildung 1 gezeigten Domänenmodell eines Fahrzeugverleihs in Form eines UML-Klassendiagramms zu finden.

Für die Bearbeitung dieser Aufgaben haben Sie insgesamt **20 Minuten** Zeit.

Bitte bearbeiten Sie so viele Aufgaben wie möglich und beantworten Sie bitte die Fragen bezüglich der empfundenen Schwierigkeit sowie bezüglich der zur Verfügung stehenden Zeit!

Aufgabe 2.1

Untersuchen Sie das Modell hinsichtlich der folgenden Model Smells. Notieren Sie bitte jeweils die Anzahl der gefundenen Smell-Vorkommen in die vorgesehenen Zellen der ersten Tabelle. In der zweiten Tabelle notieren Sie bitte für jeden gefundenen Smell die involvierten Elemente.

Model Smell	Beschreibung	Anzahl
Konkrete Superklasse	Eine abstrakte Klasse besitzt eine konkrete Superklasse.	
Diamond Problem	Eine Klasse erbt mehrfach von einer anderen Klasse.	
Gleiche Klassennamen	Zwei in unterschiedlichen Paketen definierte Klassen haben den gleichen Namen.	
Keine Spezifikation	Eine abstrakte Klasse besitzt keine konkrete Unterklasse.	
Spekulative Allgemeinheit	Eine abstrakte Klasse besitzt nur eine einzige Unterklasse.	
Unbenutzte Klasse	Eine Klasse hat keine Ober- oder Unterklasse, ist nicht mit einem Interface assoziiert und ist nicht Typ eines externen Attributes, einer Operation oder Parameters.	
Äquivalente Attribute	Jede Geschwisterklasse der besitzenden Klasse eines Attributes besitzt ein äquivalentes Attribut.	
Abstraktes Paket	Ein Paket hat einen zu hohen Anteil an abstrakten Klassen (hier: höher als **0.4**).	
Primitive Obsession	Eine Klasse besitzt mehr Attribute mit primitivem Datentyp als der angegebene Grenzwert (hier: mehr als **3**).	
Große Klasse	Eine Klasse besitzt mehr Features (Attribute und Operationen; auch geerbte) als der angegebene Grenzwert (hier: mehr als **6**).	

Gefundener Model Smell	Involvierte Elemente

Frage 4.2.1

Wie empfanden Sie den Schwierigkeitsgrad dieses Aufgabenteils (Erkennen von Model Smells)?

1	2	3	4	5
sehr einfach	eher einfach	angemessen	eher schwer	sehr schwer

Frage 4.2.2

Wie empfanden Sie die für diesen Aufgabenteil zur Verfügung stehende Zeit?

1	2	3	4	5
viel zu kurz	eher zu kurz	angemessen	eher zu lang	viel zu lang

3 Durchführen von Refactorings auf UML-Klassendiagrammen

Ziel dieser Aufgabe ist es, die in Aufgabe 2 entdeckten Model Smells in dem in Abbildung 1 gezeigten Domänenmodell eines Fahrzeugverleihs (teilweise) zu eliminieren. Dazu werden neben manuellen Änderungen am Modell insbesondere Refactorings (wie in der Vorlesung vorgestellt) verwendet, die Sie in der unten stehenden Tabelle in der Spalte *Maßnahmen* finden.

Für die Bearbeitung dieser Aufgaben haben Sie insgesamt **20 Minuten** Zeit.

Bitte bearbeiten Sie so viele Aufgaben wie möglich und beantworten Sie bitte die Fragen bezüglich der empfundenen Schwierigkeit sowie bezüglich der zur Verfügung stehenden Zeit!

Nach der Bearbeitung dieser Aufgabe exportieren Sie bitte das Projekt in ein zip-Archiv (*File* → *Export...* → *General* → *Archive File* → *Select All* → *Browse* → … → *Save* → *Finish*) und senden Sie dieses bitte per Email an arendt@mathematik.uni-marburg.de.

Anschließend schließen Sie bitte das Projekt (*Rechtsklick auf Projekt* → *Close Project*) und beenden Sie bitte Eclipse.

Die zu bearbeitenden Aufgaben beinhalten die folgenden Refactorings:

- ***Pull Up Attribute***: Bei diesem Refactoring wird ein Attribut von einer Klasse in eine seiner direkten Superklassen verschoben. Voraussetzung ist, dass ALLE weiteren Unterklassen der Superklasse ein solches Attribut (gleicher Name, Typ, Multiplizität, Sichtbarkeit 'public', etc.) besitzen. Nach dem Refactoring besitzt die Superklasse nun dieses Attribut und die entsprechenden Attribute in den Unterklassen sind verschwunden.

- ***Remove Superclass***: Bei diesem Refactoring wird eine Superklasse aus dem Modell entfernt. Voraussetzung ist, dass diese Klasse mindestens eine Unterklasse besitzt. Sämtliche Features (Attribute, Operationen, etc.) der Klasse werden bei diesem Refactoring in ihre Unterklassen verschoben bzw. kopiert, sodass anschließend alle Unterklassen diese Features besitzen. Danach werden die Vererbungs-Beziehungen (Generalisierungen) zu der Klasse entfernt und letztendlich die nun leere Klasse aus dem Modell gelöscht.

- ***Rename Class***: Dieses Refactoring wird dazu benutzt, eine Klasse umzubenennen. Wichtig ist, dass der neue Klassenname noch nicht im entsprechenden Namespace (meistens das enthaltende Paket) vorhanden ist.

Gefundener Smell	Involvierte Elemente	Maßnahmen
Abstraktes Paket	Paket *Allgemeines*	---
	Paket *Angebote*	siehe Smell *Spekulative Allgemeinheit*
Diamond Problem	Klassen *Subunternehmer*, *Mitarbeiter*, *Kunde* und *Person*	---
Äquivalente Attribute	Klasse *LKW*; Attribut *leistung*	Refactoring *Pull Up Attribute* auf *leistung*
	Klasse *LKW*; Attribut *hersteller*	Refactoring *Pull Up Attribute* auf *hersteller*
	Klasse *LKW*; Attribut *kennzeichen*	Refactoring *Pull Up Attribute* auf *kennzeichen*
	Klasse *Auto*; Attribut *leistung*	s.o.
	Klasse *Auto*; Attribut *hersteller*	s.o.
	Klasse *Auto*; Attribut *kennzeichen*	s.o.
	Klasse *Motorrad*; Attribut *leistung*	s.o.
	Klasse *Motorrad*; Attribut *hersteller*	s.o.
	Klasse *Motorrad*; Attribut *kennzeichen*	s.o.
Gleiche Klassennamen	Klassen *Verleih::Fahrzeugverleih* und *Angebote::Fahrzeugverleih*	Refactoring *Rename Class* auf *Angebote::Fahrzeugverleih* zu *Fahrzeugvermietung*
Große Klasse	Klasse *Verleih::Fahrzeugverleih*	---
	Klasse *Kunde*	---
	Klasse *Subunternehmer*	---
Spekulative Allgemeinheit	Klassen *Angebote::Service* und *Angebote::Fahrzeugverleih*	Refactoring *Remove Superclass* auf *Angebote::Service*
Unbenutzte Klasse	Klasse *Rechnung*	Manuelle Änderung: Hinzufügen eines neuen öffentlichen Attributes *rechnungen* vom Typ *Rechnung* mit Multiplizität *0..** zu Klasse *Verleih::Fahrzeugverleih*

Frage 4.3.1

Wie empfanden Sie den Schwierigkeitsgrad dieses Aufgabenteils (Durchführen von Modell-Refactorings)?

1	2	3	4	5
sehr einfach	eher einfach	angemessen	eher schwer	sehr schwer

Frage 4.3.2

Wie empfanden Sie die für diesen Aufgabenteil zur Verfügung stehende Zeit?

1	2	3	4	5
viel zu kurz	eher zu kurz	angemessen	eher zu lang	viel zu lang

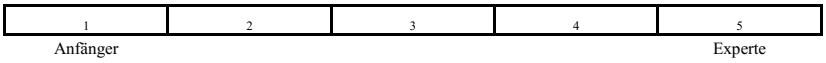
4 Fragebogen

Zum Abschluss des heutigen Tutoriums bitte ich Sie, die folgenden Fragen zu beantworten. Dabei geht es, wie bei der Bearbeitung der vorhergehenden Aufgaben nicht darum, die Antworten im "Sinne des Tutors" zu geben. Im Gegenteil: Für die Auswertung des Fragenbogens sind insbesondere ehrliche Antworten von Interesse! (Auch hier noch einmal der Hinweis: **Es erfolgt keine Benotung!!!**)

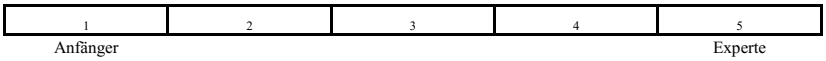
Für die Beantwortung der Fragen haben Sie maximal **10 Minuten** Zeit.

Frage 4.0.1
In welchem Studiengang und in welchem Semester studieren Sie?

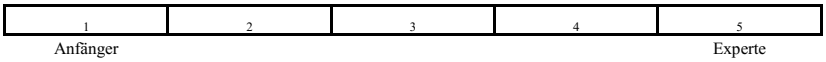
Frage 4.0.2
Welche Erfahrungen haben Sie mit der Modellierung mit UML, insbesondere mit Klassendiagrammen?



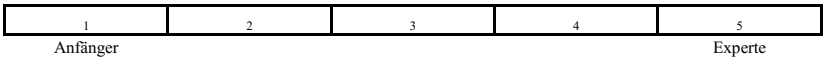
Frage 4.0.3
Welche Erfahrungen haben Sie mit der Berechnung von Softwaremetriken, insbesondere Modellmetriken?



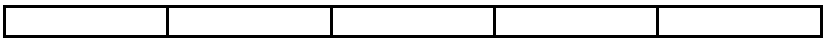
Frage 4.0.4
Welche Erfahrungen haben Sie mit dem Auffinden von Bad Smells, insbesondere Model Smells?



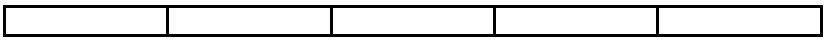
Frage 4.0.5
Welche Erfahrungen haben Sie mit der Durchführung von Refactorings, insbesondere Modell-Refactorings?



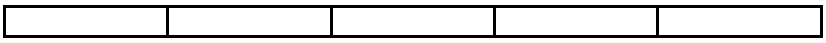
Frage 4.4.1
Wie oft dachten Sie während des ersten Aufgabenteils (**Berechnung von Metriken**) daran, dass das verwendete Modellierungswerkzeug (hier: Eclipse Papyrus) eine solche Funktionalität bereitstellen sollte?



Frage 4.4.2
Wie oft dachten Sie während des zweiten Aufgabenteils (**Auffinden von Model Smells**) daran, dass das verwendete Modellierungswerkzeug (hier: Eclipse Papyrus) eine solche Funktionalität bereitstellen sollte?



Frage 4.4.3
Wie oft dachten Sie während des dritten Aufgabenteils (**Durchführen von Refactorings**) daran, dass das verwendete Modellierungswerkzeug (hier: Eclipse Papyrus) eine solche Funktionalität bereitstellen sollte?



Frage 4.5
Wie fanden Sie dieses Tutorium? Was hat Ihnen gefallen, was nicht? Was könnte verbessert werden?

1 Berechnen von Metriken für UML-Klassenmodelle

Ziel dieser Aufgabe ist es, vorgegebene Metriken für das in Abbildung 1 gezeigte Domänenmodell eines Fahrzeugverleihs in Form eines UML-Klassendiagramms zu berechnen.

Benutzen Sie dazu bitte die entsprechende Funktionalität von EMF Refactor und beachten Sie bitte die folgende Vorgehensweise:

- Aktivieren Sie die zu berechnende(n) Metrik(en) in den Projekteigenschaften (*Projekt markieren → Project → Properties → EMF Quality Assurance → Metrics Configuration → Metriken auswählen → OK*).
- Starten Sie eine Metriken-Berechnung, indem Sie das entsprechende Kontextelement im graphischen Editor auswählen und im Kontextmenü (Rechtsklick) ***EMF Quality Assurance (use existing techniques) → Calculate Configured Metrics (on element)*** auswählen. (Für Metriken auf Modellebene klicken Sie bitte auf einen Zwischenraum zwischen den Paketen.)

Für die Bearbeitung dieser Aufgaben haben Sie insgesamt **20 Minuten** Zeit.

Bitte bearbeiten Sie so viele Aufgaben wie möglich und beantworten Sie bitte die Fragen bezüglich der empfundenen Schwierigkeit der jeweiligen Teilaufgabe sowie bezüglich der verfügbaren Zeit!

Aufgabe 1.1

Berechnen Sie bitte die folgenden Metriken auf Modellebene und tragen Sie die Ergebnisse bitte in die vorgesehenen Zellen der folgenden Tabelle ein.

Metrik	Beschreibung	Wert
AKLM	Anzahl aller Klassen im Modell	
AVBM	Anzahl aller Vererbungs-Beziehungen zwischen Klassen	
VBvsKL	Durchschnittliche Anzahl an Vererbungs-Beziehungen pro Klasse	
AVHM	Anzahl aller Vererbungs-Hierarchien im Modell	
MaxDIT	Maximale Tiefe der Vererbungsbäume im Modell	
AATM	Gesamtanzahl der Attribute aller Klassen im Modell (eigene und geerbte)	
ATvsKL	Durchschnittliche Anzahl an Attributen pro Klasse (eigene und geerbte)	
AOPM	Gesamtanzahl der Operationen aller Klassen im Modell (eigene und geerbte)	
OPvsKL	Durchschnittliche Anzahl an Operationen pro Klasse (eigene und geerbte)	
AELM	Gesamtanzahl der Elemente im Modell (Pakete, Klassen, Interfaces, Attribute, Operationen, Parameter, Assoziationen, Assoziationsenden, Vererbungs-Beziehungen, Interface-Realisierungen, Aufzählungen, und Aufzählungs-Literale)	

Frage 4.1.1

Wie empfanden Sie den Schwierigkeitsgrad dieser Aufgabe (Metrikenberechnung auf Modellebene)?

1	2	3	4	5
sehr einfach	eher einfach	angemessen	eher schwer	sehr schwer

Aufgabe 1.2

Berechnen Sie bitte die folgenden Metriken auf Paketebene für die Pakete ***Verleih*** und ***Allgemeines*** und tragen Sie die Ergebnisse bitte in die vorgesehenen Zellen der folgenden Tabelle ein.

Metrik	Beschreibung	Verleih	Allgemeines
AKLP	Anzahl der Klassen im Paket		
AASP	Anzahl der Assoziationen im Paket		
ASvsKL	Verhältnis zwischen der Anzahl der Assoziationen und der Anzahl der Klassen im Paket		
AP	Anteil der abstrakten Klassen an der Gesamtanzahl aller Klassen im Paket (Abstraktheitsgrad)		
AATKLP	Anzahl aller Attribute in Klassen im Paket (auch geerbte)		
AOPKLP	Anzahl aller Operationen in Klassen im Paket (auch geerbte)		
Ca	Afferent coupling: Anzahl der Klassen in anderen Paketen, die von Klassen innerhalb des Paketes abhängen (Abhängigkeit bedeutet: Typ eines Attributes, einer Operation oder eines Parameters; Superklasse)		
Ce	Efferent coupling: Anzahl der Klassen in anderen Paketen, von denen die Klassen innerhalb des Paketes abhängen (Abhängigkeit wie oben)		
TC	Total coupling: afferent coupling + efferent coupling		
I	Verhältnis zwischen efferent coupling und total coupling (Instabilität)		

Frage 4.1.2

Wie empfanden Sie den Schwierigkeitsgrad dieser Aufgabe (Metrikenberechnung auf Paketebene)?

1	2	3	4	5
sehr einfach	eher einfach	angemessen	eher schwer	sehr schwer

Aufgabe 1.3

Berechnen Sie bitte die folgenden Metriken auf Klassenebene für die Klassen ***Verleih::Fahrzeugverleih***, ***Allgemeines::Datum*** und ***Verleih::Motorrad*** und tragen Sie die Ergebnisse bitte in die vorgesehenen Zellen der folgenden Tabelle ein.

Metrik	Beschreibung	Fahrzeugverleih	Datum	Motorrad
ASUPKL	Anzahl aller Superklassen der Klasse (transitive Hülle)			
MaxDITK	Tiefe in der Vererbungs-Hierarchie (Maximum aufgrund Mehrfachvererbung)			
AAEATKL	Anzahl der eigenen Attribute der Klasse, die in allen Geschwisterklassen äquivalente Attribute haben			
AATKL	Anzahl der Attribute der Klasse (eigene und geerbte)			
AATPTKL	Anzahl der Attribute der Klasse mit primitivem Datentyp (eigene und geerbte)			
AOPKL	Anzahl der Operationen der Klasse (eigene und geerbte)			
AASKL	Anzahl der Assoziationen mit anderen Klassen oder mit sich selbst (auch geerbte Assoziationen)			
AFEKL	Anzahl der Features (Attribute und Operationen) der Klasse (eigene und geerbte)			
CBC	Coupling between classes: Anzahl der Attribute der Klasse sowie der navigierbaren Assoziationsenden, die eine andere Klasse als Typ haben (eigene und geerbte)			
AEBKL	Anzahl der externen Benutzungen der Klasse als Typ von Attributen, Operationen und Parametern (auch ihrer Superklassen)			

Frage 4.1.3

Wie empfanden Sie den Schwierigkeitsgrad dieser Aufgabe (Metrikenberechnung auf Klassenebene)?

1	2	3	4	5
sehr einfach	eher einfach	angemessen	eher schwer	sehr schwer

Frage 4.1.4

Wie empfanden Sie die für diesen Aufgabenteil zur Verfügung stehende Zeit?

1	2	3	4	5
viel zu kurz	eher zu kurz	angemessen	eher zu lang	viel zu lang

2 Auffinden von Model Smells in UML-Klassenmodellen

Ziel dieser Aufgabe ist es, Model Smells in dem in Abbildung 1 gezeigten Domänenmodell eines Fahrzeugverleihs in Form eines UML-Klassendiagramms zu finden.

Benutzen Sie dazu bitte die entsprechende Funktionalität von EMF Refactor und beachten Sie bitte die folgende Vorgehensweise:

- Aktivieren Sie die zu findenden Model Smells in den Projekteigenschaften (*Projekt markieren → Project → Properties → EMF Quality Assurance → Smells Configuration → Model Smells auswählen → OK*). Für Metrik-basierte Model Smells tragen Sie bitte hier auch den in der Aufgabenstellung angegebenen Grenzwert ein. **Vorsicht: Dezimaltrennzeichen ist der Punkt, also z.B. 4.2 statt 4,2!**
- Starten Sie eine Smell-Suche auf dem kompletten Modell, indem Sie im graphischen Editor auf einen Zwischenraum zwischen den Paketen klicken und im Kontextmenü (Rechtsklick) ***EMF Quality Assurance (use existing techniques) → Find Configured Model Smells*** auswählen.

Für die Bearbeitung dieser Aufgaben haben Sie insgesamt **20 Minuten** Zeit.

Bitte bearbeiten Sie so viele Aufgaben wie möglich und beantworten Sie bitte die Fragen bezüglich der empfundenen Schwierigkeit sowie bezüglich der zur Verfügung stehenden Zeit!

Aufgabe 2.1

Untersuchen Sie das Modell hinsichtlich der folgenden Model Smells. Notieren Sie bitte jeweils die Anzahl der gefundenen Smell-Vorkommen in die vorgesehenen Zellen der ersten Tabelle. In der zweiten Tabelle notieren Sie bitte für jeden gefundenen Smell die involvierten Elemente.

Model Smell	Beschreibung	Anzahl
Konkrete Superklasse	Eine abstrakte Klasse besitzt eine konkrete Superklasse.	
Diamond Problem	Eine Klasse erbt mehrfach von einer anderen Klasse.	
Gleiche Klassennamen	Zwei in unterschiedlichen Paketen definierte Klassen haben den gleichen Namen.	
Keine Spezifikation	Eine abstrakte Klasse besitzt keine konkrete Unterklasse.	
Spekulative Allgemeinheit	Eine abstrakte Klasse besitzt nur eine einzige Unterklasse.	
Unbenutzte Klasse	Eine Klasse hat keine Ober- oder Unterklasse, ist nicht mit einem Interface assoziiert und ist nicht Typ eines externen Attributes, einer Operation oder Parameters.	
Äquivalente Attribute	Jede Geschwisterklasse der besitzenden Klasse eines Attributes besitzt ein äquivalentes Attribut.	
Abstraktes Paket	Ein Paket hat einen zu hohen Anteil an abstrakten Klassen (hier: höher als **0.4**).	
Primitive Obsession	Eine Klasse besitzt mehr Attribute mit primitivem Datentyp als der angegebene Grenzwert (hier: mehr als **3**).	
Große Klasse	Eine Klasse besitzt mehr Features (Attribute und Operationen; auch geerbte) als der angegebene Grenzwert (hier: mehr als **6**).	

Gefundener Model Smell	Involvierte Elemente

Frage 4.2.1

Wie empfanden Sie den Schwierigkeitsgrad dieses Aufgabenteils (Erkennen von Model Smells)?

1	2	3	4	5
sehr einfach	eher einfach	angemessen	eher schwer	sehr schwer

Frage 4.2.2

Wie empfanden Sie die für diesen Aufgabenteil zur Verfügung stehende Zeit?

1	2	3	4	5
viel zu kurz	eher zu kurz	angemessen	eher zu lang	viel zu lang

3 Durchführen von Refactorings auf UML-Klassendiagrammen

Ziel dieser Aufgabe ist es, die in Aufgabe 2 entdeckten Model Smells in dem in Abbildung 1 gezeigten Domänenmodell eines Fahrzeugverleihs (teilweise) zu eliminieren. Dazu werden neben manuellen Änderungen am Modell insbesondere Refactorings (wie in der Vorlesung vorgestellt) verwendet, die Sie in der unten stehenden Tabelle in der Spalte *Maßnahmen* finden.

Benutzen Sie dazu bitte die entsprechende Funktionalität von EMF Refactor und beachten Sie bitte die folgende Vorgehensweise:

- Aktivieren Sie die zu verwendenden Refactorings in den Projekteigenschaften (*Projekt markieren → Project → Properties → EMF Quality Assurance → Refactorings Configuration → Refactorings auswählen → OK*).
- Starten Sie ein Refactoring, indem Sie im graphischen Editor das entsprechende **Kontextelement** auswählen (z.B. das zu verschiebende Attribut beim Refactoring *Pull Up Attribute)* und im Kontextmenü (Rechtsklick) ***Papyrus UML Model Refactorings → Name des Refactorings*** auswählen.

Für die Bearbeitung dieser Aufgaben haben Sie insgesamt **20 Minuten** Zeit.

Bitte bearbeiten Sie so viele Aufgaben wie möglich und beantworten Sie bitte die Fragen bezüglich der empfundenen Schwierigkeit sowie bezüglich der zur Verfügung stehenden Zeit!

Nach der Bearbeitung dieser Aufgabe exportieren Sie bitte das Projekt in ein zip-Archiv (*File → Export... → General → Archive File → Select All → Browse → … → Save → Finish*) und senden Sie dieses bitte per Email an arendt@mathematik.uni-marburg.de.

Anschließend schließen Sie bitte das Projekt (*Rechtsklick auf Projekt → Close Project*) und beenden Sie bitte Eclipse.

Die zu bearbeitenden Aufgaben beinhalten die folgenden Refactorings:

- ***Pull Up Attribute***: Bei diesem Refactoring wird ein Attribut von einer Klasse in eine seiner direkten Superklassen verschoben. Voraussetzung ist, dass ALLE weiteren Unterklassen der Superklasse ein solches Attribut (gleicher Name, Typ, Multiplizität, Sichtbarkeit 'public', etc.) besitzen. Nach dem Refactoring besitzt die Superklasse nun dieses Attribut und die entsprechenden Attribute in den Unterklassen sind verschwunden.

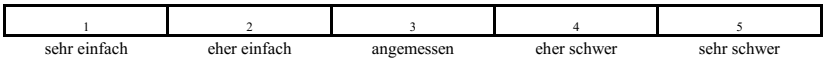
- ***Remove Superclass***: Bei diesem Refactoring wird eine Superklasse aus dem Modell entfernt. Voraussetzung ist, dass diese Klasse mindestens eine Unterklasse besitzt. Sämtliche Features (Attribute, Operationen, etc.) der Klasse werden bei diesem Refactoring in ihre Unterklassen verschoben bzw. kopiert, sodass anschließend alle Unterklassen diese Features besitzen. Danach werden die Vererbungs-Beziehungen (Generalisierungen) zu der Klasse entfernt und letztendlich die nun leere Klasse aus dem Modell gelöscht.

- ***Rename Class***: Dieses Refactoring wird dazu benutzt, eine Klasse umzubenennen. Wichtig ist, dass der neue Klassenname noch nicht im entsprechenden Namespace (meistens das enthaltende Paket) vorhanden ist.

Gefundener Smell	Involvierte Elemente	Maßnahmen
Abstraktes Paket	Paket *Allgemeines*	---
	Paket *Angebote*	siehe Smell *Spekulative Allgemeinheit*
Diamond Problem	Klassen *Subunternehmer*, *Mitarbeiter*, *Kunde* und *Person*	---
Äquivalente Attribute	Klasse *LKW*; Attribut *leistung*	Refactoring *Pull Up Attribute* auf *leistung*
	Klasse *LKW*; Attribut *hersteller*	Refactoring *Pull Up Attribute* auf *hersteller*
	Klasse *LKW*; Attribut *kennzeichen*	Refactoring *Pull Up Attribute* auf *kennzeichen*
	Klasse *Auto*; Attribut *leistung*	s.o.
	Klasse *Auto*; Attribut *hersteller*	s.o.
	Klasse *Auto*; Attribut *kennzeichen*	s.o.
	Klasse *Motorrad*; Attribut *leistung*	s.o.
	Klasse *Motorrad*; Attribut *hersteller*	s.o.
	Klasse *Motorrad*; Attribut *kennzeichen*	s.o.
Gleiche Klassennamen	Klassen *Verleih::Fahrzeugverleih* und *Angebote::Fahrzeugverleih*	Refactoring *Rename Class* auf *Angebote::Fahrzeugverleih* zu *Fahrzeugvermietung*
Große Klasse	Klasse *Verleih::Fahrzeugverleih*	---
	Klasse *Kunde*	---
	Klasse *Subunternehmer*	---
Spekulative Allgemeinheit	Klassen *Angebote::Service* und *Angebote::Fahrzeugverleih*	Refactoring *Remove Superclass* auf *Angebote::Service*
Unbenutzte Klasse	Klasse *Rechnung*	Manuelle Änderung: Hinzufügen eines neuen öffentlichen Attributes *rechnungen* vom Typ *Rechnung* mit Multiplizität *0..** zu Klasse *Verleih::Fahrzeugverleih*

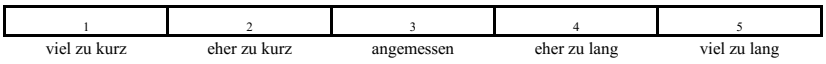
Frage 4.3.1

Wie empfanden Sie den Schwierigkeitsgrad dieses Aufgabenteils (Durchführen von Modell-Refactorings)?



Frage 4.3.2

Wie empfanden Sie die für diesen Aufgabenteil zur Verfügung stehende Zeit?



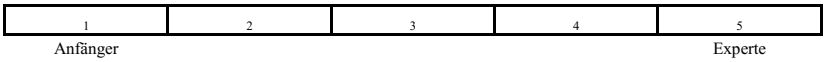
4 Fragebogen

Um Abschluss des heutigen Tutoriums bitte ich Sie, die folgenden Fragen zu beantworten. Dabei geht es, wie bei der Bearbeitung der vorhergehenden Aufgaben nicht darum, die Antworten im "Sinne des Tutors" zu geben. Im Gegenteil: Für die Auswertung des Fragenbogens sind insbesondere ehrliche Antworten von Interesse! (Auch hier noch einmal der Hinweis: **Es erfolgt keine Benotung!!!**)

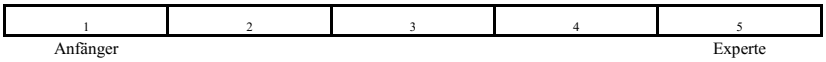
Für die Beantwortung der Fragen haben Sie maximal **10 Minuten** Zeit.

Frage 4.0.1
In welchem Studiengang und in welchem Semester studieren Sie?

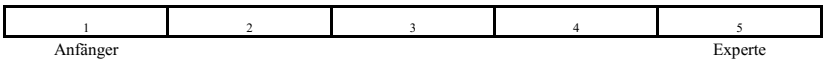
Frage 4.0.2
Welche Erfahrungen haben Sie mit der Modellierung mit UML, insbesondere mit Klassendiagrammen?



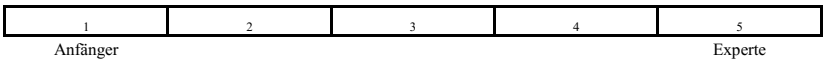
Frage 4.0.3
Welche Erfahrungen haben Sie mit der Berechnung von Softwaremetriken, insbesondere Modellmetriken?



Frage 4.0.4
Welche Erfahrungen haben Sie mit dem Auffinden von Bad Smells, insbesondere Model Smells?



Frage 4.0.5
Welche Erfahrungen haben Sie mit der Durchführung von Refactorings, insbesondere Modell-Refactorings?



Frage 4.4.1
In wie weit war die von EMF Refactor bereitgestellte Funktionalität hilfreich bei der Berechnung der Metriken in Aufgabenteil 1?

gar nicht hilfreich	wenig hilfreich	bedingt hilfreich	hilfreich	sehr hilfreich

Frage 4.4.2
In wie weit war die von EMF Refactor bereitgestellte Funktionalität hilfreich beim Auffinden von Model Smells in Aufgabenteil 2?

gar nicht hilfreich	wenig hilfreich	bedingt hilfreich	hilfreich	sehr hilfreich

Frage 4.4.3
In wie weit war die von EMF Refactor bereitgestellte Funktionalität hilfreich beim Durchführen von Refactorings in Aufgabenteil 3?

gar nicht hilfreich	wenig hilfreich	bedingt hilfreich	hilfreich	sehr hilfreich

Frage 4.5
Wie fanden Sie dieses Tutorium? Was hat Ihnen gefallen, was nicht? Was könnte verbessert werden?

H

STUDY MATERIAL EXPERIMENT EX_SPEC

This appendix contains material of the experiment `Ex_Spec` used in Chapter 14 of this thesis.

Philipps Universität Marburg

Prof. Dr. Gabriele Taentzer
Thorsten Arendt

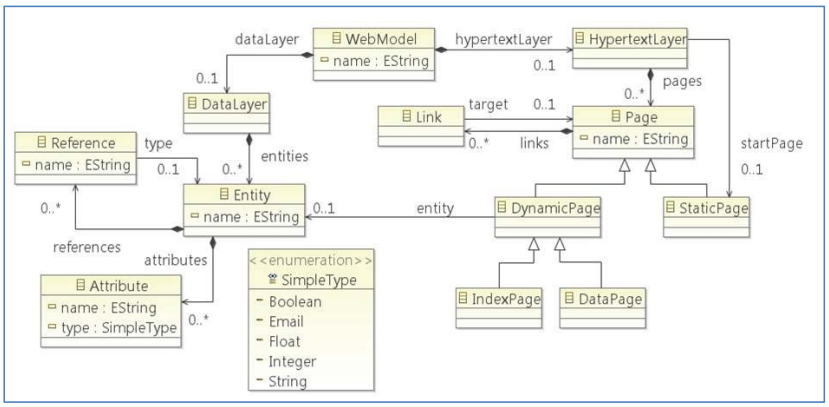
Philipps-Universität Marburg
Fachbereich Mathematik und Informatik

Modellgetriebene Softwareentwicklung

Tutorium 07

Wintersemester 13/14
Mittwoch, 11.12.2013
12.30Uhr - 16.00Uhr

Im Mittelpunkt dieses Tutoriums steht eine textuelle domänenspezifische Modellierungssprache für die Spezifikation von einfachen Webanwendungen namens Simple Web Modeling Language (SWM). Die Implementierung von SWM erfolgte mit Hilfe des im letzten Tutorium vorgestellten Frameworks Xtext gemäß der auf der nächsten Seite aufgeführten Grammatik. Die folgende Abbildung zeigt das zugehörige Ecore Metamodell:



Metamodell für SWM (Simple Web Modeling Language)

Im Rahmen dieses Tutoriums sollen Techniken für die Qualitätssicherung von Softwaremodellen spezifiziert und implementiert werden. Diese Techniken sind Modellmetriken, Smells sowie Refactorings und werden einerseits zur Analyse (Metriken und Smells) und andererseits zur Verbesserung der Qualität (Refactorings) der Modelle verwendet. Für die Ausführung der Techniken sowie für die Generierung des entsprechenden Codes wird die Werkzeugsammlung EMF Refactor verwendet. Spezifische Implementierungen sollen in Java sowie mit Hilfe von OCL-Queries durchgeführt werden, so wie sie bereits in der Vorlesung vorgestellt wurden.

Das Tutorium wird dabei zeitlich strukturiert unterteilt. Für jede Aufgabe wird eine Bearbeitungszeit von **40 Minuten** eingeplant. **Bitte notieren Sie die von Ihnen benötigte Bearbeitungszeit für jede Teilaufgabe auf diesem Aufgabenblatt in den dafür vorgesehenen Kästchen und beantworten Sie bitte die Frage bezüglich der empfundenen Schwierigkeit der jeweiligen Teilaufgabe!** Die Aufgabenblätter werden nach Ablauf der Bearbeitungszeit wieder eingesammelt. Für die Bearbeitung der Aufgaben können die verteilten Handouts verwenden werden. In Ausnahmefällen können Sie auch den Tutor um Rat fragen.

Im letzten Teil des Tutoriums wird Ihnen ein Fragebogen mit allgemeinen Fragen zu Ihren Beobachtungen bzw. Erfahrungen, die Sie bei der Bearbeitung der Aufgaben gemacht haben, verteilt. Beantworten Sie bitte die aufgeführten Fragen und geben Sie den Fragenbogen ausgefüllt an den Tutor zurück.

Bitte beachten Sie, dass die Ergebnisse dieses Tutoriums NICHT in die Benotung dieses Moduls eingehen! Sie dienen lediglich einer Studie im Rahmen der Forschungstätigkeiten des Tutors.

Vielen Dank für Ihre Teilnahme!!!

Thorsten Arendt

```
grammar org.eclipse.emf.refactor.examples.SimpleWebModel with
                                                    org.eclipse.xtext.common.Terminals

generate simpleWebModel "http://www.eclipse.org/SimpleWebModel/1.0"

WebModel:
	'webmodel' name=ID '{'
		dataLayer=DataLayer
		hypertextLayer=HypertextLayer
	'}' ;

DataLayer:
	'data {' {DataLayer}
		entities+=Entity*
	'}' ;

Entity:
	'entity' name=ID '{'
		attributes+=Attribute*
		references+=Reference*
	'}' ;

Attribute:
	'att' name=ID ':' type=SimpleType
;

enum SimpleType:
	Boolean | Email | Float | Integer | String
;

Reference:
	'ref' name=ID ':' type=[Entity]
;

HypertextLayer:
	'hypertext {'
		pages+=Page+
		'start page is' startPage=[StaticPage]
	'}' ;

Page:  StaticPage | DynamicPage ;

StaticPage:
	'static page' name=ID '{'
	links+=Link*
	'}' ;

Link:
	'link to page' target=[Page]
;

DynamicPage:  IndexPage | DataPage  ;

IndexPage:
	'index page' name=ID ('shows entity' entity=[Entity])?  '{'
	links+=Link*
	'}' ;

DataPage:
	'data page' name=ID ('shows entity' entity=[Entity])?  '{'
	links+=Link*
	'}' ;
```

Xtext Grammatik für SWM (Simple Web Modeling Language)

0 Aufsetzen der Umgebung

Verwenden Sie auch heute bitte die bereits in den vergangenen Tutorien benutzte *Eclipse Modeling Tools*-Distribution. Falls Sie diese noch nicht installiert haben, finden Sie den Download (Windows-Version) sowie eine Beschreibung der enthaltenen Komponenten (zum Zusammenstellen für Mac/Linux-User) unter *http://www.mathematik.uni-marburg.de/~swt/downloads/mdd1314/*. Führen Sie bitte zusätzlich die folgenden Schritte durch:

- Installieren Sie bitte EMF Refactor sowie die Komponenten für SWM: *Help → Install New Software… → Work with: http://download.eclipse.org/emf-refactor/updatesite-mdd1314/ → Select All → Finish → …*
- Setzen Sie bitte einen neuen, leeren Workspace auf: *File → Switch Workspace → Other… → Browse → … → OK → OK*
- Downloaden Sie das Archiv *http://www.eclipse.org/emf-refactor/downloads/mqa.zip* und importieren Sie die dort enthaltenen drei Eclipse-Projekte *de.pum.swt.mdd.swm.metrics*, *de.pum.swt.mdd.swm.smells* und *de.pum.swt.mdd.swm.refactorings* in Ihren Workspace: *File → Import... → General → Existing Projects into Workspace → Next → Select archive file → Browse → … → Finish*
- Starten Sie bitte eine neue Eclipse-Instanz (Laufzeitumgebung). Hier können später die implementierten Qualitätssicherungstechniken getestet werden.
- Downloaden Sie das Archiv *http://www.eclipse.org/emf-refactor/downloads/project.zip* und importieren Sie das dort enthaltene Eclipse-Projekt in den Workspace Ihrer Laufzeitumgebung: *File → Import... → General → Existing Projects into Workspace → Next → Select archive file → Browse → … → Finish*
- Öffnen Sie dort das Beispiel-Modell *Models/VehicleRentalCompany.swmt*. Der Editor müsste bei erfolgreicher Installation beispielsweise ein Highlighting der Schlüsselwörter *webmodel*, *data*, *entity* etc. bereitstellen.

Wichtig ist, dass ALLE Teilnehmer nach diesen Schritten die gleiche Umgebung erfolgreich aufgesetzt haben, um die folgenden Aufgaben zu bearbeiten!

Nach der Bearbeitung der Aufgaben exportieren Sie bitte die drei Projekte in ein zip-Archiv (*File → Export... → General → Archive File → Select All → Browse → … → Save → Finish*) und senden Sie dieses bitte per Email an arendt@mathematik.uni-marburg.de.

1 Spezifizieren und Implementieren von Metriken für SWM-Modelle

Ziel dieser Aufgabe ist es, Metriken für SWM-Modelle zu spezifizieren und mit Hilfe von EMF Refactor zu implementieren. Gehen Sie dabei bitte gemäß der in der Vorlesung demonstrierten und der Ihnen in dem Handout dargestellten Art und Weise vor. Für jede Metrik können Sie entscheiden, ob Sie eine konkrete Implementierung mit Java oder OCL bevorzugen oder ob die zu implementierende Metrik als Kombination aus bereits vorhandenen Metriken spezifiziert werden kann. Generieren Sie den Code bitte jeweils in das Projekt ***de.pum.swt.mdd.swm.metrics*** und vergeben Sie für jede Metrik eine neue Id.

Zum Testen (siehe Handout) verwenden Sie bitte das Beispiel-Modell *Models/VehicleRental Company.swmt* in der Laufzeitumgebung. **(Metrik zunächst in den Projekteigenschaften aktivieren!)**

Für die Bearbeitung dieser Aufgabe haben Sie insgesamt **40 Minuten** Zeit.

Bitte notieren Sie die von Ihnen benötigte Bearbeitungszeit für jede Teilaufgabe auf diesem Aufgabenblatt in den dafür vorgesehenen Kästchen und beantworten Sie bitte die Frage bezüglich der empfundenen Schwierigkeit der jeweiligen Teilaufgabe!

Aufgabe 1.1

Implementieren Sie eine Metrik, die die Anzahl aller dynamischen Seiten im Modell ermittelt.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

Aufgabe 1.2

Implementieren Sie eine Metrik, die die Anzahl aller Referenzen von Entitäten im Modell ermittelt.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

Aufgabe 1.3

Implementieren Sie eine Metrik, die die durchschnittliche Anzahl der Attribute pro Entität im Modell ermittelt.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

Aufgabe 1.4

Implementieren Sie eine Metrik, die die Anzahl der ausgehenden Referenzen einer Entität ermittelt.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

Aufgabe 1.5

Implementieren Sie eine Metrik, die die Anzahl derjenigen dynamischen Seiten ermittelt, die eine gegebene Entität referenzieren.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

2 Spezifizieren und Implementieren von Smells für SWM-Modelle

Ziel dieser Aufgabe ist es, Smells für SWM-Modelle zu spezifizieren und mit Hilfe von EMF Refactor zu implementieren. Gehen Sie dabei bitte gemäß der in der Vorlesung demonstrierten und der Ihnen in dem Handout dargestellten Art und Weise vor. Jeder Model Smell soll dabei in Java implementiert werden. Generieren Sie den Code bitte jeweils in das Projekt ***de.pum.swt.mdd.swm.smells*** und vergeben Sie für jeden Model Smell eine neue Id.

Zum Testen (siehe Handout) verwenden Sie bitte das Beispiel-Modell *Models/VehicleRental Company.swmt* in der Laufzeitumgebung. **(Smell zunächst in den Projekteigenschaften aktivieren!)**

Für die Bearbeitung dieser Aufgabe haben Sie insgesamt **40 Minuten** Zeit.

Bitte notieren Sie die von Ihnen benötigte Bearbeitungszeit für jede Teilaufgabe auf diesem Aufgabenblatt in den dafür vorgesehenen Kästchen und beantworten Sie bitte die Frage bezüglich der empfundenen Schwierigkeit der jeweiligen Teilaufgabe!

Aufgabe 2.1

Implementieren Sie einen Model Smell, der leere Entitäten, also Entitäten, die weder Attribute noch ausgehende Referenzen besitzen, im Modell entdeckt.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

Aufgabe 2.2

Implementieren Sie einen Model Smell, der Paare von Hypertext-Seiten im Modell entdeckt, die den gleichen Namen besitzen.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

Aufgabe 2.3

Implementieren Sie einen Model Smell, der Entitäten im Modell entdeckt, die von keiner dynamischen Seite referenziert werden.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

Aufgabe 2.4

Implementieren Sie einen Model Smell, der redundante Links, also Links zwischen den gleichen Hypertext-Seiten, im Modell entdeckt.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

3 Spezifizieren und Implementieren von Refactorings für SWM-Modelle

Ziel dieser Aufgabe ist es, Refactorings für SWM-Modelle zu spezifizieren und mit Hilfe von EMF Refactor zu implementieren. Gehen Sie dabei bitte gemäß der in der Vorlesung demonstrierten und der Ihnen in dem Handout dargestellten Art und Weise vor. Die Teile eines jeden Model Refactoring sollen dabei jeweils in Java implementiert werden. Generieren Sie den Code bitte jeweils in das Projekt ***de.pum.swt.mdd.swm.refactorings*** und vergeben Sie für jedes Model Refactoring eine neue Id.

Zum Testen (siehe Handout) verwenden Sie bitte das Beispiel-Modell *Models/VehicleRental Company.swmt* in der Laufzeitumgebung. **(Refactoring zunächst in den Projekteigenschaften aktivieren!)**

Für die Bearbeitung dieser Aufgabe haben Sie insgesamt **40 Minuten** Zeit.

Bitte notieren Sie die von Ihnen benötigte Bearbeitungszeit für jede Teilaufgabe auf diesem Aufgabenblatt in den dafür vorgesehenen Kästchen und beantworten Sie bitte die Frage bezüglich der empfundenen Schwierigkeit der jeweiligen Teilaufgabe!

Aufgabe 3.1

Implementieren Sie ein Model Refactoring, das eine Hypertext-Seite umbenennt. Dabei ist nach der Eingabe des neuen Namens zu überprüfen, ob nicht schon eine Seite mit diesem Namen besteht.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

Aufgabe 3.2

Implementieren Sie ein Model Refactoring, das von einer Hypertext-Seite ausgehende redundante Links entfernt.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

Aufgabe 3.3

Implementieren Sie ein Model Refactoring, das zu einer bisher nicht von dynamischen Hypertext-Seiten referenzierten Entität jeweils eine Index- und eine Daten-Seite erstellt, die diese Entität dann referenzieren. Weiterhin soll von der neuen Index-Seite ein Link zu der neuen Daten-Seite erstellt werden.

Bearbeitet (J/N)	Benötigte Zeit (Minuten)	Diese Aufgabe empfand ich als …				
		sehr schwer	eher schwer	angemessen	eher leicht	sehr leicht

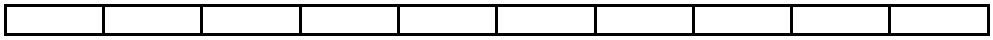
4 Fragebogen

Um Abschluss des heutigen Tutoriums bitte ich Sie, die folgenden Fragen zu beantworten. Dabei geht es, wie bei der Bearbeitung der vorhergehenden Aufgaben nicht darum, die Antworten im "Sinne des Tutors" zu geben. Im Gegenteil: Für die Auswertung des Fragenbogens sind insbesondere ehrliche Antworten von Interesse! (Auch hier noch einmal der Hinweis: **Es erfolgt keine Benotung!!!**)

Für die Beantwortung der Fragen haben Sie maximal **30 Minuten** Zeit.

Frage 4.0.1

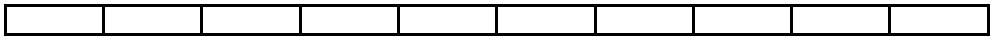
Wie schätzen Sie Ihre Qualifikation bezüglich der Programmierung mit Java ein?



Anfänger Experte

Frage 4.0.2

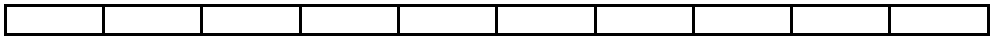
Wie schätzen Sie Ihre Qualifikation hinsichtlich der Verwendung von OCL ein?



Anfänger Experte

Frage 4.0.3

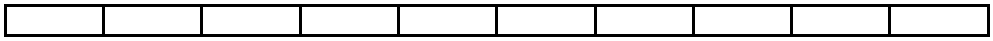
Wie schätzen Sie Ihre Qualifikation bezüglich hinsichtlich des Umgangs mit EMF ein?



Anfänger Experte

Frage 4.1.1

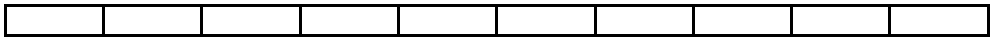
Wie beurteilen Sie den Schwierigkeitsgrad des ersten Aufgabenteils (Implementierung von Metriken)?



sehr einfach sehr schwer

Frage 4.1.2

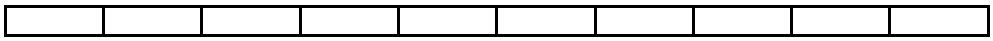
In wie weit war die von EMF Refactor bereitgestellte Codegenerierung hilfreich für die Implementierung der Metriken?



gar nicht hilfreich sehr hilfreich

Frage 4.1.3

War die Auswahlmöglichkeit der Implementierungssprache (Java oder OCL) hilfreich bei der Implementierung der Metriken?



gar nicht hilfreich sehr hilfreich

Frage 4.2.1
Wie beurteilen Sie den Schwierigkeitsgrad des zweiten Aufgabenteils (Implementierung von Model Smells)?

sehr einfach									sehr schwer

Frage 4.2.2
In wie weit war die von EMF Refactor bereitgestellte Codegenerierung hilfreich für die Implementierung der Model Smells?

gar nicht hilfreich									sehr hilfreich

Frage 4.3.1
Wie beurteilen Sie den Schwierigkeitsgrad des dritten Aufgabenteils (Implementierung von Model Refactorings)?

sehr einfach									sehr schwer

Frage 4.3.2
In wie weit war die von EMF Refactor bereitgestellte Codegenerierung hilfreich für die Implementierung der Model Refactorings?

gar nicht hilfreich									sehr hilfreich

Frage 4.4
Wie beurteilen Sie den Schwierigkeitsgrad der Aufgaben des heutigen Tutoriums insgesamt?

sehr einfach									sehr schwer

Frage 4.4.1
Warum haben Sie den Schwierigkeitsgrad der Aufgaben des heutigen Tutoriums so bewertet?

BIBLIOGRAPHY

[1] SDMetrics, 2014. URL http://www.sdmetrics.com/.

[2] SPES 2020. Software Plattform Embedded Systems, 2014. URL http://spes2020.informatik.tu-muenchen.de/home.html.

[3] Scott W. Ambler. *The Elements of UML Style*. Cambridge University Press, 2002.

[4] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Model Driven Engineering Languages and Systems (MoDELS)*, volume 6394 of *LNCS*, pages 121–135, 2010.

[5] Dave Astels. Refactoring with UML. In *Proc. 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, 2002.

[6] J. Bansiya and C.G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[7] Gabriel Barbier, Hugo Brunelière, Frédéric Jouault, Yves Lennon, and Frédéric Madiot. MoDisco, a Model-Driven Platform to Support Real Legacy Modernization Use Cases. In *Information Systems Transformation: Architecture-Driven Modernization Case Studies*, pages 365–400. The Morgan Kaufmann/OMG Press, 2010.

[8] Aline Lúcia Baroni and Fernando Brito e Abreu. An OCL-Based Formalization of the MOOSE Metric Suite. In *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Orietend Software Engineering*, 2003.

[9] V. Basili, G. Caldiera, and H. D. Rombach. Goal question metric approach. In J. C. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 528–532. John Wiley & Sons, Inc., 1994.

[10] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[11] Kent Beck and Martin Fowler. Bad Smells in Code. In Martin Fowler, editor, *Refactoring: Improving the Design of Existing Code*, pages 75–88. Addison-Wesley Professional, 1999.

[12] Technische Universität Berlin. AGG: The Attributed Graph Grammar System, 2014. URL http://user.cs.tu-berlin.de/~gragra/agg/.

[13] Technische Universität Berlin. Tiger EMF Transformation Project, 2014. URL http://user.cs.tu-berlin.de/~emftrans/.

[14] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. EMF Model Refactoring based on Graph Transformation Concepts. *ECEASST*, 3, 2006.

[15] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Model Driven Engineering Languages and Systems, MoDELS 2006*, LNCS, pages 425–439. Springer, 2006.

[16] E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *Model Driven Engineering Languages and Systems, MoDELS 2008*, volume 5301 of *LNCS*, pages 53–67. Springer, 2008.

[17] D. Binkley, M. Davis, D. Lawrie, and C. Morrell. To camelcase or under_score. In *IEEE 17th International Conference on Program Comprehension (ICPC)*, pages 158–167. IEEE, 2009.

[18] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, 1988.

[19] Barry W Boehm, John R Brown, Hans Kaspar, and Myron Lipow. *Characteristics of Software Quality*. North-Holland, Amsterdam, 1978.

[20] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring Browser for UML. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *LNCS*, pages 366–377. Springer, 2003.

[21] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[22] Lionel Briand, Prem Devanbu, and Walcelio Melo. An Investigation into Coupling Measures for C++. In *Proceedings of the 19th International Conference on Software Engineering (ICSE)*, pages 412–421, New York, NY, USA, 1997. ACM.

[23] F. Brito e Abreu and W. Melo. Evaluating the Impact of Object-Oriented Design on Software Quality. In *Proceedings of the 3rd International Software Metrics Symposium*, pages 90–99, 1996.

[24] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20 (6):476–493, 1994.

[25] Carnegy Mellon University Software Engineering Institute (CMU/SEI). Capability Maturity Model Integration (CMMI), 2014. URL http://cmmiinstitute.com/.

[26] International Electrotechnical Commission. IEC, 2014. URL http://www.iec.ch/.

[27] Inc. Cunningham & Cunningham. Model Smell, 2014. URL http://c2.com/cgi/wiki?ModelSmell.

[28] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45 (3):621–646, 2006.

[29] Brian Dobing and Jeffrey Parsons. How UML is Used. *Communications of the ACM*, 49(5):109–113, 2006.

[30] Lukasz Dobrzański. UML Model Refactoring - Support for Maintenance of Executable UML Models. Master's thesis, Department of Systems and Software Engineering, Blekinge Institute of Technology, 2005.

[31] Technische Universität Dresden. JaMoPP, 2014. URL http://www.jamopp.org.

[32] Technische Universität Dresden. Refactory, 2014. URL http://www.modelrefactoring.org/index.php/Refactoring.

[33] Alexander Egyed. Instant Consistency Checking for the UML. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 381–390. ACM, 2006.

[34] Alexander Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *Software Engineering, IEEE Transactions on*, 37(2):188–204, 2011.

[35] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[36] Hartmut Ehrig, Claudia Ermel, and Karsten Ehrig. Refactoring of Model Transformations. *Electronic Communications of the EASST*, 18, 2009.

[37] Claudia Ermel, Frank Hermann, Jürgen Gall, and Daniel Binanzer. Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars. *Electronic Communications of the EASST*, 54:1–14, 2012.

[38] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, 2008.

[39] International Organization for Standardization / International Electrotechnical Commission. IEC 62304:2006 – Medical device software – Software life cycle processes, 2014. URL http://www.iso.org/iso/catalogue_detail.htm?csnumber=38421.

[40] International Organization for Standardization / International Electrotechnical Commission. ISO/IEC 25010:2011 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2014. URL http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=35733.

[41] International Organization for Standardization / International Electrotechnical Commission. ISO/IEC 9126-1:2001 – Software engineering – Product quality – Part 1: Quality model, 2014. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749.

[42] International Organization for Standardization. ISO, 2014. URL http://www.iso.org/iso/home.html.

[43] The Eclipse Foundation. Business Intelligence and Reporting Tools (BIRT), 2014. URL http://www.eclipse.org/birt/.

[44] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF), 2014. URL http://www.eclipse.org/modeling/emf/.

[45] The Eclipse Foundation. EMF Compare, 2014. URL http://eclipse.org/emf/compare/.

[46] The Eclipse Foundation. EMF Query, 2014. URL http://www.eclipse.org/projects/project.php?id=modeling.emf.query.

[47] The Eclipse Foundation. EMF Refactor, 2014. URL http://www.eclipse.org/emf-refactor/.

[48] The Eclipse Foundation. EMF Validation, 2014. URL http://www.eclipse.org/projects/project.php?id=modeling.emf.validation.

[49] The Eclipse Foundation. Eclipse Modeling Project, 2014. URL http://www.eclipse.org/modeling/.

[50] The Eclipse Foundation. Eclipse, 2014. URL http://www.eclipse.org/.

[51] The Eclipse Foundation. Epsilon, 2014. URL http://www.eclipse.org/epsilon/.

[52] The Eclipse Foundation. EuGENia, 2014. URL http://www.eclipse.org/epsilon/doc/eugenia/.

[53] The Eclipse Foundation. Graphical Modeling Project (GMP), 2014. URL http://www.eclipse.org/modeling/gmp/.

[54] The Eclipse Foundation. Henshin, 2014. URL http://www.eclipse.org/henshin/.

[55] The Eclipse Foundation. Java Development Tools (JDT), 2014. URL http://www.eclipse.org/jdt/.

[56] The Eclipse Foundation. Java Emitter Templates (JET), 2014. URL http://www.eclipse.org/modeling/m2t/?project=jet.

[57] The Eclipse Foundation. MoDisco, 2014. URL http://www.eclipse.org/MoDisco/.

[58] The Eclipse Foundation. Automating Eclipse PDE Unit Tests using Ant, 2014. URL http://www.eclipse.org/resources/resource.php?id=424.

[59] The Eclipse Foundation. Papyrus, 2014. URL http://www.eclipse.org/papyrus/.

[60] The Eclipse Foundation. Sirius, 2014. URL http://eclipse.org/sirius/.

[61] The Eclipse Foundation. ViaTra, 2014. URL http://www.eclipse.org/viatra2/.

[62] The Eclipse Foundation. Xtext, 2014. URL http://www.eclipse.org/Xtext/.

[63] The Eclipse Foundation. ATL Transformation Language, 2014. URL http://www.eclipse.org/atl/.

[64] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[65] Martin Fowler. Code Smell, 2014. URL http://martinfowler.com/bliki/CodeSmell.html.

[66] David S. Frankel. *Model Driven Architecture Applying MDA*. John Wiley & Sons, 2003.

[67] Leif Frenzel. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. *Eclipse Magazin*, 5, 2006.

[68] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[69] Marcela Genero. *Defining and Validating Metrics for Conceptual Models*. PhD thesis, University of Castilla-La-Mancha, Ciudad Real, Spain, 2002.

[70] Marcela Genero, Mario Piattini, and Coral Calero. Early Measures for UML Class Diagrams. *L'Objet*, 6(4):489–515, 2000.

[71] Marcela Genero, M. Esperanza Manso, Mario Piattini, and Francisco Garcia. Early Metrics for Object Oriented Information Systems. In Dilip Patel, Islam Choudhury, Shushma Patel, and Sergio Cesare, editors, *OOIS 2000*, pages 414–425. Springer, 2001.

[72] Marcela Genero, Mario Piattini, and Coral Calero. A Survey of Metrics for UML Class Diagrams. *Journal of Object Technology*, 4 (9):59–92, 2005.

[73] Marcela Genero, Ana M Fernández-Saez, H James Nelson, Geert Poels, and Mario Piattini. Research Review: A Systematic Literature Review on the Quality of UML Models. *Journal of Database Management (JDM)*, 22(3):46–70, 2011.

[74] J. Ghayathri and E. Mohana Priya. Software Quality Models: A Comparative Study. *International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE)*, 2(1): 42–51, 2013.

[75] SparxSystems Software GmbH. Enterprise Architect, 2014. URL http://www.sparxsystems.de/.

[76] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[77] Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[78] Groove. GRaphs for Object-Oriented VErification, 2014. URL http://groove.cs.utwente.nl/.

[79] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[80] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):321–274, 1987.

[81] R. Harrison, S. Counsell, and R. Nithi. Coupling Metrics for Object-Oriented Design. In *Proceedings of the Fifth International Software Metrics Symposium*, pages 150–157, 1998.

[82] IBM. Rational Software Architect (RSA), 2014. URL http://www-03.ibm.com/software/products/us/en/ratisoftarch.

[83] CollabNet Inc. ArgoUML, 2014. URL http://argouml.tigris.org/.

[84] Ralf Reißing. Towards a Model for Object-Oriented Design Measurement. In *ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering (QAOOSE)*, pages 71–84, 2001.

[85] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[86] Hyoseob Kim and Cornelia Boldyreff. Developing Software Metrics Applicable to UML Models. In *ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, 2002.

[87] Barbara Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[88] Reinhold Achatz Manfred Broy Klaus Pohl, Harald Hönninger. *Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology*. Springer, 2012.

[89] Günter Kniesel and Helge Koch. Static Composition of Refactorings. *Science of Computer Programming*, 52:9–51, 2004.

[90] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture - Foundations and Applications*, LNCS, pages 128–142. Springer, 2006.

[91] Dimitrios S. Kolovos, Richard F. Paige, Fiona Polack, and Louis M. Rose. Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.

[92] Rainer Koschke. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[93] Piotr Kosiuczenko. Redesign of UML Class Diagrams: A Formal Approach. *Software & Systems Modeling*, 8(2):165–183, 2009.

[94] Al Lake and Curtis Cook. Use of Factor Analysis to Develop OOP Software Complexity Metrics. In *Proceedings of the 6th Annual Oregon Workshop on Software Metrics*. Citeseer, 1994.

[95] Christian F. J. Lange. Improving the Quality of UML Models in Practice. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 993–996, New York, NY, USA, 2006. ACM.

[96] Christian F.J. Lange. Empirical investigations in software architecture completeness. Master's thesis, Department of Mathematics and Computing Science, Technical University Eindhoven, The Netherlands, 2003. http://www.win.tue.nl/~clange/papers/Thesis_CLange.pdf.

[97] Christian F.J. Lange. *Assessing and Improving the Quality of Modeling: A series of Empirical Studies about the UML*. PhD thesis, Department of Mathematics and Computing Science, Technical University Eindhoven, The Netherlands, 2007. http://www.langomat.de/research/thesis/thesis.pdf.

[98] Christian F.J. Lange, Bart DuBois, Michel R.V. Chaudron, and Serge Demeyer. An Experimental Investigation of UML Modeling Conventions. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *LNCS*, pages 27–41. Springer, 2006.

[99] Wei Li and Sallie Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23(2):111 – 122, 1993.

[100] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, Inc., 1994.

[101] Francisca Losavio, Ledis Chirinos, Nicole Lévy, and Amar Ramdane-Cherif. Quality Characteristics for Software Architecture. *Journal of Object Technology*, 2(2):133–150, 2003.

[102] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management . *Information and Software Technology*, 51(12):1631–1645, 2009.

[103] No Magic. MagicDraw, 2014. URL http://www.nomagic.com/products/magicdraw.html.

[104] Florian Mantz. Syntactic Quality Assurance Techniques for Software Models. Master's thesis, Department of Mathematics and Computing Science, Philipps-University Marburg, Germany, 2009.

[105] M. Marchesi. OOA Metrics for the Unified Modeling Language. In *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, pages 67–73, 1998.

[106] Slavisa Marković. Composition of UML Described Refactoring Rules. In *OCL and Model Driven Engineering, UML 2004 Conference Workshop*, pages 45–59, 2004.

[107] Slaviša Marković and Thomas Baar. Refactoring OCL Annotated UML Class Diagrams. *Software and Systems Modeling*, 7: 25–47, 2008.

[108] R.C. Martin. OO Design Quality Metrics: An Analysis of Dependencies. In *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, 1994.

[109] Robert Cecil Martin. *Designing Object-Oriented C++ Applications*. Prentice Hall, 1995.

[110] Robert Cecil Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2003.

[111] T.J. McCabe. A Complexity Measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.

[112] Jim A. McCall, Paul K. Richards, and Gene F. Walters. Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality. Technical Report ADA049014, General Electric Co. Sunnyvale California, 1977.

[113] Jacqueline A. McQuillan and James F. Power. On the Application of Software Metrics to UML Models. In Thomas Kühne, editor, *Models in Software Engineering*, volume 4364 of *LNCS*, pages 217–226. Springer, 2007.

[114] Tom Mens, Gabriele Taentzer, and Dirk Müller. Model-Driven Software Refactoring. In J. Rech and C. Bunse, editors, *Model-Driven Software Development: Integrating Quality Assurance*, pages 170–203. IGI Global, Hershey, 2008.

[115] David Miranda, Marcela Genero, and Mario Piattini. Empirical Validation of Metrics for UML Statechart Diagrams. In Olivier Camp, JoaquimB.L. Filipe, Slimane Hammoudi, and Mario Piattini, editors, *International Conference on Enterprise Information Systems (ICEIS)*, pages 101–108. Springer Netherlands, 2005.

[116] Parastoo Mohagheghi, Vegard Dehlen, and Tor Neple. Definitions and Approaches to Model Quality in Model-Based Software Development - A Review of Literature. *Information and Software Technology*, 51(12):1646–1669, 2009.

[117] G.C. Murphy, D. Notkin, and K.J. Sullivan. Software Reflexion Models: Bridging the Gap Between Design and Implementation. *Software Engineering, IEEE Transactions on*, 27(4):364–380, 2001.

[118] Mel O'Cinneide and Paddy Nixon. Composite Refactorings for Java Programs. In *Proc. of Workshop on Formal Techniques for Java Programs at ECOOP 2000*, pages 129–135, 2000.

[119] Bernd Oestereich. *Die UML 2.0 Kurzreferenz für die Praxis*. Oldenbourg Verlag, 2004.

[120] OMG. Meta-Object Facility (MOF), 2014. URL http://www.omg.org/mof/.

[121] OMG. Object Constraint Language (OCL), 2014. URL http://www.omg.org/spec/OCL/.

[122] OMG. Object Management Group, 2014. URL http://www.omg.org/.

[123] OMG. Unified Modeling Language (UML), 2014. URL http://www.uml.org/.

[124] OMG. UML 2.4.1 Superstructure, 2014. URL http://www.omg.org/spec/UML/2.4.1/.

[125] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[126] Oracle. Java, 2014. URL http://www.java.com/.

[127] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kenneth Baclavski and Haim Kilov, editors, *Proc. 10th OOPSLA Workshop on Behavioral Semantics*, pages 187–199. Northeastern University, 2001.

[128] Damien Pollet, Didier Vojtisek, and Jean-Marc Jézéquel. OCL as a Core UML Transformation Language. In *WITUML: Workshop on Integration and Transformation of UML models (held at ECOOP 2002), Malaga, Spain*, 2002.

[129] Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, volume 2863 of *LNCS*, pages 159–174. Springer, 2003.

[130] Alexander Pretschner and Wolfgang Prenninger. Computing refactorings of state machines. *Software and Systems Modeling*, 6 (4):381–399, 2007.

[131] Jörg Rech and Sebastian Weber. Werkzeuge zur Ermittlung von Software-Produktmetriken und Qualitätsdefekten. Technical Report 108.05/D, Fraunhofer Institut Experimentelles Software Engineering, 2005.

[132] Jan Reimann, Mirko Seifert, and Uwe Aßmann. Role-Based Generic Model Refactoring. In *Model Driven Engineering Languages and Systems, 13th International Conference, MoDELS 2010*, LNCS, pages 78–92. Springer, 2010.

[133] Arthur J Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[134] Donald B Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999.

[135] Emmad I. M. Saadeh. *Fine-grained Transformations for Refactoring*. PhD thesis, University of Pretoria, South Africa, 2009.

[136] Markku Sakkinen. Disciplined Inheritance. In *ECOOP*, volume 89, pages 39–56, 1989.

[137] Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML Profiles to Generate Plugins From Visual Model Transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):5 – 16, 2005.

[138] Lars Schneider. Development of a Refactoring Plug-in for the Eclipse Modeling Framework. Master's thesis, Department of Mathematics and Computing Science, Philipps-University Marburg, Germany, 2009.

[139] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2001.

[140] Bran Selic. A Systematic Approach to Domain-Specific Language Design Using UML. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 2–9, 2007.

[141] Philipp Seuring. Design and Implementation of a UML Model Refactoring Tool. Master's thesis, Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam, 2005.

[142] Siemens. Siemens Corporate Technology (CT), 2014. URL `http://www.ct.siemens.de`.

[143] Ian Sommerville. *Software Engineering, 9th Edition*. Addison-Wesley, 2010.

[144] Dave Steinberg, Frank Budinsky, Marcelo Patenostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison Wesley, 2008.

[145] Harald Störrle. On the Impact of Layout Quality to Understanding UML Diagrams. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 135–142, 2011.

[146] Harald Störrle. Towards clone detection in UML domain models. *Software & Systems Modeling*, 12(2):307–329, 2013.

[147] G. Sunyé, D. Pollet, Y. Le Traon, and J. Jézéquel. Refactoring UML models. In *Proc. UML 2001: 4th International Conference on the Unified Modeling Language*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.

[148] Gabriele Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In Manfred Nagl, Andreas Schürr, and Manfred Münch, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 1779 of *LNCS*, pages 481–488. Springer Berlin Heidelberg, 2000.

[149] Gabriele Taentzer. Towards Generating Domain-Specific Model Editors with Complex Editing Commands. In *In Proc. Intern. Workshop Eclipse Technology eXchange(eTX)*, 2006.

[150] Mathupayas Thongmak and Pornsiri Muenchaisri. Using UML Metamodel to Specify Patterns of Design Refactorings. In *Proceedings of the 8th National Computer Science and Engineering Conference (NCSEC)*, 2004.

[151] Bhuvan Unhelkar. *Verification and Validation for Quality of UML 2.0 Models*. Wiley-Interscience, 2005.

[152] Carnegy Mellon University. Software Engineering Institute (CMU/SEI), 2014. URL `http://www.sei.cmu.edu/`.

[153] Marcel van Amstel, Mark van den Brand, and Phu H. Nguyen. Metrics for Model Transformations. In *Ninth Belgian-Netherlands Software Evolution Workshop (BENEVOL)*, 2010.

[154] Heiko van Elsuwe and Doris Schmedding. Metriken für UML-Modelle. *Informatik Forschung und Entwicklung*, 18(1):22–31, 2003.

[155] Norman G. Vinson and Janice A. Singer. A Practical Guide to Ethical Research Involving Humans. In Forrest Shull, Dag Sjøberg, and Janice A. Singer, editors, *Guide to Advanced Empirical Software Engineering*, pages 229–256. Springer, 2008.

[156] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley, 2006.

[157] Yair Wand and Ron Weber. Research Commentary: Information Systems and Conceptual Modeling–A Research Agenda. *Information Systems Research*, 13(4):363–376, 2002.

[158] Manuel Wimmer, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. A Catalogue of Refactorings for Model-to-Model Transformations. *Journal of Object Technology*, 11(2):21–40, 2012.

[159] Niklaus Wirth. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[160] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development*, pages 199–217. Springer, 2005.

[161] Min Zhang, Nathan Baddoo, Paul Wernick, and Tracy Hall. Improving the Precision of Fowler's Definitions of Bad Smells. In *Software Engineering Workshop 2008*, pages 161–166. IEEE, 2008.