Diplomarbeit

Generisches Modellrefactoring für EMFText

bearbeitet von

Jan Reimann

geboren am 16.03.1982 in Potsdam

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Betreuer: Dipl.-Inf. Mirko Seifert
Hochschullehrer: Prof. Dr. rer. nat. habil. Uwe Aßmann

Eingereicht am 12. Juli 2010

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Thema:

Generisches Modellrefactoring für EMFText

Bearbeiter: *Jan Reimann*
Immatrikulationsnummer: *2926972*
Zeitraum: *15.01.2010 – 14.07.2010*

Zielstellung:

In der Modellgetriebenen Softwareentwicklung (Model-driven Software Development – MDSD) bilden Modelle das zentrale Artefakt bei der Erstellung von Software. Im Verlauf eines Entwicklungsprozesses entsteht eine Vielzahl solcher Modelle, welche ständig angepasst, umstrukturiert und reorganisiert werden muss. Dieser Prozess wird als Refactoring bezeichnet und ist im Bereich der Programmiersprachen sowohl theoretisch untersucht, als auch praktisch (in Werkzeugen) umgesetzt. Obwohl die Refaktorisierung von Modellen ähnlich zu der von Programmen ist, existieren kaum entsprechende Werkzeuge. Zudem bieten Modelle durch ihre wohlfundierte Beschreibung in Form von Metamodellen neue Möglichkeiten zur Spezifikation von Refaktorisierungen.

Ziel der Arbeit ist es, zu untersuchen, welche Kategorien von Refactorings auf Modellen existieren und wie sich existierende Refactorings für Programmiersprachen übertragen lassen. Dabei sollen Refactoring-Kataloge analysiert werden, um eine möglichst umfassende Untersuchung zu gewährleisten. Weiterhin gilt es zu prüfen, welche Refactorings auf allen Modellen, bestimmten Teilklassen oder nur auf speziellen Modellen ausführbar sind.

Der praktische Teil der Arbeit liegt in der Implementierung einer Refaktoring-Erweiterung für EMFText, welche die erarbeiteten Konzepte praktisch umsetzt.

Teilziele:

- Analyse und Kategorisierung von Refactorings auf Modellen
- Implementierung der erarbeiteten Konzepte
- Evaluierung anhand von Beispielszenarien

Betreuer: Dipl.-Inf. Mirko Seifert (mirko.seifert@tu-dresden.de)

Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Dresden, den 11.01.2010

Prof.Dr.rer.nat.habil. U. Aßmann
verantwortl. Hochschullehrer

Danksagung

An dieser Stelle wird den Menschen gedankt, ohne die diese Arbeit nicht zustande gekommen wäre. Dazu zählen vor allem meine Eltern, die mich schon immer dazu ermutigt haben, Dinge und Zusammenhänge zu hinterfragen. Nicht zuletzt dadurch konnte ich mich so kritisch mit dem Thema dieser Arbeit auseinandersetzen. Insbesondere ist meiner Mutter in der Rolle der Lektorin zu danken. Zu vergessen sind auch nicht meine Großeltern, die mir das letzte halbe Jahr sehr erleichtert haben. Außerdem gilt meiner Freundin Claudi und meinen Brüdern besonderer Dank, da sie mir immer den nötigen Rückhalt und auch die so wichtige Ablenkung gegeben haben. Vor allem Claudi hatte viel Geduld und konnte mir den einen oder anderen Stress vom Halse halten. Sehr wertvolle Arbeit hat außerdem mein Betreuer Mirko Seifert geleistet, der meine Arbeit durch seine konstruktive Kritik vorangetrieben hat. Dafür möchte ich allen Personen danken.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Seitdem Computer das Licht der Welt erblickt haben, galt es ihnen mitzuteilen, wie sie uns Menschen unterstützen und welche Berechnungen sie uns abnehmen können, da der menschliche Geist fehleranfällig ist und schnell Informationen vergessen kann. Computer hingegen sind Maschinen und führen stets die Schritte aus, die man ihnen aufgetragen hat. Diese Versinnbildlichung des Computers als Freund und Helfer hat durchaus ihre Berechtigung, denn der Computer ist heutzutage aus unserem Alltag nicht mehr wegzudenken. Anfangs kommunizierte der Mensch mit dem Computer auf nahezu gleicher Ebene, indem er ihn maschinennah mit einer Assemblersprache programmierte. Im Laufe der Zeit abstrahierte man jedoch auf immer höhere Stufen, um sich bei der Programmierung dem realen Problem anzunähern. Schnell erkannte man, dass Programmcode wiederverwendbar sein muss, um die Fehleranfälligkeit auf ein Minimum zu reduzieren. Diese Entwicklung brachte zuerst das Paradigma der *Prozeduralen Programmierung* auf und ging bald über in die *Objektorientierung*, aus der schließlich die *Komponentenbasierte Entwicklung* hervorgegangen ist. Die beiden letztgenannten sind heute vorherrschend.

Durch diese Entwicklung ist man nun in der Lage, komplexe Computer-Systeme zu implementieren, die dann in maschinenlesbaren Code übersetzt werden. Aufgrund der steigenden Komplexität solcher Systeme müssen sie vor der Umsetzung entworfen werden, um die Komplexität weitestgehend kontrollieren zu können. Somit ist es schon vor der Implementierung möglich, getreu dem Prinzip *divide et impera*, das zu lösende Problem in Teile zu zerlegen und die Verantwortlichkeiten zu trennen (Separation of Concerns (SoC)). Das Design der Software ist demnach die Grundlage, um ein System zu verstehen und Probleme schon vorher zu identifizieren, statt im Nachhinein mit ihnen leben zu müssen [Pfl98, S. 193].

Da der Software-Entwicklungsprozess jedoch im Allgemeinen nicht linear verläuft, sondern stets neue Funktionalitäten hinzukommen oder existierende an neue Anforderungen angepasst werden müssen, handelt es sich um einen iterativen Prozess, bei dem Design und auch Code in Wechselwirkung oft verändert werden müssen. Hierbei spricht man von der Evolution des Software-Systems, dessen Struktur konstant verändert und weiterentwickelt werden muss, wie es schon Lehman in den Gesetzen der Software-Evolution [Leh96] formuliert hat.

Wie oben schon erwähnt, ist Wiederverwendbarkeit von Code wichtig, um Fehlerquellen zu reduzieren, da dadurch Codewiederholung vermieden wird. Oft werden allerdings solche Problemquellen erst in einem fortgeschrittenen Stadium der Implementierung erkannt, wodurch sich schlechtes Design des Systems im Programmcode widerspiegelt. Um das Design der Software im Nachhinein zu verbessern, kann vorhandener Code umstruk-

turiert werden. Allerdings besteht dabei immer die Gefahr, dass sich Fehler einschleichen oder sich das Verhalten des Systems ungewollt verändert. Opdyke und Johnson haben in [OJ90] den Begriff *Refactoring* eingeführt und beschreiben damit genau die strukturellen Veränderungen vorhandenen Codes, die das Verhalten des Systems nicht verändern. Seither wurde dieses Gebiet der Informatik intensiv erforscht und erkannt, dass Refactoring das Design eines Software-Systems sogar verbessern kann. Dadurch wird außerdem erreicht, dass Software einfacher verstanden und Fehler gefunden werden und auch die Programmierung schneller vonstatten gehen kann [FBB+99, S. 55]. Fowler et al. sagen mit der folgenden kleinen Beobachtung, wie bedeutungsvoll es ist, dass Code auch für den Menschen seine Bedeutung einfach kommunizieren soll:

> *„Any fool can write code that a computer can understand. Good programmers write code that humans can understand.“* [FBB+99, S. 15]

Diese nicht ganz wörtlich zu nehmende Aussage beinhaltet jedoch einen wichtigen Kern. Gutes Design zeichnet sich dadurch aus, dass die Software leicht von Menschen verstanden werden kann, auch wenn es nur der Programmierer ist, der sich nach langer Zeit wieder mit einem eigenen Programm beschäftigen muss.

In den letzten Jahren hat die Modellgetriebene Software-Entwicklung (MDSD) mehr und mehr an Bedeutung gewonnen [SK09]. Dadurch hat die Software-Entwicklung dahingehend einen Wandel erfahren, dass nun nicht mehr nur konventionelle Programmiersprachen benutzt, sondern auch Domain Specific Languages (DSLs) entwickelt werden, um die Arbeit mit Problemen einer bestimmten Domäne zu erleichtern, sowie die Metamodellierung voranzutreiben. Damit ist es möglich, speziell auf einen konkreten Anwendungsbereich zugeschnittene Sprachen und Werkzeuge zu erstellen, die auch von Nicht-Programmierern benutzt werden können. Sie kennen sich in der Domäne aus und ihnen wird meist eine einfache Notation zur Verfügung gestellt, Probleme zu beschreiben, auszutauschen oder zu lösen. An diesem Umstand lässt sich erkennen, dass die maschinelle Verarbeitung von Problemen und ihren Lösungen durch den Computer nun einer sehr viel größeren Anwendergruppe erschlossen wurde, da auch Nicht-Programmierer ihre Ideen dem verarbeitenden Computer kommunizieren können und diese nicht mehr nur dokumentarischen Charakter haben.

Die bereitgestellten Modellierungs-Werkzeuge haben bisher allerdings nur den Charakter von Editoren. Da man aber auch in anderen Anwendungsbereichen als der reinen Programmierung von Werkzeugen profitieren kann, die bei Design und Umsetzung der Problemlösung unterstützen, sollten Anwender statt einfachen Editoren vielmehr Integrated Development Environments (IDEs) zur Verfügung gestellt bekommen, da so die tägliche Arbeit erleichtert und zugleich auch das Modell-Design verbessert werden kann. Aus diesem Grund ist es wichtig, dass Refactorings nicht nur manuell mit DSLs und Modellierungs-Werkzeugen durchgeführt werden können, sondern diese auch (semi-)automatisch für Modelle durchführbar sind. Nur so kann die Fehleranfälligkeit reduziert werden. Man spricht deshalb von *Modell-Refactoring*. Dieses befindet sich momentan noch in einem Stadium der Forschung und Entwicklung [MRG09]. Da viele DSLs und Metamodelle strukturelle Gemeinsamkeiten und Ähnlichkeiten besitzen, ist es wichtig, dass Modell-Refactorings generisch definiert werden können [MTM07], also nicht

für jede DSL von neuem, obwohl aufgrund der ähnlichen Strukturen im Kern dieselben Schritte durchgeführt werden würden. Dieser Herausforderung stellten sich bisher noch nicht viele Forschergruppen – die vorliegende Arbeit wird jedoch einige der aufgekommenen Fragen beantworten und einen Lösungsvorschlag vorstellen.

1.2 Zielstellung

Aus den oben genannten Problemen ergibt sich das Hauptziel dieser Arbeit – eine Möglichkeit zu schaffen, Modell-Refactorings generisch zu definieren, damit sie für beliebige DSLs und Metamodelle wiederverwendet werden können. Die Wiederverwendbarkeit ist von essentieller Bedeutung, da nur so vermieden werden kann, dass gleiche Modell-Refactorings neu entwickelt werden müssen, wenn sie in unterschiedlichen Domänen angewendet werden sollen. Durch die Generalität können außerdem Entwicklungskosten und -zeit reduziert werden, sowie Modell-Refactorings einer breiten Masse an DSLs ebenso wie für Programmiersprachen zur Verfügung gestellt werden.

Um dieses Ziel zu erreichen, gilt es, folgende Teilaufgaben zu erfüllen: Es muss zuerst untersucht werden, inwiefern sich vorhandene Refactorings aus der Welt der Programmiersprachen auf Modelle übertragen lassen. Daraus werden sich Kategorien ergeben, die angeben, welche Refactorings auf alle Modelle anwendbar sind, welche nur auf bestimmte Teilklassen und welche gar auf spezielle Teilmodelle. Dafür ist es nötig, existierende Ansätze zu analysieren und vorhandene Refactoring-Kataloge zu untersuchen, um den aktuellen Stand von Forschung und Technik intensiv mit einbeziehen zu können.

Aus der Analyse ergeben sich Vor- und Nachteile vorhandener Ansätze, welche dazu genutzt werden, Erkenntnisse für diese Arbeit zu gewinnen und neue Ideen zu generieren. Diese bilden die Grundlage für das zu entwickelnde Konzept, welches anschließend im Kontext von EMFText[1] prototypisch umgesetzt wird. Abschließend werden Konzeption und Implementierung an zahlreichen Modell-Refactorings sowie unterschiedlichen DSLs und Metamodellen validiert, um Aussagen über die Güte dieses Ansatzes treffen zu können.

1.3 Überblick über die Arbeit

Kapitel 2 bildet das wissenschaftliche Fundament für die weitere Arbeit. Darin erfolgt die detaillierte Klärung der Begriffe *Refactoring* und *Modell-Refactoring*. Des Weiteren wird nochmals auf *MDSD* und *DSLs* und deren Zusammenhang zur *Metamodellierung* und zum Modellierungs-Stack der *Meta Object Facility (MOF)* eingegangen. Außerdem wird das Framework *EMFText* vorgestellt und es wird erläutert, wie sich die vorliegende Arbeit darin eingliedert.

In der Literatur und im Web zu findende Refactoring-Kataloge werden im Kapitel 3 analysiert. Da diese Kataloge Refactorings in der ursprünglichen Bedeutung, also keine Modell-Refactorings, spezifizieren, wird darauf aufbauend untersucht, inwiefern diese auf Modelle übertragen werden können und welche Kategorien sich daraus ergeben.

[1] `http://www.emftext.org`

Die gewonnenen Erkenntnisse aus den beiden vorangegangenen Kapiteln fließen dann in die Anforderungsanalyse in Kapitel 4 ein. Dort werden funktionale und nichtfunktionale Anforderungen an das zielführende Konzept aufgestellt.

In Kapitel 5 werden dann bestehende Ansätze vorgestellt und hinsichtlich der in 4 aufgestellten Anforderungen bewertet. Die sich daraus ergebenden Vor- und Nachteile bilden die Grundlage für Kapitel 6. Dort wird das Konzept vorgestellt, welches die in Abschnitt 1.1 genannten Probleme Rollen-basiert löst.

Um das entwickelte Konzept zu validieren, wird es zunächst in Kapitel 7 prototypisch umgesetzt und anschließend mit ausgewählten Modell-Refactorings und Metamodellen evaluiert. Dabei werden Designentscheidungen erläutert und Erweiterungspunkte genannt. Abschließend enthält Kapitel 8 die Zusammenfassung sowie eine kritische Bewertung der Arbeit und gibt einen Ausblick auf künftige Forschungsarbeiten.

2 Grundlagen und Begriffsklärung

Um eine solide begriffliche Basis zu schaffen, werden in diesem Kapitel wichtige Konzepte zum Refactoring in ihrer ursprünglichen Bedeutung erklärt. Um die Brücke zum Modell-Refactoring zu schlagen, wird in Abschnitt 2.2 kurz dargestellt, wie sich die Entwicklung auf höhere Abstraktionsstufen vollzogen hat und welche Chancen sich daraus ergeben. Die Vertiefung der textuellen Modell-Repräsentationsform und deren Zusammenhang zu EMFText in Abschnitt 2.4 runden dieses Kapitel ab.

2.1 Code-Refactoring

Wie in Kapitel 1.1 schon erwähnt, haben William Opdyke und Ralph Johnson den Begriff *Refactoring* erstmals in [OJ90] definiert. Zwei Jahre später hat sich Opdyke in seiner Dissertation [Opd92] weiter damit auseinandergesetzt und so den Begriff etabliert. Definiert wird damit die Umstrukturierung vorhandenen Programm-Codes unter Beibehaltung des Verhaltens, um dessen Wiederverwendbarkeit und Design-Qualität zu erhöhen sowie den Code verständlicher, leichter lesbar und weniger fehleranfällig zu gestalten. Opdyke hat in dieser Arbeit insgesamt 29 Refactorings definiert. Als Beispiel sei hier *convert code segment to function* genannt, welches Anweisungen einer vorhandenen Programmfunktion in eine neue verschiebt und einen Aufruf zu dieser an der alten Position einfügt.

Refactorings können dabei weitreichende Auswirkungen haben, da sie durchgeführt werden können, nachdem ein Software-Artefakt schon wiederverwendet wurde. Das bedeutet, dass schließlich auch Clients refaktoriert[1] werden müssen, was eine sehr schwierige Aufgabe darstellen kann, führt man sie manuell durch, da das Ausmaß der Änderungen kaum zu überblicken ist. Aus diesem Grund führte Opdyke an, dass Programmierer durch Refactoring-Werkzeuge unterstützt werden müssen, damit sie die Komplexität bewältigen können. Als wichtig wurde in der Dissertation herausgestellt, dass immer bestimmte Vorbedingungen erfüllt sein müssen, bevor ein Refactoring durchgeführt werden darf. Diese Vorbedingungen sind die Voraussetzungen für die Verhaltenserhaltung.

Später wurde von Fowler et al. ein Buch veröffentlicht [FBB+99], das heute das Standardwerk zum Thema Code-Refactoring darstellt. Die Autoren stellen hier die besondere Bedeutung des Testens heraus, mit dessen Hilfe man das Verhalten vor und nach dem Refactoring vergleichen kann. Somit kann auf unformale Art und Weise überprüft werden, ob das Verhalten durch ein Refactoring erhalten bleibt. Die folgende Definition stammt aus diesem Buch und enthält alle essentiellen Faktoren des Code-Refactorings.

[1] In der vorliegenden Arbeit wird das englische Substantiv „Refactoring“ benutzt, um die Konsistenz zum Thema zu wahren, und weil sich im deutschen Sprachraum auch der englische Terminus eingebürgert hat. Als Verb hingegen wird „refaktorieren“ verwendet.

> *„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. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.“* [FBB+99, S. xvi]

Des Weiteren werden die Umstrukturierungen von Opdyke aufgegriffen und erweitert sowie daraus ein Katalog erstellt, der ähnlich wie der der Entwurfsmuster aus [GHJV04] erscheint. Nach einer kurzen Einführung folgen stets Motivation des Refactorings, die durchzuführenden Schritte und ein Beispiel. Auf diesen Katalog wird im Kapitel 3 näher eingegangen. Trotz des Mangels an Formalität, so sagen die Autoren, sind Refactorings weniger riskant, wenn sie formal beschrieben sind [FBB+99, S. xiii]. Auf dieser Grundlage können Refactoring-Werkzeuge geschaffen werden, wofür die Autoren erste Anforderungen aufgestellt haben. Auf diese wird im Kapitel 7 näher eingegangen, da dort die Konzeption aus Kapitel 6 in einem Werkzeug umgesetzt wird.

Mit diesem Abschnitt wurde eine Einführung in das Code-Refactoring gegeben, welches den Ursprung des Modell-Refactorings darstellt. Um jedoch dahin überzugehen, muss vorher genau geklärt werden, was Modelle konkret sind und wie sie erstellt werden können. Dazu dient das nachfolgende Kapitel.

2.2 Model Driven Software Development (MDSD)

Das Problem der Trennung von Sachverhalten oder Problemlösungen von ihrer Zielplattform-spezifischen Implementierung mit dem Ziel, die Evolution beider separat kontrollieren zu können, ist prinzipiell nicht neu. Dabei wird der Begriff *Plattform* universell für viele Bedeutungen benutzt. Meistens bezieht er sich auf die Umgebung, in der ein Software-System ausgeführt wird. Davon abhängig ist die Art oder Technik der Kompilierung, so dass die ausführende Umgebung die Anweisungen auch verstehen und interpretieren kann. Der im Software-System, dass es auszuführen gilt, abgebildete Sachverhalt muss jedoch in jeder Umgebung derselbe sein. Aus diesem Grund sollte dieser stets unabhängig von der ausführenden Plattform implementiert sein.

In der reinen Objektorientierung wurde dieses Problem der Trennung mit dem *Model-View-Controller*-Prinzip erstmals in Angriff genommen [Bé05]. Hier bezieht sich der Begriff *Plattform* jedoch auf die grafische Umsetzung. Dabei enthält das *Model* die eigentlichen Sachverhalte des Problems und seiner Lösung. Die *View* hingegen stellte hier die Plattform, im Sinne der Art der visuellen Umsetzung, dar. Sie muss die im *Model* definierte Information dem Benutzer präsentieren. Dabei müssen *Model* und *View* lose gekoppelt sein, so dass die Informationen unabhängig von ihrer Darstellung, beispielsweise in einem Balken-Diagramm, verändert und die Art der Präsentation ausgetauscht werden können. Die Synchronisation beider Bestandteile wird mit dem *Controller* gesichert. Das oben genannte Problem konnte mit diesem Prinzip jedoch noch nicht in seiner Ganzheit gelöst werden, da bei der Implementierung der *View* dasselbe Problem in einem anderen Kontext zutage tritt. An dieser Stelle hat der reine objektorientierte Ansatz seine Grenzen erreicht und ein Übergang zu einem komplett neuen Grad an

Abstraktion war unvermeidlich [Bé05]. Die Autorin Kleppe spricht zusätzlich von einer Komplexitätskrise, die zwar kurzfristig durch die Verwendung von Frameworks und wiederkehrenden Mustern unterdrückt, aber langfristig nur durch eine höhere Abstraktionsebene bewältigt werden kann, um genau verstehen zu können, was im Inneren einer Software passiert [Kle08, S. 8].

Des Weiteren wuchs der Druck, Kosten und Zeit bei der Software-Entwicklung zu minimieren, sowie gleichzeitig die Qualität zu erhöhen. Dies umzusetzen vermochten jedoch nur wenige Gruppen fähiger Programmierer [GS03], wodurch allmählich ein Wandel vom objektorientierten Paradigma zur MDSD vollzogen wird, in der das primäre Artefakt im Software-Lebenszyklus nicht mehr der Code, sondern vielmehr das Modell ist. Daraus ergibt sich die Frage, was genau unter einem Modell zu verstehen ist. Die Object Management Group (OMG) setzt folgende, sehr allgemein gehaltene Definition an:

> *„A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.“* [OMG03, S. 2-2]

Der Kern dieser Definition besagt, dass ein Modell eine Abstraktion der Realität (des Systems) darstellt und einen bestimmten Zweck abbildet. Ähnlich, aber sehr viel detaillierter, definiert Jeff Rothenberg ein Modell wie folgt:

> *„A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.“* [Rot89]

An dieser Stelle soll keine neue Definition eingeführt werden, da obige Auszüge ein sehr gutes Grundverständnis von Modellen vermitteln.

Die MDSD begegnet dem Problem der Trennung der Sachverhalte von ihrer Implementierung mit einer anderen, sehr viel allgemeineren Vorgehensweise: Die Elemente, die zur Lösung des Problems führen, werden hier unabhängig von der Plattform modelliert und anschließend Plattform-spezifisch transformiert. So kann klar zwischen beiden Welten getrennt werden. Die Transformation verläuft dabei oft in mehreren Schritten und ist eine Aneinanderreihung mehrerer Transformationen von Modell zu Modell (M2M) bis hin zu Modell zu Code (M2C). Daraus lässt sich erkennen, dass der Software-Entwicklungsprozess nun stark vom Modell als Artefakt gesteuert wird, um so den für den jeweiligen Zweck nötigen Abstraktionsgrad zu erhalten und Modelle so zu trennen, dass die dadurch beschriebenen Sachverhalte möglichst in sich abgeschlossen sind (SoC). Modelle dienen somit nicht mehr nur der Dokumentation, sondern werden im Entwicklungsprozess von Software aktiv als primäre Artefakte benutzt, wodurch die Qualität der Software enorm verbessert werden kann [SV05, S. 16].

Hiermit ist die Basis für eine MDSD gelegt. Der folgende Abschnitt erläutert nun die formalen Grundlagen, die vonnöten sind, um Modelle überhaupt erstellen zu können und zu unterscheiden, zu welcher Modellfamilie ein Modell eigentlich gehört.

2.2.1 Meta Object Facility

Die MOF stellt eine Architektur zur Metamodellierung dar. Durch diese werden die Grundlagen für die Austauschbarkeit von Modellen gelegt. Sie ist seit 1997 ein Standard der OMG und liegt derzeit in Version 2.0 vor [SBPM08, S. 40].

MOF besteht aus einem Kern, dessen Spezifikation in [OMG06a] zu finden ist. Der Ursprung dieser Spezifikation war, dass Datenaustausch zwischen verschiedenen Systemen von der Kompatibilität ihrer Metadaten abhängt. In vielen Systemen werden proprietäre Modelle der Metadaten genutzt, was den Datenaustausch über Systemgrenzen hinaus sehr behindert. MOF bietet dafür ein *Metadata Management Framework*, um die Entwicklung und Interoperabilität von modell- und Metadaten-getriebenen Systemen zu ermöglichen [OMG06a, S. 5]. MOF besteht aus einer Metamodell-Architektur, in welcher Elemente einer konzeptionellen Ebene Elemente der darunter liegenden beschreiben [OMG02] – diese sind demnach konform zu den darüber liegenden. In Abbildung 2.1 ist beispielhaft die MOF-Architektur dargestellt, angelehnt an [Rei09, S. 8].

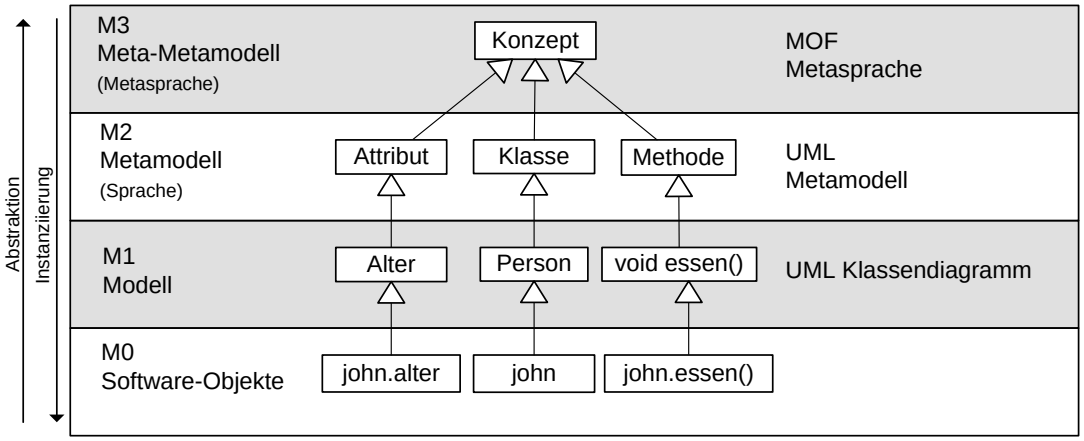


Abbildung 2.1: MOF-Architektur

Die erste Spalte in dieser Abbildung enthält den Namen der MOF-Ebene, die Art des beschriebenen Modells dieser Ebene sowie die Angabe der Sprache, die der Ebene entspricht [Fla02]. In der zweiten Spalte ist die Instanziierungshierarchie an einem Beispiel dargestellt. Die dritte Spalte beinhaltet ein Beispiel der jeweiligen Ebene. Die Abbildung ist von oben nach unten zu lesen. So kann mit einer Metasprache ein Metamodell beschrieben werden. Ein Modell ist Instanz eines Metamodells. Beispielsweise ist ein UML-Klassendiagramm konform zum UML-Metamodell.

Eine für die Praxis relevante Untermenge der MOF bildet Essential MOF (EMOF). Damit wird der Teil der Spezifikation abgedeckt, mit dessen Hilfe einfache Metamodelle mit einfachen Konzepten erstellt werden können [OMG06a, S. 31]. EMOF teilt die Konzepte, die auch im UML-Metamodell für Klassendiagramme enthalten sind. Dadurch ist es möglich, Metamodelle mit UML-Werkzeugen zu erstellen.

In diesen Modellierungs-Stack kann nun jedes Modell eingeordnet werden. Ein Modell der M1-Ebene ist stets konform zu seinem Metamodell aus M2, welches somit die oben schon erwähnte Modellfamilie bildet. Dabei legen M2-Modelle die Regeln fest, die eindeutig definieren, wann ein M1-Modell Instanz eines Metamodells ist und wie diese erzeugt werden können.

Im nachfolgenden Abschnitt wird eine weitere Technik der MDSD erläutert. Außerdem wird darauf eingegangen, dass Modelle, im Gegensatz zu Code, unterschiedliche Repräsentationen haben können. Die sich daraus ergebenden Vorteile sind ausschlaggebend für eine immer weitere Verbreitung.

2.2.2 DSLs und Modell-Repräsentationen

Seitdem sich die maschinelle Verarbeitung von Informationen durch den Computer immer weiter verbreitet, erfährt die Rechentechnik einen Umschwung in Richtung Domänenspezifischer Sprachen (DSLs) [Tah08]. Konventionelle Programmiersprachen (wie beispielsweise Java oder C#) dienen dem Lösen jeder Art von Problemen [@Fow09] und werden deshalb in diesem Zusammenhang *General Purpose Languages* genannt. DSLs werden jedoch entwickelt, um spezifische Klassen von Problemen zu lösen [Tah08]. Das bedeutet, dass sie eine konkrete Domäne fokussieren und Mittel zur Verfügung stellen, um Ideen darin maschinell verarbeitbar zu kommunizieren. Der Ursprung von DSLs liegt darin, dass Personengruppen in einer speziellen Domäne oder einem speziellen Aufgabenbereich Fachwissen besitzen und dafür auch eine spezielle Sprache benutzen [Tah08]. Diese Personen werden *Domänen-Experten* genannt. Dadurch, dass DSLs einen bestimmten Bereich des realen Lebens abbilden, sind sie für Nicht-Programmierer sehr viel leichter zugänglich als Programmiersprachen und ermöglichen somit die Zusammenarbeit zwischen Domänen-Experten und Programmierern [Tah08].

Hier ist es wichtig, zwischen der abstrakten und konkreten Syntax von DSLs zu unterscheiden [@Fow05]. Die abstrakte Syntax drückt aus, welche Bestandteile eine Problembeschreibung oder -lösung in einer speziellen Domäne haben muss, und wie die Zusammenhänge aussehen. Sie definiert demnach ein abstraktes Syntax-Modell [Kle08, S. 41], mit dem zwar die Ausdrucksstärke der DSL festgelegt wird, nicht jedoch ihre Erscheinung. Es gibt verschiedene Möglichkeiten, die abstrakte Syntax einer DSL zu definieren. In der MDSD wird dazu die Metamodellierung verwendet, da damit Graphen definiert werden können und nicht nur Bäume [Kle08, S. 53]. Dies hat den Vorteil, dass Zusammenhänge wie Verweise direkt in Modellen abgebildet werden können und nicht wie bei kontextfreien Grammatiken eine zusätzliche Symboltabelle benötigt wird, um zu unterscheiden, auf welchen Knoten im Baum ein Bezeichner verweist.

Die konkrete Syntax definiert die Notation der DSL, wie sie auf MOF-Ebene M1 instanziiert werden kann. Dies bedeutet insbesondere, dass für das Metamodell einer DSL unterschiedliche Notationen definiert werden können. Mit jeder Notation kann somit eine Modell-Instanz unterschiedliche Repräsentationen besitzen. Diese können verschiedene Ausprägungen annehmen. Zum einen ist es möglich, eine graphische Syntax zu definieren, die ihre Stärken in der Darstellung von Relationen hat, sowie die Möglichkeit bietet, Ausschnitte beliebig nah heran zu zoomen [HJK+09]. Als Beispiel ist in Abbil-

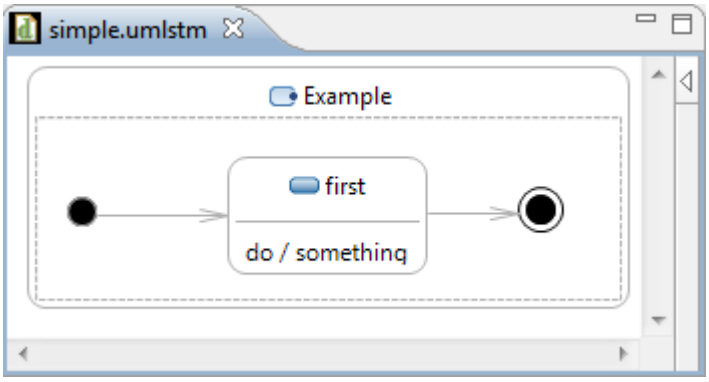
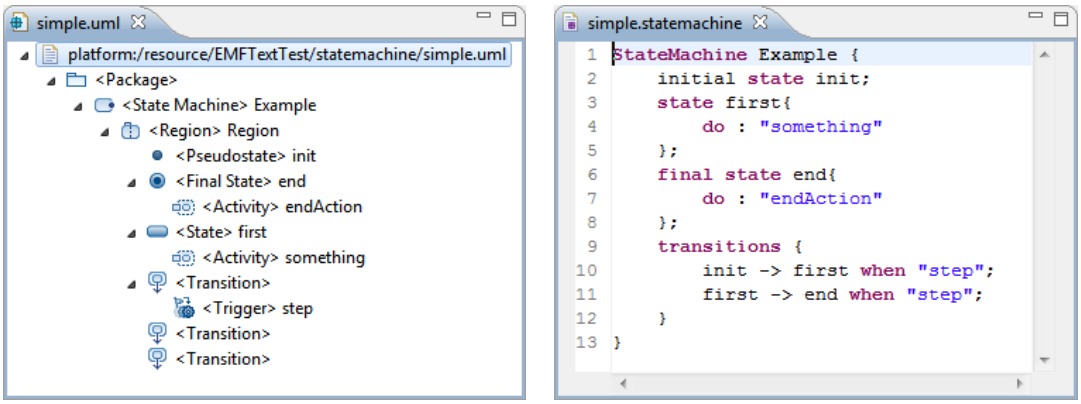


Abbildung 2.2: Zustandsmaschine graphisch



(a) Baum-artig (b) textuell

Abbildung 2.3: Weitere Repräsentationen einer Zustandsmaschine

dung 2.2 die graphische Repräsentation einer sehr einfachen UML-Zustandsmaschine dargestellt. Dort wurden lediglich drei Zustände modelliert: ein initialer Zustand, der Zustand `first`, mit der Aktivität `something`, sowie ein finaler Zustand. Außerdem sind zwei Transitionen als Pfeile abgebildet. Eine andere Repräsentationsform ist die des Baumes. Das Modell aus Abbildung 2.2 wird nun in Abbildung 2.3(a) als Baum präsentiert. Eine dritte Repräsentation ist in Abbildung 2.3(b) textuell dargestellt. Hierbei handelt es sich wieder um dasselbe Modell wie in den anderen beiden Illustrationen.

An diesen drei Beispielen ist zu erkennen, dass DSLs unterschiedliche Repräsentationsformen besitzen können. Das darunter liegende Modell bleibt jedoch immer dasselbe. Für die verarbeitende Maschine spielt die Repräsentation keine Rolle. Denkbar ist auch, dass zu einem Metamodell mehrere verschiedene textuelle Notationen existieren. Diesen Fall könnte man in zweierlei Hinsicht interpretieren. Zum einen kann man von einer Sprache sprechen, die zwei separate konkrete Syntaxen besitzt. Zum anderen kann man auch von zwei verschiedenen Sprachen mit gemeinsamer abstrakter Syntax ausgehen [@Fow05].

Ein wichtiger Punkt an dieser Stelle ist, dass sich die Repräsentationen einer DSL nicht ausschließen müssen. Sie können sehr wohl nebeneinander existieren. Wann welche verwendet wird, hängt lediglich von den Gewohnheiten und Vorlieben des Nutzers ab. Außerdem existieren Sprachen mit hybrider Syntax. Die UML beispielsweise besitzt eine graphische Notation, in die für die Angabe von Attributen und Operationen in Klassen eine textuelle Notation eingebettet wurde [Kle08, S. 5]. Durch eine Repräsentation wird demnach ein Vokabular festgelegt, das sich sehr nah an der Domäne orientiert, wodurch der Anwender der DSL mehr in den Fokus der Entwicklung gerückt wird [GS03].

Diese Entwicklung in der MDSD ist ein wichtiger Beitrag, die Lücke zwischen Programmierern und Fachexperten zu schließen. Des Weiteren wird dadurch zur Kommunikation zwischen DSL-Designer und Anwender beigetragen. Wurden in deren Zusammenarbeit eine oder mehrere Notationen definiert, so können daraus nun die repräsentierenden Editoren generiert werden. Dieser Vorgang ist heutzutage ohne großen Aufwand möglich. Um den generierten Editoren aber mehr den Charakter einer IDE zu verleihen, sind noch einige Hindernisse zu bewältigen. Anwender von DSLs sollten sich ebenso wie Programmierer bemühen, ihre Probleme klar und leicht verständlich zu definieren, was gezwungenermaßen auch hier ein gutes Design auf Modell-Ebene voraussetzt. Mit dem Thema, das Design von Modellen zu verbessern, setzt sich das nachfolgende Kapitel auseinander.

2.3 Code als Modell und Modell-Refactoring

Durch den Einzug der MDSD in die Entwicklungsprozesse von Software muss auch auf diesem Gebiet die Frage des Designs von Modellen diskutiert werden. Da nun Modelle die primären Artefakte der Entwicklung darstellen, sind auch für sie Verständlichkeit und Aussagekraft unerlässlich. Nur dann kann sichergestellt werden, dass sie einfach wiederverwendet werden und andere Personen, nicht nur ihr Designer, damit umgehen können. Diese Behauptungen erinnern stark an die in 1.1 erläuterten Beobachtungen und aufgeführten Notwendigkeiten, stets das Design eines Software-Systems zu verbessern, um Fehlerquellen zu vermeiden und die Wiederverwendbarkeit zu erhöhen. Bei näherer Betrachtung ist jedoch zu erkennen, dass Programmier- und Modellierungssprachen oft gar nicht weit auseinander liegen. Dies ist außerdem gar nicht erwünscht, da der letzte Schritt einer MDSD-Transformationskette nur schwach strukturiert und getypt ist [HJSW10]. Dieser Schritt stellt eine M2C-Generierung von einem Modell in den jeweiligen Programm-Code (beispielsweise Java) dar. Der Grund dafür ist, dass M2C-Engines nicht auf eine bestimmte Sprache beschränkt sind und demnach keinerlei Informationen bezüglich der Syntax der Zielsprache besitzen. Deshalb existieren derartige Transformatoren nur Text-basiert und überprüfen in keiner Weise die Korrektheit des Generats. Dies ist jedoch bei zu kompilierenden und ausführbaren Artefakten essentiell, da sie sonst vom entsprechenden Compiler nicht als solche erkannt werden.

Betrachtet man nun aber Programmiersprachen aus dem Blickwinkel der Modellierung, so ist zu erkennen, dass diese Probleme damit gelöst werden können. Die abstrakte Syntax einer Programmiersprache kann durch ein Metamodell manifestiert werden und

die konkrete Syntax entspricht einer Notation, die exakt so definiert ist wie die Grammatik der Programmiersprache. Ein Beispiel für die Metamodellierung der Spezifikation einer Programmiersprache wurde in dem Projekt JaMoPP[2] gezeigt, in dem für Java ein Metamodell definiert und eine konkrete Syntaxdefinition umgesetzt wurde. Dadurch kann Java nun nicht mehr nur als Programmiersprache, sondern ebenso als Modellierungssprache betrachtet werden [HJSW10], wodurch das oben angesprochene Problem der mangelnden Typung bei M2C-Transformationen überwunden wird. Auf der Basis des Metamodells wird nun der letzte Generierungsschritt nach Java von einer M2M-Transformation abgelöst. So können auch konventionelle Programmiersprachen nahtlos in einen MDSD-Prozess eingegliedert werden und es findet kein Technologie-Bruch statt. Die Betrachtungsweise des Codes als Modell hat den Vorteil, dass nun immer das Metamodell eine Rolle spielt, indem es die Regeln für seine Instanzen vorgibt. Dadurch sind M2M-Transformationen sehr viel strukturierter, da anhand des Metamodells immer die statische Semantik geprüft werden kann. Ist ein vermeintliches Modell nicht Metamodell-konform, kann die Transformation gar nicht erst ausgeführt werden.

Auf dieser Grundlage ist die Wichtigkeit des Refactorings von Modellen leicht einzusehen. Nicht nur Modelle, die mittels DSLs erstellt wurden, sondern auch Modelle von Programmiersprachen müssen ein gutes und durchdachtes Design aufweisen. Um dieses Ziel erreichen zu können, wünscht man sich, dass die Editoren der verschiedenen Metamodelle ausgeprägter die Eigenschaften und Werkzeuge von IDEs besitzen [@Fow05, TDDN00]. Modell-Refactorings sind ein erster Schritt in diese Richtung. In [ZLG05] definieren Zhang et al. Modell-Refactorings als Modell-Transformationen, die vor der Ausführung definierte Vorbedingungen zu erfüllen haben und das Verhalten des Modells nicht verändern. Auch van der Straeten et al. führen eine sehr ähnliche Definition an:

> *„Model refactoring is a transformation used to improve the structure of a model while preserving its behaviour.“* [vdSJM07]

So oder ähnlich wird der Begriff Modell-Refactoring in den meisten Veröffentlichungen verstanden, wobei die Frage des sich nicht ändernden Verhaltens noch nicht zufriedenstellend geklärt werden konnte. So lockern die Autoren von [MRG09] diese Thematik auf, indem sie argumentieren, dass es durchaus eine gewisse Toleranz diesbezüglich geben sollte, solange man vorher genau identifizieren kann, in welcher Art das Verhalten des Modells modifiziert wird. Eine ausführliche Diskussion über die Beibehaltung des Verhaltens wird im Kapitel 6.4 geführt. In dieser Arbeit wird davon ausgegangen, dass das Verhalten, beziehungsweise die Semantik, eines Modells nicht als absolut angesehen werden kann, da verschiedene Anwender unterschiedliche Prioritäten haben und demzufolge auch auf andere Eigenschaften Wert legen. Deshalb wird für die weitere Arbeit folgende Definition eingeführt:

> *Ein Modell-Refactoring ist eine Modell-Transformation, die aus mehreren Einzelschritten bestehen kann, und nur durchgeführt werden darf, wenn Vor-*

[2] `http://www.jamopp.org`

und Nachbedingungen erfüllt sind. Es muss sowohl horizontale als auch vertikale Modell-Beziehungen konsistent halten.

Im ersten Teil der Definition wird aufgrund der oben erwähnten Relativität des Modell-Verhaltens darüber auch keine Aussage getroffen. Der zweite Teil besagt, dass Modelle nicht nur auf gleicher MOF-Ebene in Beziehung stehen können, sondern auch Ebenen-übergreifend. Hiermit ist die nach MOF-Definition gegebene Beziehung zwischen einem Metamodell und seinen Instanzen gemeint. Denkbar ist zum Beispiel, dass in einem Modell-Refactoring der Name einer Metaklasse auf M2-Ebene geändert wird. Dann müssen auch deren Instanzen konsistent gehalten und die Referenzen darauf umbenannt werden.

Auf diesem Weg wurden die Grundlagen des Modell-Refactorings gelegt und eine für diese Arbeit geltende Definition gefunden. Im folgenden Kapitel wird nun eine Einordnung in das EMFText-Framework vorgenommen, welches zur Validierung des zu entwickelnden Konzepts und dessen Implementierung benutzt wird.

2.4 Textuelle Modelle und EMFText

Eine besondere Repräsentationsform von Modellen ist der Text. Mit dieser Notation hat sich das Projekt EMFText auseinandergesetzt. Die Idee der textuellen Repräsentation kam auf, als Metamodelle mit der Zeit weiterentwickelt wurden. Dadurch haben sich vorhandene Teile geändert und es sind neue hinzugekommen. Als Instanzen der Metamodelle, die solch eine Evolution durchlaufen haben, noch in einer XML-Struktur gespeichert wurden, stand man mehreren Problemen gegenüber. Zum einen war die Struktur des Metamodells in der XML-Datei abgebildet, um die einzelnen Elemente den Metaklassen zuordnen zu können. Hat sich die Struktur des Metamodells geändert oder wurden beispielsweise Teile umbenannt, so waren die XML-Dateien nicht mehr Metamodell-konform und mussten manuell angepasst werden. Dies stellte ein Problem dar, da das manuelle Ändern von XML-Dateien sehr aufwändig ist. Außerdem ist es für den Menschen beim Lesen solcher Dateien schwer, die Übersicht zu behalten, da sie oft mit Metainformationen, wie zum Beispiel Identifikatoren, angereichert sind. Deshalb können XML-Dateien nur mit dementsprechenden Editoren effektiv bearbeitet werden. Die EMFText-Entwickler erkannten, dass man diesen Problemen mit einer vereinfachten textuellen Repräsentation begegnen kann. Betrachtet man nochmals Abbildung 2.3(b), so ist nachzuvollziehen, dass dafür eine Abbildung des eigentlichen Metamodells auf eine konkrete Syntax – also eine Grammatik der Sprache – vonnöten ist. In [HJK+09] haben die EMFText-Entwickler gezeigt, dass eine Abbildung von Metamodell auf eine kontextfreie Grammatik einer Sprache möglich ist. Wurde eine textuelle Syntax für das Metamodell festgelegt, kann ein Parser textuelle Repräsentationen dann in die Modelle überführen. Hier ist noch ein weiterer Vorteil gegenüber der XML-Repräsentation zu erkennen: der Parser gibt an, wo Syntaxfehler vorliegen, und weist die Datei nicht einfach zurück. In Abbildung 2.4 ist das Vorgehensmodell von EMFText ausführlich dargestellt. In dieser Abbildung ist links zu erkennen, dass das Eingabe-Metamodell vorher

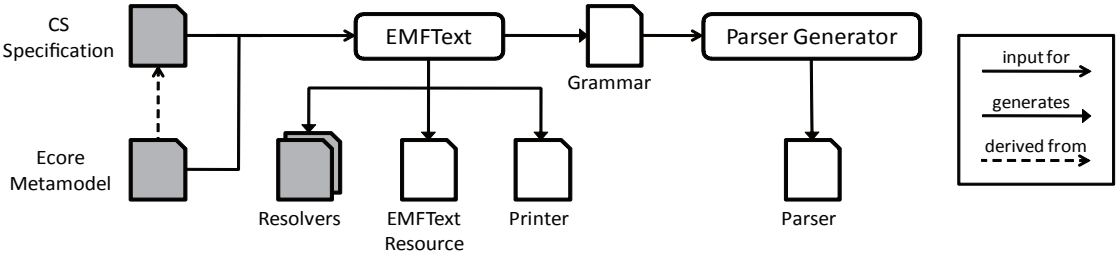


Abbildung 2.4: Vorgehensmodell von EMFText nach [@EMF09]

mit Ecore[3] erstellt werden muss. Ecore ist eine Umsetzung des EMOF-Standards und die Metasprache des Eclipse Modeling Framework (EMF), welches derzeit den Maßstab bei der Modell-Spezifikation in der MDSD darstellt [BEK+06b, MTM07, TMM08]. Für das entsprechende Metamodell wird die konkrete Syntax (*CS Specification*) definiert. Mit dem EMFText-Framework werden dann von der EMF-Implementierung benötigte Artefakte sowie eine Grammatik generiert, die als Eingabe für den Parser-Generator ANTLR[4] dient. Dieser erstellt anhand der definierten Syntax automatisch einen Parser, der textuelle Beschreibungen von Instanzen des spezifischen Metamodells erkennt und den Metaklassen zuordnen kann. Nun ist auch verständlich, warum eine textuelle Repräsentation resistenter gegenüber der Metamodell-Evolution ist. Es muss bei Änderungen nicht mehr jedes einzelne Modell in XML-Repräsentation angepasst werden, sondern nur die konkrete Syntax der textuellen Notation. In seltenen Fällen müssen zusätzlich leichte Änderungen in den textuellen Modellen durchgeführt werden, jedoch nur bei expliziten Änderungen der Syntax.

Weitere wichtige Komponenten, die von EMFText generiert werden, sind die *Resolver*. Diese überwinden die in Kapitel 2.3 angesprochenen Probleme von M2C -Transformationen bezüglich der fehlenden Typung der generierten Texte. Ein Resolver wird in EMFText für jede referenzierende Relation zwischen Metaklassen generiert. Eine Metaklasse ist referenzierend, wenn sie auf eine andere verweist. Instanzen der verwiesenen Metaklasse können dabei nicht von der referenzierenden erzeugt werden. Die Aufgabe eines Resolvers ist es, anhand der übergebenen Textteile das referenzierte Element im Modell zu finden. Somit findet eine Typisierung statt, wodurch die statische Semantik der durch die konkrete Syntax definierten Sprache umgesetzt wird [HJSW10].

Des Weiteren wird von EMFText ein Printer generiert, der die entgegengesetzte Aufgabe des Parsers übernimmt. Der Printer serialisiert ein Modell, welches beispielsweise nur im Arbeitsspeicher vorliegt, wieder anhand der definierten textuellen Syntax. So konnte das EMFText-Team erfolgreich für das schon weiter oben erwähnte Metamodell der Programmiersprache Java eine textuelle Syntax definieren. Damit können Java-Artefakte nun „modelliert“ werden, wodurch die Probleme der traditionellen Code-Generierung eines MDSD-Prozesses überwunden werden, da dieser nun in einer M2M-Transformation erzeugt und lediglich als textuelle Java-Syntax repräsentiert wird.

[3] `http://www.eclipse.org/modeling/emf/?project=emf`
[4] `http://www.antlr.org`

Das EMFText-Framework generiert zu einer konkreten Syntax außerdem einen Text-Editor, wie er schon in Abbildung 2.3(b) zu sehen war. Dieser bietet viele Stärken, wie Code-Vervollständigung und -Navigation (durch die Resolver) sowie Code-Faltung und Syntax-Hervorhebung. Außerdem wird in einer Outline-Sicht stets die Baum-Repräsentation des aktuellen Modells angezeigt. So ein Editor unterstützt den Anwender sehr bei der Erstellung von textuellen Modellen, es fehlt jedoch das Modell-Refactoring, welches ihn zu einer Entwicklungsumgebung aufwerten könnte. An dieser Stelle fügt sich die vorliegende Arbeit ein. Die Implementierung des Konzeptes soll als weitere Funktionalität in einen Editor, der mit EMFText generiert wurde, integrierbar sein.

Diese Arbeit profitiert noch unter einem anderen Gesichtspunkt von EMFText. Auf dessen Webseite wird der sogenannte *Syntax-Zoo*[5] bereitgestellt. Dieser dokumentiert die derzeit mit EMFText erstellten DSLs und erläutert kurz deren Verwendung. Alle dort vorgestellten DSLs sind einschließlich ihrer Metamodelle, einiger Beispiel-Instanzen und der Spezifikationen ihrer konkreten Syntaxen unter der angegebenen Adresse verfügbar. Dieser Syntax-Zoo wird vor allem im Kapitel 7.9, in dem die Validierung vorgenommen wird, als Bezugsquelle diverser DSLs dienen, um den in dieser Arbeit vorgestellten Ansatz auf Allgemeingültigkeit zu prüfen.

Damit wurden die Funktionsweise von EMFText vorgestellt sowie der Rahmen für die Implementierung erläutert. Im nachfolgenden Kapitel werden nun Refactoring-Kataloge analysiert und es wird untersucht, unter welchen Bedingungen Code-Refactorings auf Modell-Refactorings übertragen werden können.

[5] `http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo`

3 Analyse von Refactoring-Katalogen

Um untersuchen zu können, welche Kategorien es für Modell-Refactorings bezüglich der Übertragbarkeit vorhandener Code-Refactorings gibt, werden in diesem Kapitel zunächst die bedeutendsten Kataloge vorgestellt. Es wird ein Überblick über die für die Praxis wichtigsten Refactorings gegeben und diese werden einzeln erläutert. Darauf aufbauend wird untersucht, unter welchen Bedingungen die Refactorings auf Modelle angewendet werden können und es werden dafür Kategorien formuliert.

Ziel dieser Untersuchung ist es, zuerst ein Grundverständnis über vorhandene Code-Refactorings zu vermitteln, sowie zu zeigen, welche Teilschritte durchgeführt werden müssen und welche Probleme sie lösen können. Des Weiteren liegt der Fokus dieser Untersuchung in der Überprüfung der allgemeinen Übertragbarkeit auf beliebige Metamodelle und DSLs. Aus diesem Grund wird analysiert, inwiefern man sich vom Kontext objektorientierter Programmiersprachen lösen kann und ob die Refactorings auch auf Metamodelle übertragbar sind, die in ganz anderen Domänen eingesetzt werden.

3.1 Ein Querschnitt vorhandener Code-Refactorings

Fowler et al. haben in [FBB+99] den wohl bekanntesten Refactoring-Katalog[1] aufgestellt. Die darin beschriebenen Code-Refactorings haben einen aussagekräftigen Namen und werden, wenn angebracht, anhand von UML-Klassendiagrammen vor und nach dem Refactoring erläutert. Danach werden die durchzuführenden Schritte beschrieben, Hinweise gegeben, die es zu beachten gilt, und das Refactoring wird an einem Beispiel vertieft. Außerdem wird dargelegt, wann Refactorings durchgeführt werden sollten. Allgemeingültige Metriken, die schlechtes Design indizieren, gibt es nicht [FBB+99, S. 75], Anzeichen dafür werden aber als *Bad Smells* bezeichnet. Diese sind Indikatoren dafür, wann ein Refactoring anzuwenden ist, müssen aber von jedem Programmierer selbst bewertet und an seine Anforderungen für gutes Design angepasst werden. Als Beispiele seien an dieser Stelle *Duplicated Code*, *Long Method*, *Large Class* und *Long Parameter List* erwähnt. An den Bezeichnungen ist die Bedeutung sehr gut zu erkennen, weshalb sie nur bei einigen Refactorings kurz erläutert werden.

Kerievsky hat einen vom Aufbau her ähnlichen, aber sehr viel detaillierteren Katalog als den zuvor genannten aufgestellt[2]. Dieser widmet sich insbesondere dem Thema der *Design Patterns* aus [GHJV04], welche nicht separat, sondern stets im Kontext von Refactorings betrachtet werden müssen [Ker04, S. 22]. *Design Patterns* können demnach das Ziel eines Refactorings darstellen, da die Präsenz von *Design Patterns* sowohl das De-

[1] Katalog verfügbar unter `http://www.refactoring.com/catalog`
[2] auszugsweise verfügbar unter `http://www.industriallogic.com/xp/refactoring/catalog.html`

sign von Software verbessert, als auch zur Wiederverwendung von Code beitragen kann. Kerievsky warnt aber auch davor, durch zu intensive oder unverhältnismäßige Verwendung von *Design Patterns* genau das Gegenteil zu bewirken und Code unübersichtlich zu machen – er nennt dieses Phänomen *Over-Engineering* [Ker04, S. 22ff].

Die beiden zuvor beschriebenen Werke sind die wichtigsten Nachschlagewerke im Bereich des Code-Refactorings. In beiden Büchern werden die Beispiele jeweils mit der Programmiersprache Java umgesetzt, die aber nur der Veranschaulichung dient, während das Verfahren nicht darauf beschränkt ist. Außerdem sind einige Kataloge online verfügbar, deren Web-Adressen im Folgenden aufgelistet sind.

(a) `http://www.jetbrains.com/idea/features/refactoring.html`

(b) `http://www.jetbrains.com/resharper/features/code_refactoring.html`

(c) `http://www.fortranrefactoring.com.ar/Catalog.html`

In der Reihenfolge der Auflistung wurden dort die Refactorings mit Java, .NET-Sprachen beziehungsweise Fortran veranschaulicht. Bis auf wenige Ausnahmen unter den Fortran-Refactorings sind alle anderen auf andere objektorientierte Programmiersprachen übertragbar. Im Folgenden werden ausgewählte Refactorings aus diesen fünf Katalogen vorgestellt. Sie wurden nach ihrer praktischen Relevanz selektiert.

3.1.1 Extract Method

Dieses Refactoring ist eines der am häufigsten benutzten und wird in [FBB+99, S. 110], (a) und (b) erläutert. In (c) wird ein gleichartiges vorgestellt, welches dort lediglich als *Extract Procedure* bezeichnet wird. Eine leichte Abwandlung dieses Refactorings wird in [Ker04, S. 148ff] unter dem Namen *Compose Method* vorgestellt. Dort wird die Komposition von großen Methoden als Ganzes betrachtet, wobei die einzelnen Schritte auf *Extract Method* zurückzuführen sind. Dieses Refactoring sollte angewendet werden, wenn gleicher Code innerhalb mehrerer Methoden einer Klasse vorkommt (*Duplicated Code*), oder wenn eine Methode aus sehr vielen Anweisungen besteht (*Long Method*). Hierbei werden die gewünschten Anweisungen in eine neue Methode extrahiert, deren Bezeichner dem Zweck entspricht, und ein Methodenaufruf zu dieser wird an der ursprünglichen Position der Anweisungen eingefügt. Die Herausforderung bei *Extract Method* liegt darin, über die Anweisungen der ursprünglichen Methode zu folgern und daraus abzuleiten, welche Variablen in der neuen Methode lokal deklariert werden können und welche als zusätzliche Parameter übergeben werden müssen. Auf diese Weise können große Methoden im Interesse einer besseren Lesbarkeit zerlegt werden oder mehrfach gleich vorkommender Code kann wiederverwendet werden. Das entgegengesetzte Refactoring heißt *Inline Method*, wobei alle Methoden-Aufrufe durch ihren Körper ersetzt werden.

3.1.2 Remove Parameter

Remove Parameter sollte angewendet werden, wenn eine Methode einen oder mehrere Parameter enthält, die in den Anweisungen keine Verwendung haben [FBB+99, S. 277].

Durch komplexe Methoden-Signaturen (*Long Parameter List*) wird der Programm-Code schwer lesbar und die Intention der Methode ist schwer zu erkennen. Dies trifft genau dann zu, wenn für den entsprechenden Methoden-Aufruf (des Clients) erst die zu übergebenden Werte, beziehungsweise Objekte, der Parameter ermittelt werden müssen, obwohl sie eigentlich gar nicht verwendet werden. Dadurch wird Zeit verschwendet, was sich in den Entwicklungskosten niederschlägt. Bei *Remove Parameter* muss überprüft werden, welche Parameter im Inneren einer Methode nicht verwendet werden, um diese anschließend aus der Signatur zu entfernen. Zuletzt müssen die Methoden-Aufrufe angepasst werden.

3.1.3 Rename Method

Dieses Refactoring wird sehr häufig angewendet und ist Bestandteil von [FBB+99, S. 273], (a), (b) und (c). Die drei letztgenannten Kataloge beschränken sich allerdings nicht nur auf Methoden, sondern verallgemeinern *Rename Method* in Richtung der Umbenennung von Elementen im Allgemeinen. Es sollte angewendet werden, wenn der Zweck eines Bezeichner tragenden Elementes nicht ersichtlich ist. In diesem Fall wird das jeweilige Element umbenannt und es müssen alle Verweise darauf aktualisiert werden. Dieses Refactoring ist zwar einfach durchzuführen, kann aber trotzdem sehr komplexe, weitreichende Auswirkungen nach sich ziehen. Wird beispielsweise eine Klassen-Methode umbenannt, die nur von anderen Klassen oder Methoden aufgerufen wird, so sind die anzupassenden Aufrufe verhältnismäßig überschaubar. Befindet sich jedoch diese Methode zum Beispiel in einem Interface und demnach weit oben in der Vererbungshierarchie, so kann die Aufgabe der Aktualisierung sehr komplex sein. Für *Rename Method* ist eine adäquate Werkzeugunterstützung unerlässlich.

3.1.4 Pull Up Member

In den Katalogen (a) und (b) wird das Hochziehen von Elementen einer Klasse in der Vererbungshierarchie *Pull Up Member* genannt. In [FBB+99, S. 320ff] hingegen wird dieses Refactoring nochmals unterteilt in das Hochziehen von Attributen, von Methoden oder von Konstruktor-Körpern. Hier ist *Duplicated Code* ein Anzeichen dafür, *Pull Up Member* durchzuführen. Besitzen beispielsweise mehrere Klassen eine Methode, die jeweils gleich implementiert ist, so kann diese Methode in eine gemeinsame Oberklasse verschoben werden. Dadurch vermeidet man mehrere Fehlerquellen für die gleiche Funktionalität. Ein sehr ähnliches Refactoring, welches angewendet werden kann, wenn noch keine gemeinsame Oberklasse existiert, ist *Extract Superclass*. Hierbei wird vor dem Hochziehen zusätzlich eine Klasse erzeugt, von der die anderen erben. Das Gegenstück von *Pull Up Member* ist *Push Down Member*. Dabei werden Elemente in der Klassenhierarchie nach unten verschoben.

3.1.5 Encapsulate Field

Ähnlich zum *Information Hiding*-Kriterium von Parnas [Par72], das ausdrückt, dass komplizierte Design-Entscheidungen gekapselt werden, also dem Client verborgen blei-

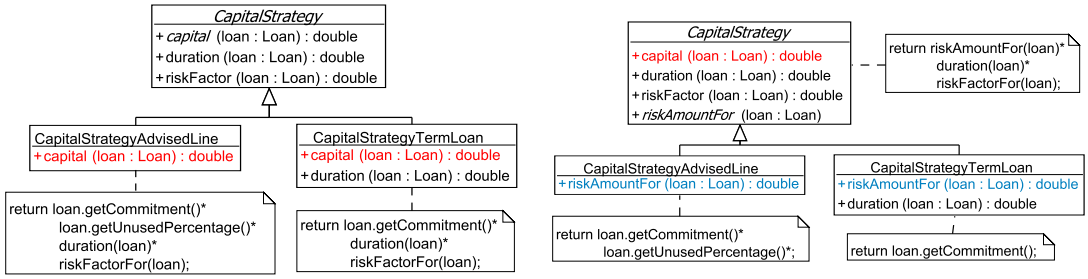
ben sollen, werden mit diesem Refactoring öffentlich zugängliche Daten gekapselt. Mit *Encapsulate Field* wird ein für andere Klassen öffentlich zugängliches Attribut mit sprachspezifischen Techniken gekapselt. Dies bedeutet, dass dafür entweder Zugriffsmethoden für das Lesen und Schreiben definiert werden (wie beispielsweise in Java), oder dass für den Zugriff auf das Attribut eine Property deklariert wird (wie beispielsweise in C#). In beiden Fällen wird die Sichtbarkeit des Attributs auf die besitzende Klasse beschränkt. Nach der Kapselung müssen alle Referenzen auf das vorher sichtbare Attribut dahingehend angepasst werden, dass der Zugriff nun über die Property oder die definierten Methoden vonstatten geht. *Encapsulate Field* wird in [FBB+99, S. 206], (a), (b) und (c) erläutert.

3.1.6 Move Method

Dieses Refactoring ist in den Katalogen [FBB+99, S. 142], (a) und (c) anzutreffen. *Move Method* ist weniger restriktiv als *Pull Up Member* und *Push Down Member*, denn es beschränkt sich nicht nur darauf, Methoden entlang einer Vererbungshierarchie zu verschieben. Ein Anzeichen für die Notwendigkeit dieses Refactorings kann *Large Class* sein. Wird einer Klasse, da sie viele Methoden besitzt, sehr viel Verantwortung zugeschrieben, sollte überlegt werden, ob die Verantwortlichkeiten eventuell getrennt werden können (SoC). Eine andere Indikation kann vorliegen, wenn Klassen zu stark gekoppelt sind. Dies ist der Fall, wenn in der vermeintlich zu verschiebenden Methode andere Objekte öfter referenziert werden als die Instanz der Klasse, zu der sie selbst gehört. Mittels *Move Method* können Methoden dann in eine andere Klasse verschoben werden. Wichtig dabei ist, vorher zu überprüfen, ob die Ziel-Klasse schon eine Methode mit gleichem Namen und gleicher Signatur enthält. In so einem Fall kann vorher *Rename Method* durchgeführt werden. Danach müssen alle Aufrufe dieser Methode aktualisiert werden, oder die ursprüngliche Methode bleibt erhalten und delegiert zu der neuen.

3.1.7 Form Template Method

Ein sehr viel komplexeres Refactoring stellt *Form Template Method* dar. Es wird in [FBB+99, S. 345ff] und [Ker04, S. 232ff] detailliert erläutert. *Template Method* ist ein *Design Pattern* und dient dazu, *Duplicated Code* dann zu vermeiden, wenn zwei Methoden in Unterklassen ähnliche Anweisungen in derselben Reihenfolge ausführen, sich die Anweisungen jedoch unterscheiden [GHJV04, S. 366ff]. Anders ausgedrückt werden damit Variationspunkte eines Algorithmus in den Unterklassen implementiert. Zur Veranschaulichung dieses Zusammenhangs dient Abbildung 3.1(a). Die in dieser Abbildung als Kommentar dargestellten Implementierungen entsprechen den rot markierten Methoden `capital`. Beide unterscheiden sich nur in einer Anweisung – die anderen Anweisungen sind gleich. In dem Design Pattern *Template Method* erhält die abstrakte Oberklasse eine neue abstrakte Methode (hier `riskAmountFor`), in der die abgeleiteten Unterklassen (blau markiert) ihre spezifischen Anweisungen implementieren. Die vorerst abstrakte Methode der Oberklasse stellt nun die Implementierung bereit, indem die gleichen Anweisungen dorthin verschoben werden und die neue abstrakte Methode an



(a) Vorher (b) Nachher

Abbildung 3.1: *Form Template Method* nach [Ker04, S. 233]

entsprechender Stelle aufgerufen wird. Bei *Form Template Method* wird für jeden spezifischen Teil der ursprünglich überschreibenden Methode *Extract Method* durchgeführt – alle dabei erzeugten Methoden erhalten dieselbe Signatur. Dadurch sind nun die Anweisungen in den überschreibenden Methoden jeweils gleich. Diese können nun mittels *Pull Up Member* in die gemeinsame Oberklasse hochgezogen werden. Das Endprodukt des Beispiels ist in Abbildung 3.1(b) dargestellt.

3.1.8 Consolidate Conditional Expression

Dieses Refactoring wird nur in Katalog [FBB+99, S. 240] erläutert. Mit diesem Refactoring wird eine Sequenz von konditionalen Tests, die jeweils verschieden sind, jedoch alle in derselben Aktion resultieren, zu einem konditionalen Ausdruck zusammengeführt. Dieser wird anschließend in eine eigene Methode extrahiert. Daran ist zu erkennen, dass *Consolidate Conditional Expression* einen Spezialfall von *Extract Method* darstellt. Er unterscheidet sich von diesem aber dahingehend, dass er nur angewendet werden kann, wenn es sich bei den zu verschiebenden Anweisungen ausschließlich um konditionale Tests handelt. Mit diesem Refactoring wird erreicht, dass zusammengehörende konditionale Tests an der Stelle ihres Aufrufes besser lesbar sind, da die vorherige Sequenz durch eine einzige Anweisung, welche ihre Intention im Namen transportiert, ersetzt wird.

3.2 Auswertung und Übertragung auf Modell-Refactorings

Die oben beschriebenen Code-Refactorings sind einige der am häufigsten verwendeten. Obwohl sie nur einen Bruchteil der Gesamtmenge darstellen, ist der gegebene Querschnitt trotzdem repräsentativ. Der Katalog in [FBB+99] unterteilt die Refactorings in sieben Kategorien, von denen sechs mit den in diesem Kapitel erläuterten Refactorings abgedeckt wurden. Weitere Refactorings wurden im Text erwähnt. Im Folgenden werden die Kategorien aufgelistet und die vorgestellten Refactorings zugeordnet.

- **Composing Methods:** *Extract Method*
- **Moving Features Between Objects:** *Move Method*

- **Organizing Data:** *Encapsulate Field*
- **Making Method Calls Simpler:** *Rename Method*, *Remove Parameter*
- **Dealing with Generalization:** *Pull Up Member*, *Form Template Method*
- **Simplifying Conditional Expressions:** *Consolidate Conditional Expression*

Die Kategorie **Big Refactorings** wurde hier nicht repräsentiert, da sie nur vier Refactorings enthält und diese aus Refactorings der anderen Kategorien zusammengesetzt sind. Des Weiteren wurden einfach durchzuführende (*Rename Method*), gewöhnliche (*Extract Method*) und komplexe (*Form Template Method*) Refactorings vorgestellt. Somit konnten alle Schwierigkeitsgrade abgedeckt werden. Außerdem sind viele Refactorings eng verwandt, wie beispielsweise *Pull Up Member* und *Extract Superclass* (siehe Abschnitt 3.1.4), oder besitzen Inverse. Ähnlich verhält es sich mit zahlreichen Refactorings, die hier keine Erwähnung fanden. Jene und die hier vorgestellten lassen sich jedoch auf wenige primitive Muster zurückführen, die dann in Kombination verwendet werden. Aus diesem Grund wurden die oben beschriebenen Refactorings ausgewählt. Die anderen dienen der Information, bringen aber keine neuen Erkenntnisse bezüglich weiterer primitiver Muster. Die ermittelten Muster werden im Folgenden benannt und die Übertragbarkeit auf Modell-Refactorings wird diskutiert.

3.2.1 Erzeugung eines Containers

In vielen Code-Refactorings wird in der einen oder anderen Art ein neues Container-Objekt erzeugt. Der Begriff *Container* wird hier bewusst nicht nur auf den offensichtlichen Container *Klasse* beschränkt. Vielmehr werden hier Gemeinsamkeiten verschiedener Elemente betrachtet, die in der objektorientierten Programmierung vorkommen können. Neben der *Klasse* sind demnach auch ein *Interface*, eine *Methode* (für ihre Anweisungen) oder ein *Package* (für Klassen) Container. Die in diesem Kapitel erläuterten Beispiele für die Erzeugung eines Containers sind *Extract Method*, *Extract Superclass*, *Encapsulate Field* oder auch *Form Template Method*. Erweitert man nun den Blickwinkel von objektorientierten Sprachen in Richtung beliebiger Metamodelle, dann zeigt sich, dass auch dort Container-Objekte erstellt werden können. Wichtiger noch: das Erzeugen von Containern kann auch in Metamodellen vollkommen anderer Domänen sehr sinnvoll sein. In dem in Kapitel 2.4 schon erwähnten Syntax-Zoo beispielsweise befindet sich eine DSL zur Spezifikation von Formularen[3]. Ein Formular kann dort aus mehreren Gruppen bestehen, welche wiederum verschiedene Fragen beinhalten. Die Fragen einer Gruppe sollten konsistent dem Zweck der Gruppe entsprechen. Wenn jedoch der Fall eintritt, dass man sich bei der Definition der Fragen immer mehr von der eigentlichen Intention entfernt, so könnte für eine bestimmte Menge von Fragen eine neue Gruppe erzeugt werden, in die die Fragen verschoben werden. Diese Umstrukturierung der Fragen erinnert sehr stark an *Extract Method*, nur dass hier der neue Container keine Methode,

[3] `http://emftext.org/index.php/EMFText_Concrete_Syntax_Forms`

sondern eine neue Gruppe ist. Anhand dieses Beispiels soll gezeigt werden, dass Metamodelle, die Kompositionsbeziehungen enthalten, Kandidaten für die Erzeugung eines neuen Container-Objekts sind. Trotzdem muss vor der Pauschalisierung dieser Aussage gewarnt werden. Es ist nicht von vornherein sinnvoll, das Erzeugen eines Containers für beliebige Metamodelle anzubieten, nur weil es Kompositionsbeziehungen enthält. Dies ist immer abhängig von der Bedeutung der Kompositionsklasse, also der Klasse, von der die Komposition ausgeht. Kommt ihr Zweck dem eines Containers derart nahe, dass Unterelemente in ihr vom Nutzer angelegt werden sollen, und ist die Verwendung der Komposition nicht nur eine geschickte Design-Entscheidung des Metamodell-Designers, die vom Nutzer aber nicht als Container verwendet werden soll, dann ist dieses Metamodell durchaus für so ein Modell-Refactoring in Erwägung zu ziehen.

3.2.2 Erzeugung eines Kind-Elementes

Ein weiteres immer wiederkehrendes Muster ist die Erzeugung eines Kind-Elementes, also die Erstellung einer Kompositionsbeziehung vom Container zu einem neuen Unterelement. In der Objektorientierung sind Kind-Elemente in Form von Klassenattributen, Anweisungen oder Parametern anzutreffen. Ein Beispiel für ein Refactoring dieses Musters ist *Add Parameter* – das Inverse von *Remove Parameter*. Das Kind-Element *Parameter* wird dabei in dem Container *Method* angelegt. Im Syntax-Zoo ist zudem eine DSL verfügbar, mit der man Feature-Modelle definieren kann, um Varianten von Software Product Lines (SPLs) zu konfigurieren. An eine Menge von Features können damit Bedingungen geknüpft werden, mit denen man beispielsweise spezifizieren kann, dass Features sich gegenseitig ausschließen. Um dies auszudrücken, wird ein *Constraint* erstellt, welches die dafür benötigten Informationen besitzt. Ein *Constraint* kann selbst keine Unterelemente aufnehmen, wird aber in dem Container *FeatureModel* erzeugt und ist somit ein Kind-Element. Dieses Muster ist meist Mittel zum Zweck und nur selten einziger Schritt eines Refactorings. Das Beispiel verdeutlicht, dass dieses Muster auch auf Metamodelle übertragbar ist, die keinen objektorientierten Charakter haben.

3.2.3 Verschieben eines Elementes

In vielen Refactorings werden Elemente von ihrer ursprünglichen Position an eine neue Stelle verschoben – so auch bei *Extract Method*, *Pull Up Member*, *Move Method* und *Form Template Method*. Es werden sowohl Container als auch Kind-Elemente verschoben und können zwei verschiedene Ausprägungen haben. Die erste Variante ist ungerichtet und verschiebt ein Element von einem Container in ein anderes Container-Objekt. Die zweite Variante ist gerichtet und verschiebt ein Element entlang eines Pfades. Dies geschieht beispielsweise bei *Pull Up Member* oder *Push Down Member*, wobei nur Klassen entlang der Vererbungshierarchie als Ziel für die Verschiebung in Frage kommen. Auch dieses Muster ist auf andere Metamodelle übertragbar. So zum Beispiel auf eine DSL zur Definition von Graphical User Interfaces (GUIs)[4]. Damit kann unabhängig von der später zu verwendenden Grafikbibliothek die Oberfläche definiert werden. Diese DSL kann

[4] `http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Simple_GUI`

auch Verwendung in einem MDSD-Prozess finden. In einer GUI-Spezifikation können unter anderem Bilder definiert werden. Diese werden entweder in einem *Panel*-Objekt oder einem *Frame*-Objekt definiert. Mit diesem Muster kann nun ein Bild von einem Container-Objekt in ein anderes verschoben werden.

3.2.4 Änderung eines Attributes

Das Ändern eines Attributes geht primär aus *Rename Method* hervor. Darin wird das Attribut *Methodenname* verändert. Allgemein ist die Umbenennung von Elementen wie Klassen, Methoden oder Attributen von entscheidender Bedeutung, da sie einfach durchzuführen ist und mit ihrer Hilfe das Design schnell verbessert werden kann. Wie in Kapitel 1.1 schon erwähnt, muss der Mensch geschriebenen Code lesen und schnell verstehen können [FBB+99, S. 15]. Deshalb ist es von immenser Signifikanz, Elemente so zu benennen, dass sie ihre Bedeutung oder ihr Verhalten schon mit dem Namen transportieren. Speziell bei Methoden kann es dann sein, dass die Implementierung einer aufgerufenen Methode erst gar nicht studiert werden muss, um den Zweck eines Code-Abschnittes zu erfassen, wenn der Methodenname kommunikativ ist. Beispielsweise wird *Rename Method* so motiviert, dass man dessen Durchführung in Betracht ziehen sollte, wenn für eine Menge von Anweisungen ein Kommentar gesetzt wird, der beschreibt, was die Anweisungen tun oder erreichen [FBB+99, S. 273]. Um das Design von objektorientierten Elementen in einem Software-System dadurch zu verbessern, dass sie aussagekräftige Namen erhalten, kann die Umbenennung eines Elementes aus *Rename Method* abgeleitet werden. Jedoch ist dies nur ein Spezialfall der Änderung eines Attributes, da beispielsweise auch die Änderung von Variablenwerten denkbar wäre. Aus diesem Grund wird hier das Muster der Änderung von Attributen eingeführt, mit dem die Werte von Elementen, wie beispielsweise ihr Name, korrigiert werden können.

3.2.5 Entfernen eines Elementes

Das letzte Verwendung findende Muster ist das Entfernen eines Elementes. Auch hier gibt es keinerlei Einschränkung, ob nun ein Container oder ein Kind-Element entfernt wird. Als Beispiel sei *Remove Parameter* (siehe Abschnitt 3.1.2) genannt, wo ein Methoden-Parameter entfernt wird, wenn er ungenutzt bleibt. Der Ausdruck „ungenutzt“ kann in zweierlei Hinsicht interpretiert werden, welche aber vereinzelt differenziert werden müssen. Zum einen ist ein Element ungenutzt, wenn es, wie im Falle von *Remove Parameter*, nicht referenziert wird. Der Parameter existiert, wird aber in der Methoden-Implementierung nicht gelesen oder gesetzt[5]. Zum anderen kann ein Element als ungenutzt angesehen werden, wenn es sich um einen Container handelt und dieser leer ist, demnach keine Unterelemente besitzt. Ein Beispiel dafür wäre eine Klasse ohne Attribute oder Methoden, nur bestehend aus ihrem Namen. In so einem Fall kann die Klasse mit diesem Muster unter Umständen entfernt werden. Dies kann nicht generalisiert werden, ist jedoch ein Indikator für die mögliche Eliminierung. Das Entfernen von leeren

[5] Bei Call-by-Value-Sprachen wie Java hat das Setzen von Methoden-Parametern ohnehin keine Auswirkungen.

Containern oder nicht referenzierten Elementen ist auch auf Modelle übertragbar. Als Beispiel sei hier auf eine DSL zur Beschreibung von Konferenzen[6] verwiesen. Damit können Vorträge definiert werden, die in *Tracks* aufgeteilt sind, aus denen dann später verschiedenste Formen und Formate von Zeitplänen generiert werden können. Wird vom Komitee der Konferenz nun ein Track eines bestimmten Schwerpunktes geplant, haben sich jedoch keine Redner dafür angemeldet, so bleibt der Track leer. In diesem Fall ist ein leerer Track nicht sinnvoll und kann mit diesem Muster entfernt werden.

3.3 Fazit

Im vorangegangenen Kapitel 3.2 wurde gezeigt, dass die in den Katalogen definierten Code-Refactorings auf primitive Muster zurückzuführen sind. Diese sind nicht nur auf objektorientierte Programmiersprachen anwendbar, sondern auch auf DSLs und Metamodelle im Allgemeinen, was mit Beispielen untermauert wurde. Mit der Übertragung der in den Refactorings enthaltenen Muster konnte vorerst nur gezeigt werden, dass diese prinzipiell auch in Modellen Anwendung finden können. Diese Muster decken auch alle in Modellen durchführbare Änderungen ab, da das Erzeugen, das Verschieben, das Ändern von Attributen und das Entfernen von Elementen den kleinsten atomaren Einheiten, bezogen auf mögliche Modelländerungen, entsprechen. Ähnlich wurden diese Einheiten auch schon von Opdyke definiert [Opd92, S. 32ff]. Eine kleine Erweiterung für Modelle muss jedoch noch formuliert werden. Wie in Kapitel 2.2.2 schon erläutert, stellen Modelle im Gegensatz zu Programm-Code Graphen dar. Das bedeutet, dass ein Modell-Element ein anderes referenzieren kann. Auch eine Referenz kann demzufolge geändert werden, weshalb dieser Anwendungsfall und das Ändern von Attributen verallgemeinert werden. Deshalb wird dieses primitve Muster nun *Ändern vorhandener Elemente* genannt.

Was dennoch offen bleibt, ist die Frage, ob die definierten Refactorings als Ganzes übertragen werden können. Für diese Betrachtungsweise bietet sich eine Kategorisierung an.

Unter die erste Kategorie fallen die Code-Refactorings, die intensiv auf den Spezifika der Programmiersprache aufbauen, für die sie umgesetzt wurden. Dazu zählt beispielsweise das Refactoring *Replace Error Code with Exception* [FBB+99, S. 310], welches bisher noch keine Erwähnung fand. Bei diesem Refactoring ist der Ausgangspunkt eine Methode, in der verschiedene Fehlersituationen meistens mit einem ganzzahligen Wert kodiert werden, welcher von der Methode zurückgegeben wird. In komplexen Software-Systemen ist aber oft die Rückgabe-Stelle einer Methode nicht der Teil, wo der Fehler behandelt und ausgewertet werden kann, da dies vom Kontext abhängt. Für diese Art der Fehlerbehandlung besitzen viele Programmiersprachen den Mechanismus der *Exceptions*. Damit können Fehler in der Kette des Methodenaufrufs nach oben propagiert werden, wodurch an der dem Kontext entsprechenden Stelle die Ausnahme behandelt werden kann. Für diesen Zweck wird mit *Replace Error Code with Exception*, wie der Name schon vermuten lässt, der vorher zurückgegebene Fehler in Form einer natürlichen Zahl durch das Initiieren einer Exception ersetzt. Diese Art der Übertragung von Feh-

[6] `http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Conference`

lern wird von vielen, jedoch nicht von allen Programmiersprachen unterstützt. In PHP beispielsweise gab es vor der Version 5 noch keine derartige Ausnahmebehandlung[7]. Deshalb sind Code-Refactorings, die auf spezifische Strukturen oder Mechanismen der umzusetzenden Programmiersprache aufsetzen, nur bedingt übertragbar. Man kann hier auch nicht von Übertragbarkeit auf objektorientierte DSLs oder Metamodelle ausgehen, da nicht garantiert ist, dass die genannten Techniken in allen Sprachen existieren (wie am PHP-Beispiel gezeigt). Da derartige Refactorings nur auf DSLs übertragbar sind, die genau dieselben Konzepte besitzen wie die Programmiersprache, für die sie entworfen wurden, wird diese Kategorie **spezifisch** genannt.

Die zweite Kategorie umfasst Code-Refactorings, die zwar auf objektorientierte Strukturen aufbauen, jedoch nicht von Besonderheiten einer Programmiersprache abhängen. Dies bedeutet, dass Refactorings dieser Kategorie allgemein auf objektorientierte DSLs übertragbar sind. Objektorientierte Metamodelle beinhalten demnach Konzepte für Klassen, Methoden, Attribute und Vererbung. Als Beispiel sei hier *Form Template Method* genannt. Die Methoden-Implementierungen dieses Refactorings wurden in Abbildung 3.1 beispielhaft mit Java dargestellt. Es ist jedoch in beliebigen objektorientierten Sprachen anwendbar, da diese sich durch die Konzepte der Klasse, Methode und Vererbung auszeichnen. Bei *Form Template Method* wird außerdem kein sprachspezifisches Konzept benutzt, weshalb es dieser Kategorie zugeordnet ist, da es weniger restriktiv als spezifische Refactorings ist. Da es sich hierbei um generell objektorientierte Refactorings handelt, wird auch diese Kategorie **objektorientiert** genannt.

Die dritte Kategorie umfasst Code-Refactorings, die nicht spezifisch- und nicht generell-objektorientiert sind, jedoch trotzdem von allgemeinen Konzepten der Programmierung abhängen. Das heißt, dass Refactorings dieser Kategorie auf Konzepten arbeiten, die neben der objektorientierten Programmierung auch in allen anderen Programmierparadigmen und auch -sprachen benutzt werden beziehungsweise vorkommen. Unter diese Kategorie fallen beispielsweise Refactorings, die konditionale Ausdrücke als Ein- oder Ausgabe besitzen. Das Refactoring *Consolidate Conditional Expression* ist ein Beispiel dafür. Dabei wird eine Sequenz von konditionalen Tests mit gleichen Rückgabewerten in einen einzigen konditionalen Ausdruck zusammengeführt und dieser wird dann extrahiert [FBB+99, S. 240]. Dieses Refactoring ist ein Spezialfall von *Extract Method*, da auch hier eine Methode extrahiert wird. Es unterscheidet sich von diesem aber dahingehend, dass es nur angewendet werden kann, wenn es sich bei den zu verschiebenden Anweisungen ausschließlich um konditionale Tests handelt. Konditionen (Ausdrücke, die auf Wahrheitswerte prüfen) sind in jeder Programmiersprache formulierbar, weshalb solche Refactorings in dieser Kategorie liegen. Hier werden Refactorings zusammengefasst, die von allgemeinen Konzepten der Programmierung abhängen und nicht auf objektorientierte Eigenschaften zu beschränken sind. Als Konsequenz dessen wird diese Kategorie **generell-programmatisch** genannt.

Die letzte hier eingeführte Kategorie enthält alle Code-Refactorings, die prinzipiell auf Modelle übertragen werden können, ohne deren Metamodelle auf objektorientierte oder programmatische Konzepte einschränken zu müssen. Dabei gilt es die Frage zu klären,

[7] `http://php.net/manual/de/language.exceptions.php`

woran zu erkennen ist, ob Refactorings allgemein übertragen werden können. In den drei zuvor definierten Kategorien wurde gezeigt, dass stets Strukturen der Programmierung Gegenstand der Umstrukturierung waren. Oberflächlich betrachtet erscheinen alle anderen Refactorings vorerst auch generell-objektorientiert. Untersucht man sie jedoch im Detail, so fällt auf, dass sie zwar in ihrer ursprünglichen Definition nur in objektorientierten Programmiersprachen angewendet wurden, diese Definition aber abstrahierbar ist. Verlässt man beispielsweise die Sichtweise, eine Klasse zu extrahieren, und begibt sich in die Position, stattdessen einen Container zu abstrahieren, leuchtet ein, dass durch Verallgemeinerung der involvierten Elemente eines Refactorings durchaus Aussagen zur Übertragbarkeit auf beliebige DSLs gemacht werden können. Ein Beispiel dafür wurde im Abschnitt 3.2.1 gegeben. Dort ging es um den Vergleich zwischen *Extract Method* und dem Extrahieren einer neuen Formular-Gruppe und dem Verschieben von Fragen dorthin. Dieser Vergleich war nur möglich, da die beteiligten Elemente von *Extract Method* auf einer abstrakten Ebene beschrieben werden konnten. Dieser Punkt ist auch ausschlaggebend für die Übertragbarkeit. Kann ein in den vorgestellten Katalogen definiertes Code-Refactoring abstrakt und ohne Verwendung von Termini der Programmierung beschrieben werden, so ist es prinzipiell auf beliebige DSLs übertragbar. Dies ist selbstverständlich trotzdem nur dann möglich, wenn das entsprechende Metamodell ähnliche Strukturen aufweist, wie sie von der ursprünglichen Refactoring-Definition vorgegeben sind. Dadurch entstehen zwei verschiedene Arten von Übertragbarkeit. Zum einen die *abstrakte Übertragbarkeit*, die nur angibt, dass ein Refactoring prinzipiell, aufgrund des Potenzials es abstrakt zu beschreiben, auch auf Modelle angewendet werden kann. Zum anderen die *strukturelle Übertragbarkeit*, die aussagt, dass das Metamodell, auf dessen Instanzen ein Refactoring angewendet werden soll, auch die strukturellen Voraussetzungen dafür mitbringt. Konkret bedeutet dies, dass beispielsweise *Extract Method* nur auf ein Metamodell übertragen werden kann, wenn dieses mindestens eine Metaklasse enthält, die mindestens zu einer anderen Metaklasse eine Kompositionsbeziehung unterhält. Nur dann kann die benötigte Container – Kind-Element – Beziehung aufgebaut werden. Erst wenn ein Refactoring sowohl abstrakt als auch strukturell übertragbar ist, kann es auf ein konkretes Metamodell angewendet werden. Refactorings dieser Kategorie besitzen das größte Potenzial über Metamodell-Grenzen hinweg wiederverwendet zu werden. Deshalb wird diese Kategorie **generisch** genannt.

Zusammenfassend kann gesagt werden, dass vorhandene Kataloge bezüglich ihrer Übertragbarkeit analysiert und, darauf aufbauend, vier Kategorien formuliert werden konnten. Diese sind aufsteigend nach ihrem Abstraktionsgrad geordnet, begonnen bei spezifisch, über objektorientiert und generell-programmatisch, bis hin zu generisch. Je mehr Sprachspezifika in ein Refactoring involviert sind, desto tiefer muss es in die Kategorien eingegliedert werden. Damit konnten Aussagen über die Übertragbarkeit gemacht werden. Ein weniger übertragbares Refactoring kann nach dieser Kategorisierung trotzdem auf Modelle angewendet werden, allerdings nur, wenn es zumindest strukturell übertragbar ist und die sprachspezifischen Konzepte enthält. Im nachfolgenden Kapitel wird nun die Anforderungsanalyse für das generische Definieren von Modell-Refactorings durchgeführt. Erkenntnisse aus diesem Kapitel fließen dahingehend ein, dass alle hier erläuterten Kategorien von Refactorings unterstützt werden müssen.

4 Anforderungsanalyse für generisches Modell-Refactoring

In den Kapiteln 1 und 2 wurden schon wichtige Punkte genannt, die im zu entwickelnden Konzept nicht fehlen dürfen. Außerdem haben sich durch die in Kapitel 3 vorgenommene Kategorisierung weitere Anforderungen ergeben. Diese werden in diesem Kapitel klar definiert und ausformuliert. Zunächst geht Abschnitt 4.1 auf funktionale Anforderungen an das Konzept des generischen Modell-Refactorings ein, während in Abschnitt 4.2 nichtfunktionale Anforderungen aufgestellt werden. Dieses Kapitel bildet die Grundlage für die Analyse existierender Ansätze in Kapitel 5.

4.1 Konzeptionelle Anforderungen

Fowler et al. haben in ihrem Buch [FBB+99] schon einige Anforderungen, die ein Refactoring-Werkzeug erfüllen muss, aufgestellt. In [MVG06] und [MTM07] wurden diese erweitert und an die neuen Bedürfnisse des Modell-Refactorings angepasst. Die dort erläuterten Anforderungen dienen hier als Grundlage und werden im Folgenden aufgegriffen und verfeinert.

1) Generizität und Wiederverwendbarkeit: Dies stellt die wichtigste Anforderung an das zu entwickelnde Konzept dar. Um sicherzustellen, dass von der Bedeutung her gleiche Modell-Refactorings nicht für jedes Metamodell erneut definiert werden müssen, ist es unerlässlich, Modell-Refactorings generisch zu definieren [MTM07], [TMM08], [MMBJ09]. Auch die Autoren von [RL04, S. 29] führen an, dass hier die Grenzen in der Werkzeugunterstützung liegen. Wie schon in Kapitel 1.2 erwähnt, können nur durch die generische Definierbarkeit Entwicklungszeit und -kosten gesenkt werden, da Modell-Refactorings auf diese Weise wiederverwendet werden können. Dies bedeutet im Detail, dass sie unabhängig von dem Metamodell, auf dessen Instanzen es angewendet werden soll, definiert werden müssen. Demnach muss eine Möglichkeit geschaffen werden, die einzelnen durchzuführenden Schritte eines Modell-Refactorings anzugeben und diese dann im Kontext des jeweiligen Metamodells durchzuführen.

2) Unterstützung aller Kategorien: Um möglichst viele Modell-Refactorings spezifizieren zu können, muss das zu entwickelnde Konzept alle in Kapitel 3.3 erläuterten Kategorien unterstützen. Als wichtiger Punkt für diese Anforderung zählt, dass die Spezifizierbarkeit über die objektorientierte und die generell-programmatische Kategorie hinausgeht und vor allem auch generische Modell-Refactorings zulässt.

Nur so wird sichergestellt, dass Modell-Refactorings auch solchen DSLs zur Verfügung gestellt werden können, die nicht den Charakter einer Programmiersprache aufweisen.

3) Unterstützung von Sprachspezifika: Trotz der zuvor geforderten Generizität, die die Grundlage für Wiederverwendbarkeit bildet, ist es möglich, dass diese nicht ausreicht, um alle Belange einer DSL bezüglich eines Modell-Refactorings zu erfüllen. Als Beispiel sei hier auf *Extract Method* verwiesen. Dort wird nach der Extraktion der neuen Methode ein Aufruf zu dieser an der ursprünglichen Position der Anweisungen gesetzt.

Eine Möglichkeit der Übertragung bietet das Metamodell der UML, speziell der Teil, mit dem Zustandsmaschinen definiert werden können. *Extract Method* ließe sich hier als *Extract Composite State* anwenden: mehrere Zustände werden in einen neuen kompositen Zustand verschoben. Der Methodenaufruf wird hier in Gestalt einer Transition zum neuen Zustand umgesetzt. Im Kern müssen dabei dieselben Schritte wie bei *Extract Method* vollzogen werden. Eine Ausnahme bildet hier lediglich die Tatsache, dass danach, neben der neuen eingehenden Transition, auch eine ausgehende Transition gesetzt werden muss, um die vor dem Modell-Refactoring vorhandenen Übergänge nicht zu zerstören. Dieses Beispiel zeigt, dass es möglich sein muss, neben der Verwendung generischer Modell-Refactorings, diese auch sprachspezifisch anzupassen. Demnach ist die Integrierbarkeit von Sprachspezifika bezüglich eines Modell-Refactorings in dessen generisch definierte Schritte erforderlich.

4) Konsistenz von Modell-Beziehungen: Wie in Kapitel 2.3 definiert, zeichnet sich ein Modell-Refactoring insbesondere dadurch aus, dass sowohl horizontale als auch vertikale Modell-Beziehungen konsistent zu halten sind. Bekannt ist diese Vorgehensweise von Code-Refactorings einschlägiger IDEs. Wird damit beispielsweise der Name einer Klasse oder Methode geändert, werden auch alle Verweise aktualisiert. Würde dies nicht geschehen, läge ein inkonsistenter Zustand der Artefakte vor. Deshalb ist es auch beim Modell-Refactoring wichtig, alle Referenzen auf die refaktorierten Modelle anzupassen. Die Aktualisierung muss auch dann durchgeführt werden, wenn das zu refaktorierende Modell ein Metamodell ist. So müssen beispielsweise bei der Änderung des Namens einer Metaklasse auch die Referenzen zu dieser in allen Instanzen angepasst werden. Dabei handelt es sich um vertikale Konsistenz, da das Metamodell in dem MOF-Stack immer eine Ebene über seinen Instanzen liegt.

5) Beibehaltung der Semantik: In Kapitel 2.3 wurde schon auf die Semantik von Modell-Refactorings eingegangen. Dort wird argumentiert, dass diese nicht als absolut anzunehmen ist, da in verschiedenen Kontexten unterschiedliche Modell-Eigenschaften eine Rolle spielen können. Dazu sei nochmals auf *Extract Method* verwiesen. Durch den zusätzlichen Aufruf einer Methode kann es bei der Ausführung passieren, dass der Ablauf um ein paar Millisekunden verzögert wird. Unter Normalbedingungen spielt dies meist keine Rolle; in Systemen mit kritischen

Realzeit-Anforderungen kann diese Verzögerung jedoch dazu führen, dass das gewünschte Verhalten eines Modells nicht mehr zufriedenstellend beibehalten wird. Demnach muss im zu entwickelnden Konzept eine Möglichkeit geschaffen werden, das gewünschte Verhalten zu beschreiben und dies zu überprüfen.

6) Angabe von Vor- und Nachbedingungen: Wie Opdyke schon in [Opd92] für Code-Refactorings erläutert hat, ist es auch für Modell-Refactorings wichtig, Vorbedingungen anzugeben. Diese können den Zustand definieren, den ein Modell innehaben muss, bevor es refaktoriert werden kann. Sind die Vorbedingungen nicht erfüllt, so darf das Modell-Refactoring nicht durchgeführt werden. Die Angabe von Nachbedingungen steht im Zusammenhang mit der Anforderung 5. Damit kann die Bedeutung, die auch danach noch erfüllt sein muss, im Kontext eines Modell-Refactorings formal angegeben werden. Ein Modell-Refactoring darf demnach nur angewendet werden, wenn sowohl die Vor- als auch die Nachbedingungen zutreffen.

7) Atomarität: Ein Modell-Refactoring muss immer in seiner Gesamtheit angewendet werden. Treten während der Durchführung Fehler auf oder können Teilschritte nicht durchgeführt werden, so besteht die Gefahr, dass das Modell, auf das es angewendet wurde, in einen invaliden Zustand versetzt wird. In diesem Fall muss das Modell-Refactoring zurückgefahren werden. Es kann demnach entweder nur komplett oder gar nicht durchgeführt werden.

8) Umkehrbarkeit: Die Umkehrbarkeit steht in direktem Zusammenhang mit Anforderung 7. Hier liegt der Fokus jedoch nicht auf möglichen auftretenden Fehlerfällen, sondern darauf, dass der Anwender eines Modell-Refactorings eine Entscheidung widerrufen können muss. Es muss ihm ermöglicht werden, ein Modell-Refactoring umzukehren, wenn er beispielsweise versehentlich ein falsches aufgerufen hat. Dann muss die Entscheidung korrigiert werden und das Modell in seinen Ausgangszustand gebracht werden.

9) Empfehlung von Modell-Refactorings: Um den Anwender effektiv unterstützen zu können, darf dieser bei der Auswahl von definierten Modell-Refactorings nicht allein gelassen werden. Er muss bei der Auswahl eines möglichen Modell-Refactorings unterstützt werden, indem in Abhängigkeit vom aktuellen Kontext Möglichkeiten der Umstrukturierung empfohlen werden. Dies ist besonders dann wichtig, wenn eine Vielzahl von Modell-Refactorings definiert wurde.

4.2 Nichtfunktionale Anforderungen

Die oben definierten Anforderungen sind vorwiegend funktionaler Natur und müssen von dem Konzept, beziehungsweise der Implementierung selbst, erfüllt werden. Nachfolgend werden weitere nichtfunktionale Anforderungen aus [RR06, S. 476ff] aufgegriffen und an diese Arbeit angepasst.

10) Interoperabilität: Die geforderte Generizität aus Anforderung 1 muss nicht nur im Konzept selbst, sondern auch in der Umsetzung erfüllt werden. Dies bedeutet zum

einen, dass keine Einschränkungen bezüglich der Metamodelle, für die Modell-Refactorings zur Verfügung gestellt werden sollen, gemacht werden dürfen, sofern diese MOF-konform sind. Zum anderen muss die Implementierung von der Repräsentation der Modelle unabhängig sein. Demnach muss es möglich sein, Modell-Refactorings unabhängig von der repräsentierenden Technologie, das heißt in beliebigen Editoren, anzuwenden.

11) Erweiterbarkeit: Um sicherzustellen, dass keine fixe Menge an Modell-Refactorings existiert, muss es eine einfache Möglichkeit geben, neue Umstrukturierungen zu definieren und diese auch dem Benutzer zur Verfügung zu stellen. Des Weiteren sollten grundlegende Erweiterungspunkte gesetzt werden, an denen neue Konzepte angebunden und neue Semantiken abgebildet werden können.

12) Leichte Benutzbarkeit und Integrationsfähigkeit: Zur Erhöhung der Akzeptanz seitens des Benutzers ist es unerlässlich, dass Modell-Refactorings leicht zu verwenden sind. Dafür ist es von essentieller Bedeutung, dass sie sich nahtlos in die gewohnte IDE einfügen und ohne großen Aufwand durchgeführt werden können. Eine Vorgabe bilden dabei die in jeder IDE schon vorhandenen Code-Refactorings, von denen sich die Anwendung der hier entwickelten Modell-Refactorings kaum unterscheiden sollte.

13) Hohe Durchführungsgeschwindigkeit: Die Durchführungsgeschwindigkeit steht im Zusammenhang mit der zuvor genannten Anforderung 12. Damit der Benutzer auch einen Vorteil aus der Automatisierung ziehen kann, müssen Modell-Refactorings angemessen schnell durchgeführt werden. Nur so kann der Nutzer effektiv bei seiner Arbeit unterstützt werden.

14) Unterstützung von Standards: Da Standards einen langen Reifeprozess durchlaufen haben, genießen sie in der Gemeinschaft der Anwender ein hohes Maß an Akzeptanz und Zuspruch. Aus diesem Grund muss garantiert sein, dass das Konzept selbst auf Standards beruht und diese weitestgehend unterstützt, um eine qualitativ hochwertige Umsetzung zu sichern.

Die in diesem Kapitel erläuterten Anforderungen stellen im nachfolgenden Kapitel die Grundlage für die Untersuchung existierender Ansätze dar. Sie bilden den Rahmen der Analyse und Bewertung sowie die in Kapitel 6 und 7 umzusetzenden Kriterien.

5 Analyse existierender Modell-Refactoring-Ansätze

Bis vor einigen Jahren wurde davon ausgegangen, dass hauptsächlich UML-Modelle für Modell-Refactorings geeignet sind [MTM07]. Dass dem nicht so ist, konnte im Kapitel 3 gezeigt werden. Nun gilt es zu analysieren, ob es existierende Ansätze aus der Forschung und Technik vermögen, die dort definierten Kategorien umzusetzen. Darüber hinaus sind die Anforderungen aus Kapitel 4 zu erfüllen, um Modell-Refactorings in dem geforderten Grad generisch zu definieren, damit sie über Metamodell-Grenzen hinweg wiederverwendet werden können.

Die noch junge Disziplin des Modell-Refactorings birgt im aktuellen Stadium der Forschung verschiedene Ansätze. Betrachtet man diese unter dem Gesichtspunkt der MOF-Architektur, so fällt auf, dass die Ansätze danach klassifiziert werden können, auf welcher MOF-Ebene die Modell-Refactorings definiert werden. Es gibt Ansätze, mit denen Modell-Refactorings auf der höchsten Ebene (M3) spezifiziert werden. Die meisten Vorstöße gab es auf der M2-Ebene. Außerdem existiert auch ein M1-Ansatz, der gerade für DSL-Nutzer interessant sein könnte. Durch diese Einteilung ist es sehr gut möglich, verschiedene Ansätze miteinander zu vergleichen. Deshalb werden im Folgenden alle mittels der MOF-Ebenen gefiltert und es wird ein Repräsentant jeder Ebene zunächst neutral im Detail vorgestellt. Deren Stärken und Schwächen werden dann anhand der im vorigen Kapitel aufgestellten Anforderungen ermittelt. Dazu dient eine Bewertung mit folgenden Kriterien: `+` steht für eine erfüllte Anforderung, die mit sehr geringem Aufwand übertragen werden könnte; mit `o` wird eine Eigenschaft bewertet, die nur bedingt nutzbar ist oder den Anforderungen nicht voll entspricht; eine negative Bewertung wird mit `-` angegeben, wenn eine Eigenschaft nicht vorhanden oder der Ansatz nicht nutzbar ist.

5.1 M3-Spezifikation am Beispiel von GenericMT

Die Spezifikation von Modell-Refactorings auf der MOF-Ebene M3 verspricht die vom Ziel-Metamodell unabhängige Definition, da dieses in der MOF-Hierarchie eine Ebene tiefer liegt. Die Autoren Tichelaar et al. haben in [TDDN00] erste Forschungsarbeit auf diesem Gebiet geleistet. Dort stellen sie ein Meta-Metamodell für objektorientierte Programmiersprachen vor, auf dessen Basis Refactorings definiert werden können. Allerdings sprechen die Autoren noch nicht von Modell-Refactoring.

Die Forschung auf der M3-Ebene wurde danach hauptsächlich in dem französischen Projekt *Triskell*[1] und der Gruppe *SoftCom*[2] aus Alabama vorangetrieben. Außerdem

[1] `http://www.irisa.fr/triskell`
[2] `http://www.cis.uab.edu/softcom/`

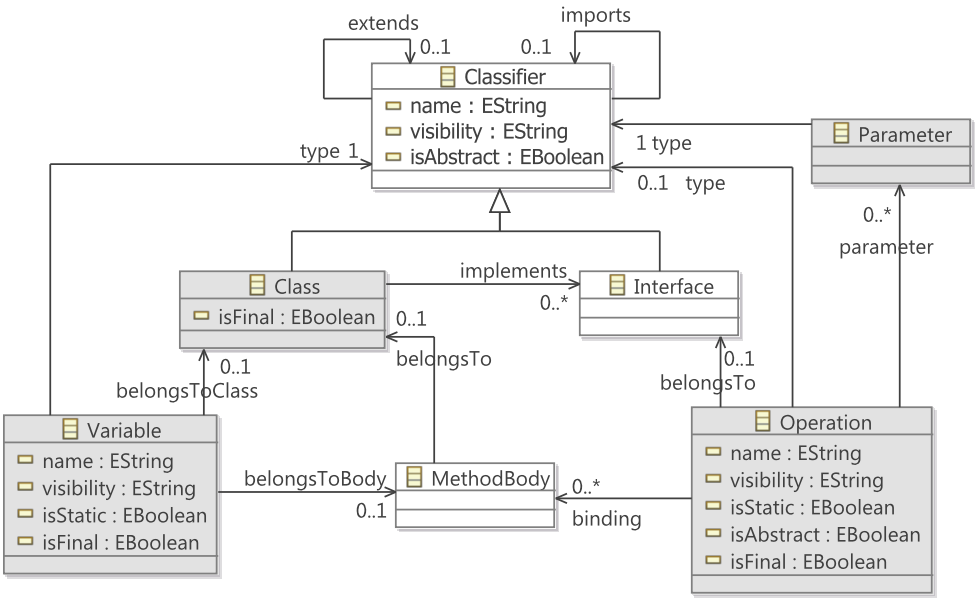


Abbildung 5.1: Ausschnitt eines Java-Metamodells nach [MMBJ09]

hat Marticorena in [Mar05] eine Meta-Sprache diskutiert, mit der man Refactorings unabhängig von der Ziel-Sprache definieren kann. Er beschränkt sich allerdings hauptsächlich auf schon vorhandene Programmiersprachen. Als Repräsentant der M3-Spezifikation wird deshalb das Triskell-Projekt gewählt, da SoftCom ein Meta-Metamodell benutzt, welches nicht auf MOF basiert [ZLG05] und der Triskell-Ansatz ähnlich zu Marticorena ist, jedoch die entscheidende Unabhängigkeit zu existierenden Programmiersprachen aufweist. Außerdem gehört die Triskell-Gruppe zu den weltweit führenden Forschern auf dem Gebiet der modellgetriebenen Entwicklung [@Tri10].

Der Ausgangspunkt für die Triskell-Gruppe war, dass Metamodelle wie Java, MOF oder UML mehrere Konzepte gemein haben [MMBJ09]. Dazu zählen beispielsweise die Konzepte der Klasse, der Methode, des Attributs und der Vererbung. Möchte man aber aufgrund der konzeptionellen Gemeinsamkeiten ein Modell-Refactoring definieren, eröffnen sich zahlreiche Probleme. Zum Verständnis der Probleme werden in den Abbildungen 5.1, 5.2 und 5.3 Ausschnitte aus den drei verwendeten Metamodellen präsentiert. Dabei kommen grau unterlegte Metaklassen in allen drei Metamodellen vor. Im Wesentlichen sind in diesen Abbildungen folgende fünf Probleme zu erkennen:

- Konzeptionell gleiche Elemente besitzen unterschiedliche Namen: das Konzept des Attributs heißt in der MOF und UML `Property` und im Java-Metamodell `Variable`.
- Typen konzeptionell gleicher Attribute sind verschieden: die Sichtbarkeit einer Klasse (Attribut `visibility`) wird in Java mit einem String und in der UML mit einem Wert der Aufzählung `VisibilityKind` angegeben.

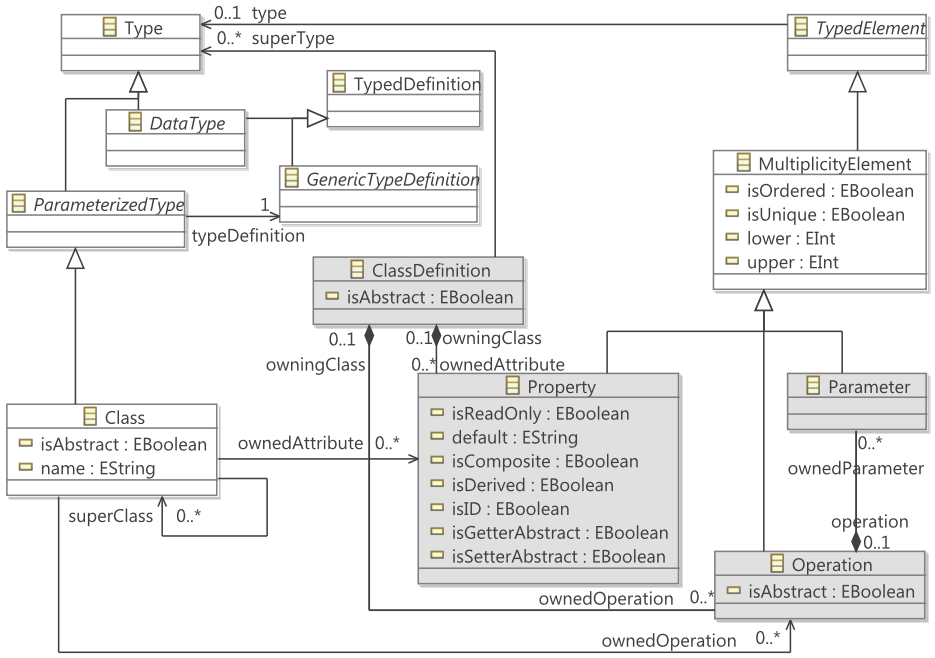


Abbildung 5.2: Ausschnitt des MOF-Metamodells nach [MMBJ09]

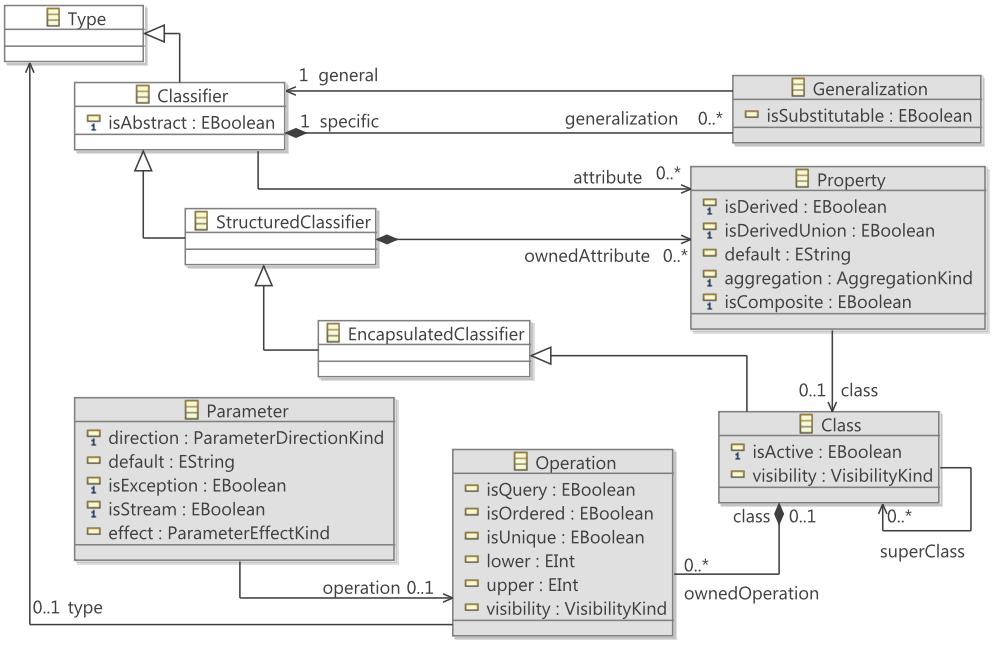


Abbildung 5.3: Ausschnitt des UML-Metamodells nach [MMBJ09]

- Zusätzliche oder fehlende Elemente: das MOF-Metamodell besitzt kein Konzept zur Modellierung der Sichtbarkeit.
- Entgegengesetzte Relationen fehlen: eine Relation ist bidirektional, wenn beide teilnehmenden Elemente zum jeweils anderen navigieren können; die UML-Relation zwischen `Classifier` und `Generalization`, zur Spezifikation von Vererbungsbeziehungen, ist bidirektional, wohingegen das Konzept der Vererbung in Java und MOF nur einseitig navigierbar ist.
- Konzeptionell gleiche Metaklassen sind unterschiedlich verbunden: in Java sind das Konzept des Attributs (`Variable`) und der Methode (`Operation`) nicht direkt vom Konzept der Klasse aus erreichbar.

Aus diesen Problemen resultiert die Tatsache, dass Modell-Refactorings trotz der konzeptionellen Gemeinsamkeiten der Metamodelle nicht wiederverwendet werden können, da die Metamodelle viele strukturelle Unterschiede aufweisen. Daraus ergab sich das Ziel für Triskell, eine Möglichkeit zu schaffen, Modell-Refactorings generisch zu beschreiben und diese auf verschiedene Metamodelle anzuwenden, um die Produktivität und Qualität der MDSD zu erhöhen [MMBJ09]. Um dieses Ziel zu erreichen, haben die Triskell-Forscher ein Meta-Metamodell entwickelt, welches als kleinste gemeinsame Basis der drei vorher genannten Metamodelle dient. Dieses Meta-Metamodell heißt **GenericMT**. Auf dessen Grundlage werden Modell-Refactorings in diesem Ansatz mit der Metasprache **Kermeta** spezifiziert. Zunächst wird Kermeta im Folgenden vorgestellt, bevor danach das GenericMT präsentiert wird.

Kermeta

Kermeta ist eine Sprache zur Metamodellierung, mit der es nicht nur möglich ist, die Struktur von Metamodellen, sondern auch die dynamische Semantik, also das Verhalten von Metaklassen, zu beschreiben [Hub08, S. 46]. Kermeta wurde auf der Grundlage von EMOF entwickelt. Deshalb ist der strukturelle Teil vollständig kompatibel mit EMOF [MFJ05]. Dieser Teil wurde um Metaklassen zur Spezifikation des Verhaltens von Methoden erweitert. Die Erweiterung von existierenden Metamodellen um neue Strukturen und Verhalten ist die Eigenschaft, durch die sich Kermeta auszeichnet. Dadurch ist es möglich, Metamodelle getrennt voneinander zu entwickeln und diese dann wiederzuverwenden, indem sie wie Aspekte ineinander verwoben werden [MMBJ09]. Eine weitere Eigenschaft von Kermeta ist die Definition von abgeleiteten Eigenschaften (*derived properties*). Diese werden in Abhängigkeit von anderen Eigenschaften einer Klasse berechnet und können als Verhalten mit Kermeta beschrieben werden. Des Weiteren wurde für Kermeta eine Entwicklungsumgebung entwickelt [@Ker10], die auch einen Interpreter enthält. Damit können die in Kermeta definierten Operationen auch in jeder Metamodell-Instanz ausgeführt werden.

GenericMT

Um Modell-Refactorings generisch zu beschreiben, wird eine Struktur benötigt, auf der die Definition durchgeführt werden kann. Die Triskell-Gruppe hat zu diesem Zweck das

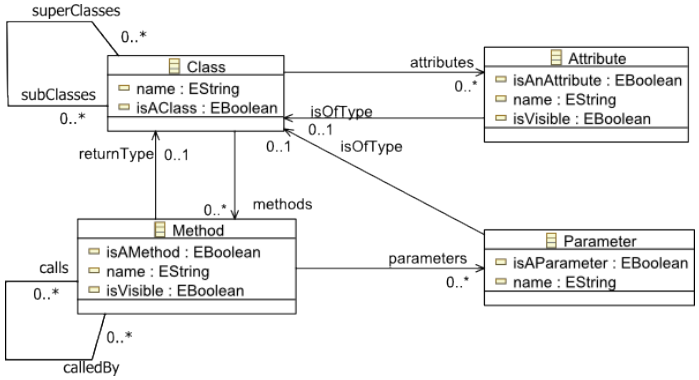


Abbildung 5.4: GenericMT nach [MMBJ09]

leichtgewichtige Metamodell GenericMT entwickelt, in dem diejenigen objektorientierten Konzepte enthalten sind, die auch die oben vorgestellten Ziel-Metamodelle gemein haben. GenericMT ist in Abbildung 5.4 dargestellt. Auf dessen Basis werden nun Modell-Refactorings unabhängig von dem Metamodell, auf das sie später angewendet werden sollen, mit Kermeta spezifiziert. Dazu spezifiziert man Operationen, in denen das Modell-Refactoring definiert wird. In Listing 5.1 ist beispielsweise ein Ausschnitt der Implementierung von *Pull Up Method* dargestellt. In diesem Listing wird deutlich, dass die Klasse

Listing 5.1: *Pull Up Method* mit Kermeta nach [MMBJ09]

```
1  package refactor;
2  class Refactor<MT:GenericMT> {
3    operation pullUpMethod(source : MT::Class, target : MT::Class,
4      meth : MT::Method) : Void
5    //Preconditions
6    pre sameSignatureInOtherSubclasses is do
7      target.subClasses.forAll{sub |
8        sub.methods.exists{op | haveSameSignature(meth, op)}}
9    end
10   //Operation body
11   is do
12     target.methods.add(meth)
13     source.methods.remove(meth)
14   end
15 }
```

`Refactor` generisch deklariert ist und einen Parameter vom Typ `GenericMT` erwartet. Das vorangestellte `MT` dient dem Zugriff auf die übergebene Klasse bei der Implementierung. Diese Technik ist ausschlaggebend für die generische Spezifikation der neuen Operation `pullUpMethod`. Darin wird das Modell-Refactoring definiert. Nun bleibt nur noch die Frage offen, wie die Anwendung eines Modell-Refactorings auf ein bestimmtes Metamodell gesteuert werden kann. Da dieses mit Kermeta auf dem GenericMT spezifiziert wurde, wird eine Abbildung des jeweiligen Ziel-Metamodells auf GenericMT benötigt. Diese Abbildung wird in dem Triskell-Ansatz wiederum mit Kermeta durchgeführt.

Dazu wird für jedes Metamodell einmalig definiert, welche Metaklassen den GenericMT-Klassen entsprechen und wie die Relationen dazwischen umgesetzt werden. Diese Art der Adaption modifiziert demnach virtuell die Struktur des Ziel-Metamodells und erzeugt so Konformität gegenüber GenericMT [MMBJ09]. In Listing 5.2 wird beispielsweise dargestellt, wie das Metamodell der UML auf GenericMT adaptiert wird. Darin ist zum

Listing 5.2: Adaption des UML-Metamodells auf GenericMT nach [MMBJ09]

```
1  package uml;
2  require "http://www.eclipse.org/uml2/2.1.2/UML"
3  aspect class Classifier {
4    reference inv_general : Generalization[0..*]#general
5  }
6  aspect class Class{
7    property superClasses : Class[0..*]#subClasses
8      getter is do
9        result := OrderedSet<uml::Class>.new
10       self.generalization.each{g | result.add(g.general)}
11     end
12   property subClasses : Class[0..*]#superClasses
13     getter is do
14       result := OrderedSet<uml::Class>.new
15       self.inv_general.each{g | result.add(g.specific)}
16     end
17 }
```

Beispiel zu erkennen, dass die Relation `superClasses` aus dem GenericMT in der UML mit der Metaklasse `Generalization` und deren Beziehungen zur Metaklasse `Classifier` simuliert wird. Das # gibt jeweils an, welche Eigenschaft die entgegengesetzte Relation darstellt. Diese Adaption wird nun beim Aufruf in das UML-Metamodell gewoben, wodurch die dort spezifizierten Aspekte im konkreten UML-Modell verfügbar sind. Durch die Parametrisierung der in Listing 5.1 gezeigten Klasse `Refactor` ist es nun möglich, die dort definierten Modell-Refactorings auf alle Metamodelle anzuwenden, für die eine zum Listing 5.2 äquivalente Adaption existiert. Beim Aufruf eines Modell-Refactorings wird dann das jeweilige adaptierte Metamodell übergeben und die Transformationsschritte können darauf angewendet werden. Des Weiteren braucht eine Abbildung auf das GenericMT nur einmal durchgeführt zu werden. Danach sind zukünftige Modell-Refactorings stets in dem jeweiligen Metamodell anwendbar.

Bewertung

Die Triskell-Gruppe konnte mittels des Meta-Metamodells GenericMT und der Metasprache Kermeta einen Ansatz umsetzen, mit dessen Hilfe Modell-Refactorings generisch zu spezifizieren sind. Durch die Parametrisierbarkeit der Spezifikation eines Modell-Refactorings mit Metamodellen, für die eine Adaption auf das GenericMT existiert, kann somit dasselbe Modell-Refactoring in verschiedenen Metamodellen wiederverwendet werden. Damit wird die Hauptanforderung der Generizität und der Wiederverwendbarkeit erfüllt. Trotzdem weist dieser Ansatz auch einige Schwächen auf, welche in der Tabelle 5.1 genannt und anschließend erläutert werden.

Tabelle 5.1: Bewertung des Triskell-Ansatzes

Bezeichnung		**Bemerkungen**
1) Generizität & Wiederverwendbarkeit	+	generische Spezifikation mit Kermeta durch Nutzung von GenericMT; dadurch Wiederverwendbarkeit in verschiedenen Metamodellen möglich
2) Kategorien	o	Modell-Refactorings nur auf objektorientierte Metamodelle übertragbar
3) Sprachspezifika	o	Spezifikation des Modell-Refactorings nicht sprachspezifisch anpassbar, jedoch kann dafür ein eigenes Modell-Refactoring definiert werden
4) vertikale & horizontale Konsistenz	-	nur Modell-lokale Aktualisierung
5) Beibehaltung der Semantik	-	keine Angaben
6) Vor- und Nachbedingungen	+	Vor- und Nachbedingungen können für die generische Spezifikation per Object Constraint Language (OCL) angegeben werden
7) Atomarität	-	keine Angaben
8) Umkehrbarkeit	-	keine Angaben
9) Empfehlung von Modell-Refactorings	-	keine Angaben
10) Interoperabilität	+	MOF-Konformität ist die einzige Anforderung an die Metamodelle
11) Erweiterbarkeit	+	Menge der Modell-Refactorings erweiterbar; außerdem bietet Kermeta exzellente Möglichkeiten, Metamodelle virtuell zu erweitern
12) Benutzbarkeit & Integrationsfähigkeit	o	für gewöhnliche Eclipse-Nutzer nicht sehr intuitiv
13) Durchführungsgeschwindigkeit	-	konnte nicht untersucht werden
14) Unterstützung von Standards	+	MOF, OCL

Eine wichtige Stärke des Triskell-Ansatzes ist es, dass mit Kermeta Vor- und Nachbedingungen spezifiziert werden können. Diese werden in Form von OCL angegeben, wodurch der Entwickler von Modell-Refactorings mit einer ihm bekannten Sprache konfrontiert wird. Außerdem bieten Kermeta und das damit verbundene Tooling eine sehr ausgereifte Möglichkeit, Metamodelle dynamisch zu erweitern und so vorhandene Elemente auf GenericMT abzubilden. Dadurch kann das GenericMT einfach auf beliebige Metamodelle übertragen werden.

Die Kermeta-Workbench integriert sich zudem nahtlos in die IDE Eclipse[3]. Auf dieser Basis setzt Kermeta selbst auch auf die EMOF-Implementierung Ecore von Eclipse auf, wodurch zwei weitere Vorteile zum Tragen kommen. Durch die Erweiterung von EMOF wird zum einen ein Standard verwendet und zum anderen auch ein Standard unterstützt. Daraus resultiert prinzipiell die einzige Anforderung an zu adaptierende Metamodelle: sie müssen MOF-Konformität aufweisen. Jedoch wird der Vorteil dieser einzigen Anforderung durch die Beschaffenheit des GenericMT wieder abgeschwächt. Dieses besitzt ausschließlich objektorientierte Konzepte, wodurch auch nur objektorientierte Modell-Refactorings umgesetzt werden können. Dies bedeutet, dass sich daraus eine zweite Anforderung an zu adaptierende Metamodelle herausstellt: sie müssen objektorientierten Charakter haben. DSLs, die diese Anforderung nicht erfüllen, können

[3] `http://www.eclipse.org`

zwar trotzdem adaptiert werden, wurden aber von der Triskell-Gruppe nicht vorgesehen. Außerdem zieht eine Adaption von Metaklassen eines nicht objektorientierten Metamodells einen Bruch in der Bedeutung der Metaklassen von GenericMT nach sich. Soll beispielsweise das in diesem Kapitel verwendete Beispiel *Pull Up Method* auf die in Abschnitt 3.2.3 vorgestellte GUI-Sprache übertragen werden, so müsste die Metaklasse `UIElement` (oder Unterklassen wie `Button`, `Label` oder `Image`) so adaptiert werden, dass sie der GenericMT-Klasse `Method` entspricht. Damit geht jegliche Intention verloren.

Zudem birgt das GenericMT einen weiteren Nachteil. Wurde ein Metamodell einmal darauf adaptiert, sind die zu refaktorierenden Strukturen im Ziel-Metamodell festgelegt. Dies bedeutet, dass ein Modell-Refactoring nicht auf mehrere zusammenhängende Teile desselben Metamodells angewendet werden kann. Beispielsweise ist das UML-Metamodell sehr komplex und beinhaltet einerseits einen Teil zur Spezifikation von Klassenmodellen und andererseits einen Teil zur Spezifikation von Zustandsmaschinen. Wie in Anforderung 3 schon erwähnt, kann *Extract Method* in Zustandsmaschinen als *Extract Composite State* angewendet werden. Dies wäre mit dem Triskell-Ansatz jedoch nicht mehr möglich, wenn die Adaption auf das GenericMT schon mit den Metaklassen des UML-Teils der Klassenmodelle abgedeckt ist. Somit ist es zum einen unmöglich, exakt zu kontrollieren, auf welche Teile eines Metamodells ein Modell-Refactoring angewendet werden kann, und zum anderen, dasselbe Modell-Refactoring auf mehrere Teile im selben Metamodell zu aktivieren.

Zusammenfassend kann man sagen, dass die Generizität des Triskell-Ansatzes als sehr positiv zu bewerten ist. Die Beschränkung auf ein vorgegebenes Meta-Metamodell birgt jedoch ihre Schwächen. Die Betrachtung dieses Vorteils und dieses Nachteils im Zusammenhang wird im Kapitel 6 durchgeführt und lässt einen neuen generischen Ansatz erhoffen.

5.2 M2-Spezifikation am Beispiel von EMF Refactor

Ein Vorteil der Spezifikation von Modell-Refactorings auf der MOF-Ebene M2 liegt im Gegensatz zur M3-Spezifikation darin, dass exakt kontrolliert werden kann, auf welchen Strukturen im Metamodell refaktoriert werden soll. Die ersten wegweisenden Erkenntnisse auf diesem Gebiet kamen von Sunyé et al. in [SPTJ01]. Darin wurden einige Modell-Refactorings für die UML vorgestellt. Die Autoren diskutieren zudem das Problem der Erhaltung der Bedeutung. Als mögliche Lösung schlagen sie die Bereitstellung von Basis-Refactorings vor, welche die Semantik garantiert bewahren. Beweise ihrer vorgestellten Modell-Refactorings konnten unformal erbracht werden.

Des Weiteren haben die Forschergruppen der Universität Marburg[4] und der Technischen Universität Berlin[5] maßgeblich dazu beigetragen, Modell-Refactorings auf der Ebene des Ziel-Metamodells zu spezifizieren. Aus ihrer Arbeit ist das Framework *EMF Refactor* entstanden, welches zur Integration in die IDE Eclipse freigegeben wurde [@EMFR]. In dieses Framework flossen mehrere theoretische Arbeiten ein, wodurch

[4] `http://www.uni-marburg.de/fb12/swt/`
[5] `http://www.tu-berlin.de/tfs`

es sich mittlerweile in einem sehr ausgereiften Stadium befindet. Außerdem stellt das Eclipse-Konsortium hohe Anforderungen an die Qualität, weshalb bei *EMF Refactor* von hoher Güte und Praxistauglichkeit ausgegangen werden kann. Deshalb wird *EMF Refactor* hier als Vertreter der Spezifikation auf MOF-Ebene M2 gewählt und vorgestellt.

Das Projekt um *EMF Refactor* hat sich zum Ziel gesetzt, Modell-Refactorings graphisch definieren zu können, um den Prozess der Spezifikation bestmöglich zu vereinfachen [Köh06]. Die wissenschaftliche Grundlage dieses Ansatzes liegt darin, dass Modelle zwar Baumstrukturen aufweisen können, jedoch prinzipiell Graphen darstellen. Um Graphen zu transformieren, wurde in diesem Zusammenhang das Werkzeug *AGG*[6] entwickelt, das es ermöglicht, neben der Transformation von Graphen diese auch formal auf Konsistenz zu analysieren [TMM08]. Darauf aufbauend wurde das Framework *EMF Tiger* entwickelt, das eine graphische Spezifizierung derartiger Graph-Transformationen ermöglicht. Zur Spezifikation eines Modell-Refactorings muss ein sogenanntes *Transformation System* definiert werden. Wie dies angegeben wird, beschreibt der folgende Abschnitt.

Transformation System

Ein *Transformation System* stellt ein System von graphischen Regeln dar, mit dem die Transformationen, die in einem Modell-Refactoring durchgeführt werden, spezifiziert werden [BEK+06b]. Ein *Transformation System* bezieht sich immer auf das Metamodell, für dessen Instanzen Modell-Refactorings zur Verfügung gestellt werden sollen, und definiert genau ein Modell-Refactoring [BEK+06a]. Eine Regel hat dabei die Objektstrukturen *Left Hand Side (LHS)* und *Right Hand Side (RHS)*. Die LHS beinhaltet die Vorbedingungen, die von einem Modell zu erfüllen sind. Analog beinhaltet die RHS die Bedingungen, die nach dem Modell-Refactoring in dem zu refaktorierenden Modell zutreffen müssen. LHS und RHS sind selbst wieder Modelle, für die eine graphische Repräsentation definiert wurde [Köh06]. In beiden Modellen können nun Objektstrukturen zwischen Metaklassen des referenzierten Metamodells definiert werden. Dieselben semantischen Objekte der LHS und RHS werden über die gleiche, dem Namen der jeweiligen Metaklasse vorangestellte Zahl miteinander verbunden. Dadurch entstehen Mappings von Objekten der LHS auf Objekte der RHS. Ein derartiges Mapping steuert nun das Erzeugen und Entfernen von Modell-Elementen. Besitzen Elemente der LHS kein Abbild in der RHS, bedeutet dies, dass diese während des Modell-Refactorings gelöscht werden. Im Gegensatz dazu werden Elemente der RHS erzeugt, auf denen keine Elemente der LHS abgebildet wurden. Dabei stellt die LHS eine Art positive Bedingung für die Durchführung des Modell-Refactorings dar. Jedoch können auch negative Bedingungen vonnöten sein, die optional als *Negative Application Conditions (NAC)* zu spezifizieren sind [MTM08]. Damit sind solche Bedingungen zu definieren, die bestimmte Strukturen im zu refaktorierenden Modell verbieten und demnach nicht vorkommen dürfen. Werden mehrere NACs angegeben, müssen alle im Modell zutreffen.

Um dies zu veranschaulichen, haben die Autoren von [BEK+06a] das Modell-Refactoring *Pull Up Attribute* für das schon erwähnte Ecore vorgestellt. Das Metamodell von

[6] `http://user.cs.tu-berlin.de/~gragra/agg/`

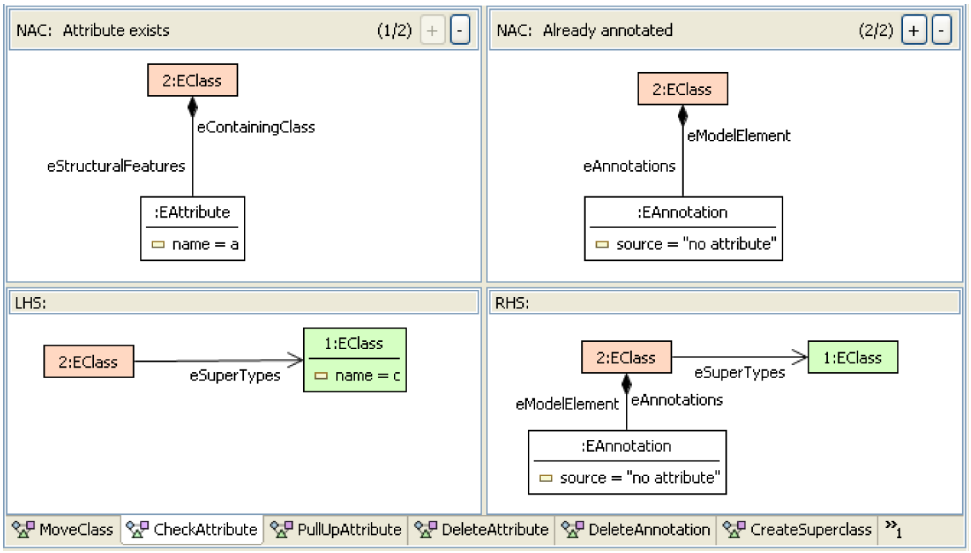


Abbildung 5.5: Regel *Check Attribute* von *Pull Up Attribute* nach [BEK+06a]

Ecore wird an dieser Stelle nicht näher erläutert, da die im Folgenden verwendeten Metaklassen und deren Beziehungen einfach zu verstehen sind. *Pull Up Attribute* kann in Ecore dazu benutzt werden, Attribute einer Metaklasse in eine ihrer Superklassen zu verschieben. Dieses Modell-Refactoring entspricht dem in Abschnitt 3.1.4 erläuterten *Pull Up Member*. Das *Transformation System* hierzu besitzt als erste Regel *CheckAttribute*, wie in Abbildung 5.5 dargestellt. Damit wird überprüft, ob von der angestrebten Superklasse mit dem Namen `c` eine Unterklasse erbt, die das zu verschiebende Attribut `a` nicht besitzt. Die Platzhalter `a` und `c` sind Variablen, welche für jede Regel definiert werden können, und dienen als Eingabe des Modell-Refactorings. Außerdem ist in Abbildung 5.5 zu erkennen, dass mit der RHS, für eine Metaklasse ohne Attribut `a`, eine Annotation mit dem Text `no attribute` erzeugt wird. Diese dient temporär der Identifizierung solcher Metaklassen ohne `a`. Ist eine derartige Annotation schon vorhanden, wird mit der zweiten NAC ausgedrückt, dass diese nicht ein weiteres Mal erstellt wird. Diese Eigenschaft der Regel ist wichtig, um keine Endlosschleife zu provozieren.

In der zweiten Regel wird dann das Verschieben des Attributes in die Superklasse spezifiziert, wenn keine Unterklasse die zuvor erklärte Annotation `no attribute` besitzt. Diese Regel ist in der Grafik 5.6 abgebildet. In der dort dargestellten RHS und der NAC *Attribute already pulled up* ist zu erkennen, dass nun das Attribut in die Superklasse verschoben wird, wenn diese noch kein äquivalentes Attribut besitzt. Nach der erfolgreichen Verschiebung, wenn also alle NACs und die LHS erfüllt sind, muss nun in jeder Unterklasse das ursprüngliche Attribut entfernt werden. Diese Regel wird in Abbildung 5.7 dargestellt. Da das Attribut in der LHS nicht in der RHS abgebildet ist, wird dieses, wie zuvor beschrieben, in diesem *Transformation System* gelöscht. War die Verschiebung nicht erfolgreich, beispielsweise, wenn eine Unterklasse das zu verschiebende Attribut nicht besitzt, dann besitzt diese die in der Regel *Check Attribute* erzeugte

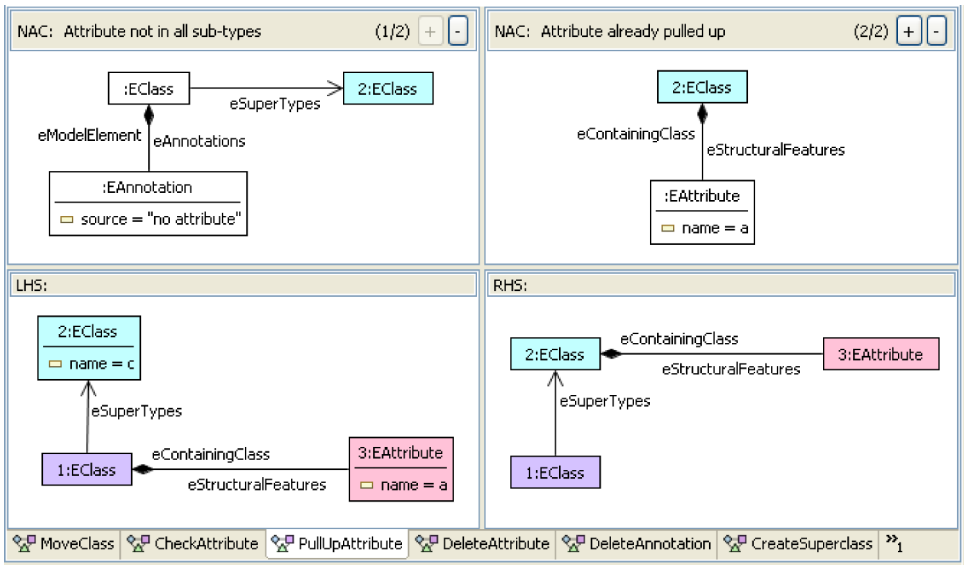


Abbildung 5.6: Regel *Pull Up Attribute* nach [BEK+06a]

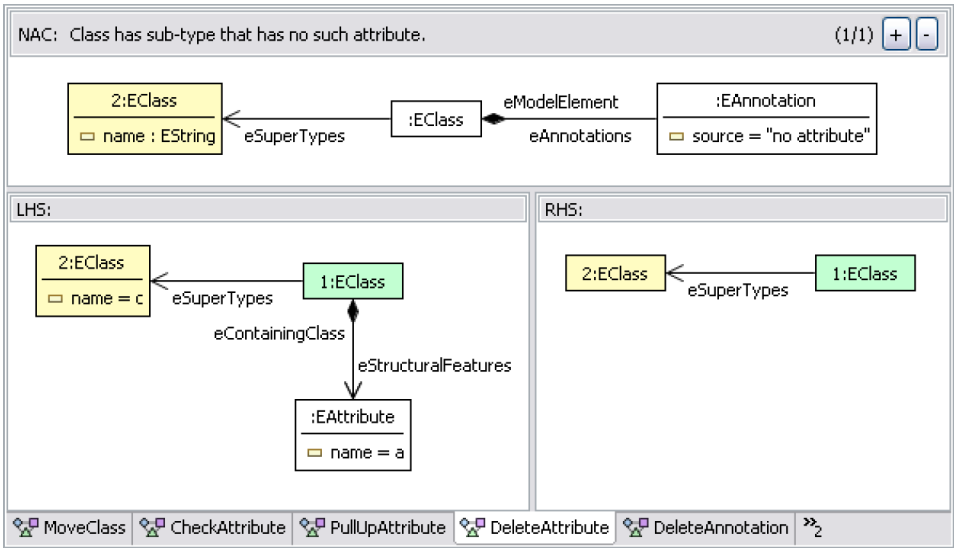


Abbildung 5.7: Regel *Delete Attribute* von *Pull Up Attribute* nach [BEK+06a]

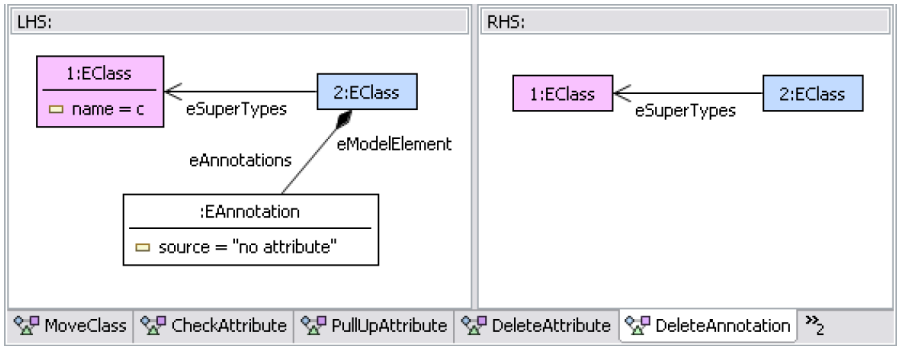


Abbildung 5.8: Regel *Delete Annotation* von *Pull Up Attribute* nach [BEK+06a]

Annotation. Diese muss im letzten Schritt entfernt werden und ist in Abbildung 5.8 zu sehen. An diesen vier erläuterten Regeln ist zu erkennen, dass die Reihenfolge der Abarbeitung von Bedeutung ist. Diese steht fest und darf nicht willkürlich geschehen. Regeln müssen dazu in einem *Transformation System* in einer Ebene definiert werden. Die Reihenfolge der Ebenen spezifiziert dann den Kontrollfluss des Modell-Refactorings. Ein weiterer Nicht-Determinismus tritt ein, wenn die Objektstruktur der LHS mehrmals im zu refaktorierenden Modell gefunden werden konnte. In so einem Fall muss der Anwender entscheiden, auf welche konkrete Struktur das Modell-Refactoring angewendet werden soll [BEK+06b].

Wie oben schon erwähnt, besteht ein Vorteil der Graph-Transformation darin, dass mit ihrer Hilfe die Regeln von Modell-Refactorings auf Konsistenz geprüft oder aber auch Abhängigkeiten zwischen verschiedenen Modell-Refactorings ermittelt werden können. Damit wird es dem Anwender beispielsweise ermöglicht festzustellen, welches Modell-Refactoring ein anderes ausschließt, wenn dieses zuerst angewendet wird. Somit können Aussagen darüber getroffen werden, in welchem Maße eine Sequenz von Modell-Refactorings die Qualität des Modells verbessert. Der Nutzer kann so bei der Wahl des durchzuführenden Modell-Refactorings unterstützt werden, denn er kann eine Empfehlung erhalten, welches Modell-Refactoring in einem bestimmten Kontext passend ist [MTR07].

Bewertung

Die Forscher und Entwickler, deren Arbeit in das Projekt *EMF Refactor* einfloss, haben damit ein ausgereiftes Framework geschaffen, mit dem sprachspezifische Modell-Refactorings spezifiziert und durchgeführt werden können. Die Tatsache, dass Modell-Refactorings immer auf Basis eines konkreten Metamodells definiert werden, bringt den Vorteil mit sich, dass alle in Kapitel 3.2 aufgestellten Kategorien unterstützt werden. Demnach spielt es im Gegensatz zum Triskell-Ansatz keine Rolle, welche Beschaffenheit die Metamodelle haben oder ob es sich um DSLs handelt, die objektorientierte Strukturen besitzen. Die nachfolgende Tabelle 5.2 enthält alle weiteren Bewertungen, die danach begründet werden.

Die größte Stärke dieses Ansatzes, Modell-Refactorings nur sprachspezifisch zu spe-

Tabelle 5.2: Bewertung des Ansatzes von *EMF Refactor*

Bezeichnung		**Bemerkungen**
1) Generizität & Wiederverwendbarkeit	-	nur sprachspezifische Modell-Refactorings möglich
2) Kategorien	+	jedes MOF-konforme Metamodell kann als Ziel dienen; demnach werden alle Kategorien unterstützt
3) Sprachspezifika	+	da nur sprachspezifische Modell-Refactorings definiert werden können, werden Spezifika sehr gut unterstützt
4) vertikale & horizontale Konsistenz	-	nur Modell-lokale Aktualisierung
5) Beibehaltung der Semantik	o	durch Angabe von NACs können teilweise Strukturen definiert werden, die ein Modell-Refactoring ausschließen, weil dadurch die im Kontext wichtigen Semantik-Eigenschaften nicht erhalten bleiben
6) Vor- und Nachbedingungen	o	Vorbedingungen können mit LHS und NACs angegeben werden
7) Atomarität	-	keine Angaben
8) Umkehrbarkeit	-	keine Angaben
9) Empfehlung von Modell-Refactorings	o	teilweise durch Analyse von Beziehungen zwischen Modell-Refactorings mit AGG möglich
10) Interoperabilität	+	MOF-Konformität ist die einzige Anforderung an die Metamodelle
11) Erweiterbarkeit	+	Menge der Modell-Refactorings erweiterbar
12) Benutzbarkeit & Integrationsfähigkeit	+	durch graphische Spezifikation sehr benutzerfreundlich
13) Durchführungsgeschwindigkeit	o	Analyse der Beziehungen zwischen den Modell-Refactorings beansprucht viel Zeit [MTR07]
14) Unterstützung von Standards	+	MOF

zifizieren, ist gleichzeitig auch seine größte Schwäche. Mit *EMF Refactor* ist es nicht möglich, Modell-Refactorings einmalig zu definieren und diese dann für verschiedene Metamodelle wiederzuverwenden. Zwar kann ein *Transformation System* als Grundlage für ein gleichartiges Modell-Refactoring eines anderen Metamodells dienen, jedoch würde der Entwickler dann dem fehleranfälligen *Copy & Paste* erliegen. Außerdem muss dieses *Transformation System* anschließend an das neue Metamodell angepasst werden, wie beispielsweise durch Austausch der analogen Metaklassen. Dieser Ansatz läuft letztendlich darauf hinaus, dass gleichartige Modell-Refactorings für jedes Metamodell neu spezifiziert werden müssen, wodurch zusätzliche Fehlerquellen entstehen und das Verfahren sehr viel aufwändiger wird.

Des Weiteren basiert *EMF Refactor*, ebenso wie der Triskell-Ansatz, auf dem in der Metamodellierung benutzten Quasistandard Ecore und setzt daher MOF-konforme Metamodelle voraus. So ist es als positiv zu bewerten, dass der Standard EMOF verwendet wird und die Modell-Refactorings auf dem Modell ausgeführt werden können, ohne dass Abhängigkeiten zur jeweiligen Modell-Repräsentation entstehen.

Eine weitere positive Eigenschaft ist, dass mit Hilfe von NACs Bedingungen spezifi-

ziert werden können, mit denen die Ausführung von Modell-Refactorings unterbunden werden kann, wenn bestimmte Teile der Semantik verändert werden. Mit einer NAC ist zwar nicht primär die Semantik spezifizierbar, sehr wohl jedoch ein kleiner Teil davon. Und zwar ist es möglich Strukturen anzugeben, die im Kontext eines konkreten Metamodells und Modell-Refactorings Teile der Semantik des jeweiligen Modells darstellen. So wurde im oben erläuterten Beispiel definiert, dass *Pull Up Attribute* nicht durchgeführt werden darf, wenn die Superklasse eine Unterklasse hat, die das zu verschiebende Attribut nicht besitzt. Wäre dies erlaubt und demnach eine Verschiebung eines Attributes im Allgemeinen möglich, so würde dies bedeuten, dass Unterklassen nach *Pull Up Attribute* ein zusätzliches Attribut besitzen. Je nach Anwendungsfall kann dies ein Kriterium für die Beibehaltung der Semantik sein oder auch nicht. Spielt es für den einen Anwender eine Rolle, ob Metaklassen neue Attribute erhalten, so kann mit dieser NAC das Modell-Refactoring ausgeschlossen werden, da damit in seinem Kontext die Semantik nicht erhalten bleibt.

In *EMF Refactor* wurden außerdem erste konkrete Erkenntnisse darüber gewonnen, wie der Nutzer bei der Wahl eines möglichen Modell-Refactorings unterstützt werden kann. Mens et al. räumen jedoch ein, dass die Analyse der Beziehungen zwischen den Modell-Refactorings derzeit noch viel Zeit in Anspruch nimmt [MTR07]. Doch wird damit eine Richtung gewiesen, die weiter verfolgt werden kann.

Zusammenfassend kann gesagt werden, dass *EMF Refactor* viele sehr gute Techniken verwendet, um Modell-Refactorings sprachspezifisch zu definieren. Mankos sind lediglich der Mangel an Generizität und die nicht vorhandene Berücksichtigung der vertikalen und horizontalen Konsistenz. Trotzdem ist nicht zuletzt der Fakt, dass dieses Framework in Eclipse integriert werden wird, ein Indikator für seine Ausgereiftheit und hohe Qualität.

5.3 M1-Spezifikation am Beispiel des Object Recorders

Die Spezifikation auf der MOF-Ebene M1 ist ein sehr pragmatischer Ansatz. Dabei werden Modell-Refactorings auf der Ebene definiert, in der sie später auch angewendet werden sollen. Dies bedeutet auch, dass bei diesem Ansatz die Stakeholder[7] *Modell-Refactoring-Designer* und *Modell-Refactoring-Nutzer*, mehr noch als bei den anderen beiden Ansätzen, nicht zwingend unterschiedliche Personen sein müssen. Die Ursache dafür ist, dass sich Designer und Entwickler auf derselben Ebene der MOF befinden und dadurch kein tieferes Wissen über das Ziel-Metamodell vorausgesetzt zu werden braucht [WSKK07]. Dadurch ist dieser Ansatz sehr benutzerfreundlich.

Es gibt derzeit nicht sehr viele Forschergruppen, die sich mit der Spezifikation von Modell-Refactorings auf der M1-Ebene auseinandersetzen. Ein erster Ansatz, der in diese Richtung geht, wurde von Varró in [Var06] veröffentlicht. Jedoch war es das Ziel dieser Arbeit, allgemein M2M-Transformationen durch Spezifikation an Beispielmodellen zu spezifizieren. Das bedeutet, dass der Fokus nicht auf dem Refactoring von Modellen lag, sondern in der Transformation eines Eingangsmodells in ein Ausgangsmodell, wobei

[7] Begriff aus der Software-Entwicklung zur Bezeichnung der unterschiedlichen Akteure, die an der Entwicklung oder Nutzung einer Software teilhaben

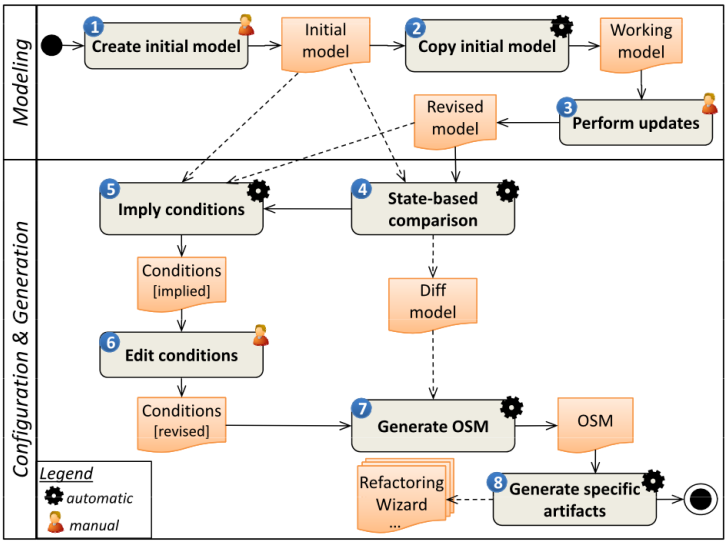


Abbildung 5.9: Prozess der Spezifikation mit dem *Operation Recorder* nach [BLS+09]

beide unterschiedliche Metamodelle besitzen.

Das österreichische Projekt *AMOR*[8] arbeitet hingegen auf dem Gebiet der Modell-Versionierung, das in engem Zusammenhang zum Modell-Refactoring steht. Diese Forschergruppe hat einen Ansatz geschaffen, Anwender von Modell-Refactorings auch zu deren Spezifikation zu befähigen. Aus den Arbeiten des AMOR-Projektes ist der *Operation Recorder* hervorgegangen, der eine Möglichkeit bietet, Modell-Refactorings zustandsbasiert zu definieren. Dieser Ansatz wird im Folgenden detailliert erläutert.

Die Grundlage dieses Ansatzes ist es, zwei verschiedene Zustände eines Beispielmodells zu vergleichen und daraus die von dem Nutzer vorgenommenen Änderungen abzuleiten. Diese Änderungen repräsentieren das durchgeführte Modell-Refactoring. Der Ursprung dieser Herangehensweise liegt darin, dass das Aufzeichnen jeder einzelnen Änderung ein intensives Eingreifen in die IDE zur Folge hätte [BLS+09]. Kann man dies vermeiden, begibt man sich nicht in Abhängigkeit vom jeweils verwendeten Modell-Editor oder der Modell-Repräsentation. Dies wird mit dem Vergleich des Modell-Zustands vor und nach dem Modell-Refactoring erreicht, was im *Operation Recorder* umgesetzt wurde. Welche einzelnen Schritte dazu notwendig sind, wird im nachfolgenden Abschnitt erklärt.

Operation Specification Process

Der Prozess der Spezifikation eines Modell-Refactorings verläuft in zwei verschiedenen Phasen. Jede Phase ist in mehrere Schritte unterteilt, bei denen der Anwender vom *Operation Recorder* unterstützt wird. Dieser Prozess ist in Abbildung 5.9 schematisch dargestellt. In der ersten Phase wird die Eingabe für die zweite Phase produziert. Dazu muss der Nutzer das *Initial Model* erzeugen, welches den Ausgangszustand repräsentiert. Dieser Schritt muss nicht bewusst als Aktion der ersten Phase durchgeführt werden, da hier

[8] `http://www.modelversioning.org`

auch schon vorhandene Modelle benutzt werden können. Daraus wird durch den *Operation Recorder* das *Working Model* erzeugt, welches eine Kopie des *Initial Model* darstellt. Zur eindeutigen Identifikation aller Modell-Elemente werden Originalelement und Kopie mit dem gleichen Identifikator annotiert. Die Teilschritte des Modell-Refactorings werden nun im *Working Model* durchgeführt. Dazu nimmt der Nutzer einmalig diejenigen Änderungen manuell vor, die später automatisch als Modell-Refactoring durchgeführt werden sollen. Damit wird auch die Intention der Identifikatoren klar. Mit ihnen ist es möglich, stets die Verbindung zwischen Originalelement und Kopie zu erhalten, auch wenn sich aussagekräftige Eigenschaften, wie beispielsweise der Name, geändert haben. Sind alle Änderungen im Modell vollzogen, stellt dieses nun das *Revised Model* dar, welches als Zielzustand zu verstehen ist. *Initial Model* und *Revised Model* dienen nun in der Phase der *Configuration & Generation* als Eingabe.

In dieser Phase wird als erstes aus den beiden Eingabemodellen automatisch das *Diff Model* erzeugt, welches die atomaren Änderungen enthält und die Differenz zwischen *Initial Model* und *Revised Model* verkörpert. Daraus leitet der *Operation Recorder* im fünften Schritt die Vor- und Nachbedingungen ab [BSW+09]. Der Nutzer kann diese nun begutachten, da sie, bezogen auf das ursprüngliche Modell, sehr spezifisch sind. Um die meist zu restriktiven Bedingungen zu lockern, werden diese vom Anwender im sechsten Schritt angepasst, wodurch eine angepasste Version der Vor- und Nachbedingungen entsteht. Diese liefern, zusammen mit dem *Diff Model*, die Eingabe für die Generierung des *Operation Specification Model (OSM)*. Die bis hierher durchgeführten Schritte werden in dem in Abbildung 5.10 dargestellten Benutzerdialog vorgenommen. In dieser Abbildung ist beispielhaft die Definition des Modell-Refactorings *Extract Composite State* dargestellt, welches die Autoren von [BLS+09] *Introduce Composite State* genannt haben. Dort sind außerdem die einzelnen durchnummerierten Schritte nochmals farbig hervorgehoben. In einem letzten Schritt werden nun aus dem OSM alle Artefakte generiert, die benötigt werden, um das spezifizierte Modell-Refactoring ohne Abhängigkeit vom *Operation Recorder* durchzuführen. Dazu zählt zum einen ein Wizard, in dem bei der Durchführung alle nötigen Benutzereingaben durchgeführt werden. Zum anderen werden die überarbeiteten Vor- und Nachbedingungen in OCL-Code transformiert, wodurch eine weitere standardisierte Sprache unterstützt wird.

Das OSM birgt noch einen weiteren wichtigen Vorteil. Da dort die Differenz zwischen zwei Modell-Versionen ebenso gespeichert ist wie die Vor- und Nachbedingungen, könnte das OSM dazu benutzt werden, eine Sequenz von manuell durchgeführten Modell-Änderungen zu einem Modell-Refactoring aufzulösen. Werden diese beispielsweise in der IDE oder auf einem Server als Modell-Historie gespeichert, können die atomaren Änderungen mit einem OSM verglichen und diese dann gegebenenfalls mit dem Bezeichner des Modell-Refactorings benannt werden. So kann die Evolution eines Modells noch aussagekräftiger dokumentiert werden.

Bewertung

Der *Operation Recorder* setzt die Spezifikation von Modell-Refactorings auf der MOF-Ebene M3 um. Damit wird der Nutzer von Modell-Refactorings konzeptionell auch in

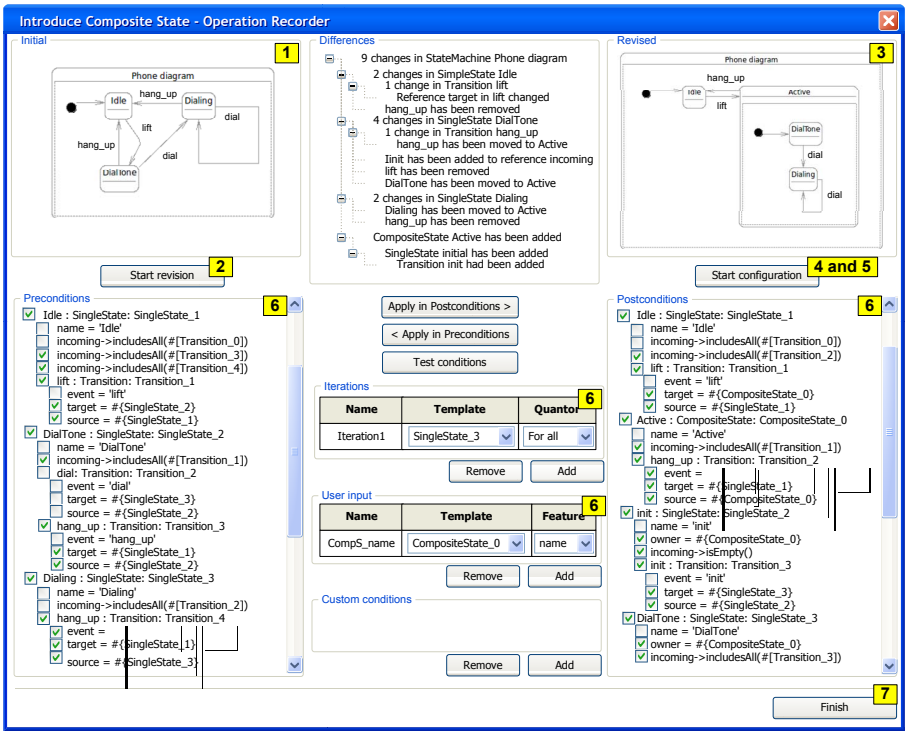


Abbildung 5.10: Benutzerdialog zur Erstellung eines Modell-Refactorings mit dem *Operation Recorder* nach [BLS+09]

dessen Entwurf, der mit dem *Operation Recorder* eng in den normalen Modellierungsprozess eingebunden werden kann, einbezogen. Modell-Refactorings sind so verhältnismäßig schnell zu entwickeln und an die Ziel-Metamodelle wird nur die Forderung der MOF-Konformität gestellt. Deshalb ist es mit diesem Ansatz möglich, alle in Kapitel 3.2 dargestellten Kategorien umzusetzen. In der nachfolgenden Tabelle 5.3 sind die Bewertungen aller Anforderungen zu finden, die anschließend erläutert werden.

Mit dem *Operation Recorder* ist es möglich, ein Modell-Refactoring mit Hilfe eines Beispielmodells zu spezifizieren. Dadurch ist der Einfluss von Sprachspezifika als sehr positiv zu bewerten. Ähnlich dem Ansatz von *EMF Refactor* resultiert aus dieser Stärke auch gleichzeitig seine größte Schwäche. Der *Operation Recorder* bietet keine Möglichkeit, Modell-Refactorings generisch zu definieren. Somit können diese nicht wiederverwendet und ähnlich auf andere Metamodelle übertragen, sondern müssen erneut spezifiziert werden.

Demgegenüber wird der Modell-Refactoring-Designer bei der Erstellung von Vor- und Nachbedingungen sehr gut unterstützt. Diese werden generiert und müssen dann manuell angepasst werden. Dieser Prozess verläuft relativ intuitiv, jedoch wird dabei, entgegen der Behauptung der Entwickler des *Operation Recorder*, tieferes Wissen über das Metamodell vorausgesetzt, da die Bezeichnungen der Elemente in den Bedingungen den Namen der Elemente im zugehörigen Metamodell entsprechen (siehe *Preconditions* und *Postconditions* in Abbildung 5.10).

Tabelle 5.3: Bewertung des *Operation Recorder*-Ansatzes

Bezeichnung		**Bemerkungen**
1) Generizität & Wiederverwendbarkeit	-	nur sprachspezifische Modell-Refactorings möglich
2) Kategorien	+	alle Kategorien werden unterstützt
3) Sprachspezifika	+	da nur sprachspezifische Modell-Refactorings definiert werden können, werden Spezifika sehr gut unterstützt
4) vertikale & horizontale Konsistenz	-	nur Modell-lokale Aktualisierung
5) Beibehaltung der Semantik	-	keine Angaben
6) Vor- und Nachbedingungen	+	werden generiert und können angepasst werden; danach werden Bedingungen in OCL-Code transformiert
7) Atomarität	+	Modell-Refactoring wird vollständig oder gar nicht durchgeführt
8) Umkehrbarkeit	o	theoretisch durch Nutzung eines *Diff Model* möglich, aber noch nicht implementiert
9) Empfehlung von Modell-Refactorings	-	keine Angaben
10) Interoperabilität	+	MOF-Konformität ist die einzige Anforderung an die Metamodelle
11) Erweiterbarkeit	o	aktuell nur Modell-Refactorings für Teilmenge der UML (Klassen und Zustandsmaschinen) und Ecore unterstützt
12) Benutzbarkeit & Integrationsfähigkeit	o	Benutzerführung etwas überladen und Durchführung eines Modell-Refactorings anders als von Eclipse bekannt, also wenig intuitiv
13) Durchführungsgeschwindigkeit	+	erwartungsgemäß performant
14) Unterstützung von Standards	+	MOF, OCL

Als einziger Ansatz unterstützt der *Operation Recorder*, zumindest theoretisch, die Umkehrung von Modell-Refactorings. Der Grund dafür ist die Verwendung des *Diff Model*. Da der *Operation Recorder* als Erweiterung der IDE Eclipse umgesetzt wurde, ist es möglich, aus einem *Diff Model* die zu einem Modell-Refactoring umgekehrte Transformation zu erzeugen [Eys08].

Der *Operation Recorder*-Ansatz zielt darauf ab, Modell-Refactorings auf der Ebene der Metamodell-Instanzen zu spezifizieren. Dadurch wird der Nutzer der Modell-Refactorings intensiver in deren Entwurf eingebunden. Um das erzeugte Modell-Refactoring dann auf verschiedene Modelle anwenden zu können, wird es auf die Ebene des jeweiligen Metamodells übertragen. Da sich dieses wieder auf MOF-Ebene M2 befindet, leidet dieser Ansatz, ebenso wie *EMF Refactor*, an der mangelnden Generizität. Die Generierung von OCL-Ausdrücken hingegen ist sehr vorteilhaft, da damit Bedingungen für Modelle formuliert werden können, die selbst keine Veränderungen im Modell hervorrufen [OMG06b].

5.4 Fazit

In diesem Kapitel wurden stellvertretend drei Ansätze zur Spezifikation von Modell-Refactorings vorgestellt. Diese unterscheiden sich darin, auf welcher MOF-Ebene die Spezifikation durchgeführt wird. Allen gemein ist, dass ihr Konzept mit EMF umgesetzt wurde, welches derzeit den Standard zur Metamodellierung darstellt, da es EMOF erfolgreich implementiert. Diese Gemeinsamkeit geht einher mit dem in Kapitel 2.4 vorgestellten Framework EMFText, welches als Validierungs-Umgebung benutzt werden wird. Des Weiteren wird von allen Ansätzen lediglich MOF-Konformität der Metamodelle gefordert, da nur so Interoperabilität garantiert werden kann. Demnach steht außer Frage, dass in dem zu entwickelnden Konzept das EMF als Umsetzung des EMOF-Standards gewählt werden wird. Weiterhin bietet keine Implementierung der vorgestellten Ansätze nur eine fixe Menge von Modell-Refactorings an. Dadurch ist es sowohl für Anwender als auch für Designer möglich, ihre Basis an Modell-Refactorings stets zu erweitern und an eigene Bedürfnisse anzupassen.

Die wichtigste Anforderung, die Generizität, ist hingegen in keinem der vorgestellten Ansätze zufriedenstellend. Mit dem Triskell-Ansatz ist es zwar möglich, Modell-Refactorings generisch zu spezifizieren, jedoch ist ihre Lösung zu statisch. Wie in Abschnitt 5.1 schon bewertet wurde, kann das GenericMT immer nur einmal auf ein Metamodell adaptiert werden. Somit ist es nicht möglich, dasselbe Modell-Refactoring mehrmals in einem Metamodell zu aktivieren. Dadurch ist nicht exakt kontrollierbar, welche Strukturen im Kontext eines definierten Modell-Refactorings konkret refaktoriert werden sollen. Dieser Mangel an Kontrolle wird mit *EMF Refactor* und dem *Operation Recorder* überwunden, da mit diesen Ansätzen Modell-Refactorings immer in Abhängigkeit von einem Metamodell definiert werden können. Mit beiden Ansätzen können zwar die zu refaktorierenden Strukturen genau bestimmt werden, jedoch fehlt hier die Generizität gänzlich. Das Resultat dessen ist, dass Modell-Refactorings nicht wiederverwendet werden können. Aus diesen Problemen resultiert, dass im zu entwickelnden Konzept eine Brücke zwischen der Generizität des Triskell-Ansatzes und der exakten Kontrolle der zu refaktorierenden Strukturen der beiden anderen Ansätze geschlagen werden muss. Es gilt demnach, beide Vorteile, die zugleich Nachteil des jeweils anderen sind, zu kombinieren.

Des Weiteren verfolgt keiner der vorgestellten Ansätze die vertikale und horizontale Konsistenz. Die automatisierte Umstrukturierung erfolgt jeweils nur in dem Modell, in dem das Modell-Refactoring aufgerufen wird, und propagiert die Änderungen nicht in solche Modelle, die Referenzen zu refaktorierten Elementen besitzen. Diese Anforderung ist jedoch sehr wichtig, da die Modell-Basis des Anwenders sonst in einen inkonsistenten Zustand überführt wird. Das manuelle Anpassen der referenzierenden Modelle ist jedoch fehlerbehaftet und kann bei vielen Referenzen ein beträchtliches Ausmaß annehmen.

Außerdem hat die Beibehaltung der Semantik bei allen Ansätzen zu wenig Berücksichtigung gefunden. Sie wird in dieser Arbeit zwar als ein relatives Maß angenommen, muss jedoch im Konzept diskutiert werden.

Alle weiteren bisher noch nicht erwähnten Anforderungen, die negativ bewertet wurden, müssen in der Implementierung realisiert werden. Beispielsweise hängt es von der Art der Umsetzung ab, ob ein Modell-Refactoring nur atomar ausgeführt oder es wider-

rufen werden kann.

Damit ist die Analyse existierender Ansätze abgeschlossen und Vor- und Nachteile jedes einzelnen konnten aufgezeigt werden. Des Weiteren wurden Hinweise für das Konzept dahingehend gegeben, welche Lösungen aufgegriffen und überarbeitet werden sollten. Im nachfolgenden Kapitel wird nun, aufbauend auf den hier gewonnenen Erkenntnissen, das zielführende Konzept vorgestellt.

6 Konzept des generischen Modell-Refactorings

Im vorangegangen Kapitel wurden existierende Konzepte zur Spezifikation von Modell-Refactorings bezüglich der in Kapitel 4 aufgestellten Anforderungen analysiert und bewertet. Die Analyse zeigte auf, dass keiner der vorgestellten Ansätze die Anforderungen zufriedenstellend erfüllt. Deshalb dienen die dort gewonnenen Erkenntnisse in diesem Kapitel als Grundlage für die Konzeption eines Ansatzes zum generischen Modell-Refactoring.

Wie in Kapitel 5.4 schon erläutert, ist es nicht wünschenswert, Modell-Refactorings auf dem MOF-Level M2 zu spezifizieren. Jedoch birgt auch die statische Definition auf M3-Ebene zu wenig Flexibilität und weist einen Mangel an Kontrolle über die zu refaktorierenden Strukturen auf. Das GenericMT des Triskell-Ansatzes stellt ein Meta-Metamodell dar, auf das alle Ziel-Metamodelle adaptiert werden müssen. Steht die Adaption einmal fest, kann der Kontext einer adaptierten Metaklasse in einem konkreten Modell-Refactoring nicht mehr geändert werden.

Als Ursache dieser Eigenschaft des Triskell-Ansatzes ist die Isolation des über allen Ziel-Metamodellen liegenden GenericMT anzusehen. Es stellt ein Klassenmodell dar, welches die enthaltenen Metaklassen isoliert definiert und damit aber nicht die Art und Weise reflektiert, wie Menschen über Objekte nachdenken [RWL96, S. 72]. Man möchte Strukturen zwischen Elementen modellieren, die angeben, wie diese in einem bestimmten Kontext interagieren. Eine Klasse, beziehungsweise Metaklasse, beschreibt jedoch nur die Fähigkeiten eines Elementes. Betrachtet man dieses Element jedoch im Kontext eines konkreten Modell-Refactorings, fällt auf, dass das Zusammenwirken mit anderen Elementen in einem anderen Kontext ganz verschieden sein kann. Diese Fokussierung auf den Kontext unterschiedlicher Modell-Refactorings ist mit dem Triskell-Ansatz nicht möglich, da diese auf Basis eines statischen Meta-Metamodells definiert werden. Das GenericMT definiert demnach eine Art physische Anforderung an die Metaklassen der Ziel-Metamodelle.

Betrachtet man Modell-Refactorings jedoch aus einer logischen Sicht, so leuchtet ein, dass ein und dieselbe Metaklasse in dem Kontext eines anderen Modell-Refactorings eine komplett andere Rolle spielen kann. Das bedeutet, dass jedes Modell-Refactoring in einem unterschiedlichen Kontext zu sehen ist und die Metaklassen, deren Instanzen am Modell-Refactoring beteiligt sind, in jedem Kontext eine andere Rolle spielen können. Daraus folgt wiederum, dass das Zusammenspiel der Teilnehmer eines Modell-Refactorings in einem Rollen-Modell abgebildet werden kann, denn Rollen-Modelle fokussieren nicht die Fähigkeiten eines Elementes, sondern die Position und Verantwortlichkeiten innerhalb der gesamten Struktur sowie das Zusammenwirken zwischen ih-

nen [RWL96, S. 72].

Die Modellierung mit Rollen wurde erstmals von Reenskaug et al. in [RWL96] erläutert und später von Riehle und Gross in [RG98] aufgegriffen und speziell auf den Entwurf von Frameworks übertragen und erweitert. Da Rollen-Modellierung bedeutet, Elemente immer in einem bestimmten Kontext zu sehen, und diese Eigenschaft sehr gut auf die Teilnehmer eines Modell-Refactorings zutrifft, werden Rollen im vorliegenden Konzept benutzt, um Modell-Refactorings zu spezifizieren. Auf Modell-Refactorings übertragen bedeutet dies, dass die Aufgabe eines an Modell-Refactorings teilnehmenden Elementes immer eine andere ist. Das heißt, dass Metaklassen in verschiedenen Modell-Refactorings unterschiedliche Rollen spielen können. Im Triskell-Ansatz spielt eine Metaklasse immer die Rolle, die von der Klasse vorgegeben ist, auf die sie adaptiert wurde. Bei den Ansätzen von *EMF Refactor* und des *Operation Recorder* stellen die in einem konkreten Modell-Refactoring benutzten Metaklassen implizit Rollen dar. Je mehr Modell-Refactorings demnach spezifiziert werden, desto komplexer werden die Kollaborationen, die von einer Metaklasse in verschiedenen Kontexten erfüllt werden müssen. Klassenmodelle können diese Komplexität jedoch nicht mehr bewältigen [RG98]. Demzufolge führt der Triskell-Ansatz für die flexible und doch exakte Spezifikation nicht zum Ziel, weshalb dafür Rollen-Modelle benutzt werden. Im folgenden Abschnitt wird eine gemeinsame Grundlage bezüglich der Rollen-Modellierung geschaffen und erläutert.

6.1 Modellierung mit Rollen nach Reenskaug, Riehle und Gross

Wie einleitend schon erwähnt, sprachen Reenskaug et al. in [RWL96] erstmals von dem Konzept der *Rolle*. Riehle und Gross haben dann in [RG98] ein informelles Metamodell zur Rollen-Modellierung angegeben und argumentiert, dass Rollen-Modelle Muster der Kollaboration von Objekten beschreiben, die bestimmte Bedingungen an die eigentlichen Objekte stellen. Ein Rollen-Modell kann dann auf ein Klassen-Modell abgebildet werden, wodurch vor der Instanziierung der Klassen schon automatisch geprüft werden kann, ob die modellierten Bedingungen des Rollen-Modells überhaupt auf die Klassen zutreffen. Riehle und Gross sprechen im Gegensatz zu Reenskaug von Rollen-Typen, mit denen die Sicht eines Objekts auf ein anderes definiert wird. Ist ein Objekt konform zu einem Rollen-Typ, so agiert dieses gemäß der Rollen-Typ-Spezifikation [RG98]. Anders ausgedrückt, spielt ein Objekt die Rolle, die von einem Rollen-Typ spezifiziert wird. Der Terminus *Rollen-Typ* wurde allerdings nur eingeführt, um den Weg dafür zu bereiten, den Typ einer Rolle auch formal mit einem angemessenen Mechanismus zu beschreiben. Da eine Formalisierung von Rollen-Typen hier aber nicht vorgenommen wird und auch nicht notwendig ist, da das Konzept der Rolle trotzdem vermittelt werden kann, werden die Termini *Rolle* und *Rollen-Typ* in dieser Arbeit gleichwertig benutzt.

Als Beispiel stelle man sich ein Framework für Editoren zum Zeichnen von zusammengesetzten Figuren vor. Darin befindet sich eine Klasse *Figure*, die Superklasse von allen Klassen der verschiedenen grafischen Objekte (wie beispielsweise von *Rectangle* oder *Circle*) ist. Neben anderen Rollen können Instanzen der Unterklassen von *Figure* zum Beispiel die Rollen *Figure*, *Child* oder *Subject* spielen. Die Klasse mit den möglichen

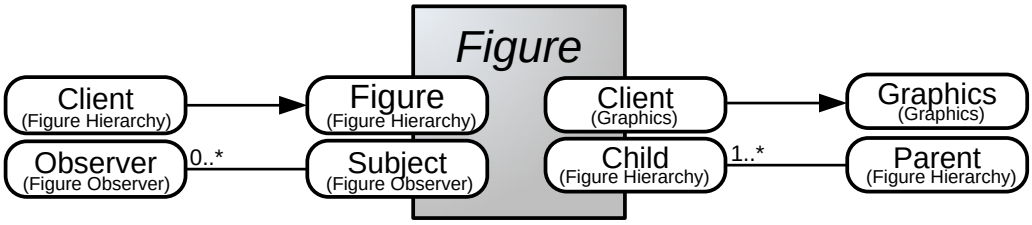


Abbildung 6.1: Die Klasse *Figure* und ihre Rollen, angelehnt an [RG98]

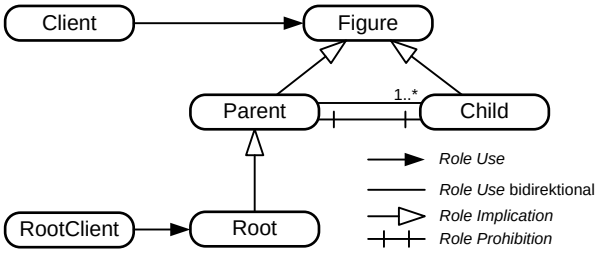


Abbildung 6.2: Rollen-Modell *Figure Hierarchy*, angelehnt an [RG98]

Rollen ist in Abbildung 6.1 dargestellt. Ein Rechteck mit abgerundeten Ecken kennzeichnet in der Abbildung eine Rolle, wobei es deren Namen beinhaltet, sowie den Namen des zugehörigen Rollen-Modells in Klammern. Letzterer kann als eine Art Namensraum angesehen werden und dient zur eindeutigen Unterscheidung[1]. Die Rolle *Figure* beschreibt hier die regulären Zeichenfunktionen eines grafischen Gebildes, wie zum Beispiel das Bilden einer Schnittmenge mit einer anderen Figur. *Child* steht für die Funktionalitäten einer Figur, Kind-Objekt in einer Objekt-Hierarchie zu sein. Wenn beispielsweise ein Kreis in ein Rechteck gezeichnet wird, dann ist der Kreis ein Kind-Element des Rechtecks. Die Rolle *Subject* spiegelt die Funktionalitäten einer Figur wider, um von anderen beobachtet werden zu können. Ändert sich zum Beispiel der Zustand einer Figur, können Beobachter diesbezüglich benachrichtigt werden. Diese Rolle ist Teil des *Design Patterns Observer* [GHJV04, S. 287].

Damit wurde gezeigt, wie Rollen auf Klassen abgebildet werden. Als nächstes muss geklärt werden, wie das Zusammenwirken in einem Rollen-Modell definiert wird. Nach Riehle und Gross ist ein Rollen-Modell die Beschreibung von Objekt-Kollaborationen mittels Rollen. Eine Rolle spezifiziert darin das Verhalten eines Objekts unter Beachtung der Intention des Rollen-Modells [RG98]. Dadurch werden Bedingungen für eine Menge von Objekt-Kollaborationen definiert, die in Abbildung 6.2 für das Rollen-Modell *Figure Hierarchy* dargestellt sind.

Abgebildet sind die Kollaborationen zwischen Rollen, die von Figuren gespielt werden, um eine Objekt-Hierarchie aufzubauen. Beispielsweise arbeitet ein Objekt, welches die

[1] Wenn nötig, wird zur eindeutigen Unterscheidung gleich benannter Rollen in unterschiedlichen Rollen-Modellen folgendes Muster zur Namensgebung benutzt: `Rollen-Modell.Rolle`

Rolle *Parent* spielt, mit mehreren *Child*-Spielern zusammen. Diese Verbindung wird mit der bidirektionalen Kollaboration *Role Use* ausgedrückt. Außerdem darf ein *Child*-Spieler nicht gleichzeitig die Rolle *Parent* spielen, weshalb zwischen beiden zusätzlich die Kollaboration *Role Prohibition* modelliert wurde. Der Grund dafür ist, dass ein *Parent* nicht im selben Kontext *Child* sein darf, insbesondere nicht sein eigenes *Child*. Jedoch ist dies in unterschiedlichen Kontexten durchaus gestattet. Das heißt, um den Aufbau einer Figur-Hierarchie zu ermöglichen und nicht nur eine flache Struktur zu gestatten, kann ein Spieler der Rolle *Player* sehr wohl auch *Child* spielen, wenn beispielsweise in einem anderen Kontext alle Kind-Elemente seines Eltern-Elements ermittelt werden sollen. Das gleichzeitige Spielen beider Rollen im selben Kontext wird mit der *Role Prohibition* allerdings ausgeschlossen [RG98].

Spieler dieser beiden Rollen müssen außerdem die Rolle *Figure* spielen. Diese Kollaboration wird durch *Role Implication* ausgedrückt, deren Verbindung an einen Vererbungs-Pfeil aus der Klassenmodellierung erinnert. *Role Implication* bedeutet jedoch in der Rollen-Modellierung, dass ein Objekt, welches eine Rolle A (*Root*) spielt, auch in der Lage sein muss, die Rolle B (*Parent*) zu spielen. Die Rolle *Root* steht in dem Beispiel aus Abbildung 6.2 für das oberste Element der Hierarchie, dem spezielle Funktionalitäten zukommen sollen. Des Weiteren existieren in diesem Rollen-Modell zwei verschiedene *Client*-Rollen. Beide werden von Objekten gespielt, die den Zugriff auf einen Spieler entweder von *Figure* oder von *Root* sicherstellen. Dies bedeutet, dass beide *Client*-Rollen jeweils eine weitere Rolle bedingen, was mit der unidirektionalen Kollaboration *Role Use* dargestellt wird.

Damit wurde eine Einführung in die Rollen-Modellierung nach Reenskaug et al., Riehle und Gross gegeben. Nun gilt es die Frage zu beantworten, wie generisches Modell-Refactoring davon profitieren kann. Im nachfolgenden Abschnitt wird dieser Frage auf den Grund gegangen und als Antwort das Konzept zur generischen Spezifikation von Modell-Refactorings gegeben.

6.2 Modellierung generischer Modell-Refactorings mit Rollen

Wie zuvor schon erläutert, sind statische Klassendiagramme nicht mächtig genug, um Kollaborationen von Elementen in verschiedenen Kontexten zu beschreiben. Als Kontext sollen hier die unterschiedlichen Modell-Refactorings angesehen werden. Ein Modell-Element spielt demnach in jedem Modell-Refactoring eine andere Rolle. Des Weiteren sind an einem Modell-Refactoring mehrere verschiedene Elemente beteiligt, die jeweils auch wieder eine Rolle spielen. So leuchtet ein, dass die Teilnehmer von Modell-Refactorings durch Rollen-Modelle zu beschreiben sind. Hier profitiert man von der Tatsache, dass mit Rollen-Modellen logische Kollaborations-Muster angegeben werden können, die auf beliebige Klassen abbildbar sind [RG98]. In Bezug auf Modell-Refactoring ist nun der Schritt zur Generizität nicht mehr weit. Definiert man für jedes Modell-Refactoring ein eigenes Rollen-Modell der teilnehmenden Elemente, braucht dieses dann nur noch auf die entsprechenden Metaklassen der Ziel-Metamodelle abgebildet zu werden. Somit wird nicht statisch adaptiert wie bei Triskell, sondern es werden stets kontextabhängig

Rollen auf Metaklassen abgebildet. Dadurch erreicht man außerdem SoC im Sinne der Modell-Refactorings. Des Weiteren kann so das Rollen-Modell eines Modell-Refactorings für beliebige Metamodelle wiederverwendet werden und man ist der generischen Spezifikation schon sehr nahe. Nun muss es nur noch ermöglicht werden, die durchzuführenden Schritte eines Modell-Refactorings nicht in Abhängigkeit vom Ziel-Metamodell, sondern lediglich auf Basis der dafür modellierten Rollen zu spezifizieren.

Um diese Vorüberlegungen abzuschließen, wird festgehalten, dass folgende drei Bestandteile benötigt werden, um Modell-Refactorings generisch zu spezifizieren:

1. eine Möglichkeit der Definition von Rollen und deren Kollaborationen
2. eine Technik, mit der man die durchzuführenden Schritte eines Modell-Refactorings auf Basis der Rollen angeben kann
3. ein Mechanismus zum Abbilden der Rollen auf das gewünschte Ziel-Metamodell

Das hier entwickelte Konzept realisiert diese drei benötigten Komponenten. Als erstes wird dazu im folgenden Abschnitt ein Metamodell vorgestellt, mit dessen Hilfe Rollen und Kollaborationen definiert werden können.

6.2.1 Metamodell zur Definition von Kollaborationen zwischen Rollen

Um Rollen und die Kollaborationen und Bedingungen zwischen ihnen modellieren zu können, wird ein formales Metamodell benötigt, welches die Struktur eines Rollen-Modells vorgibt. Zu diesem Zweck wurde das in Abbildung 6.3 dargestellte Metamodell entwickelt, das es zum einen ermöglicht, Rollen-Modelle zu erstellen, und zum anderen, schon kleine für Modell-Refactorings wichtige Details festzulegen. Das Wurzel-

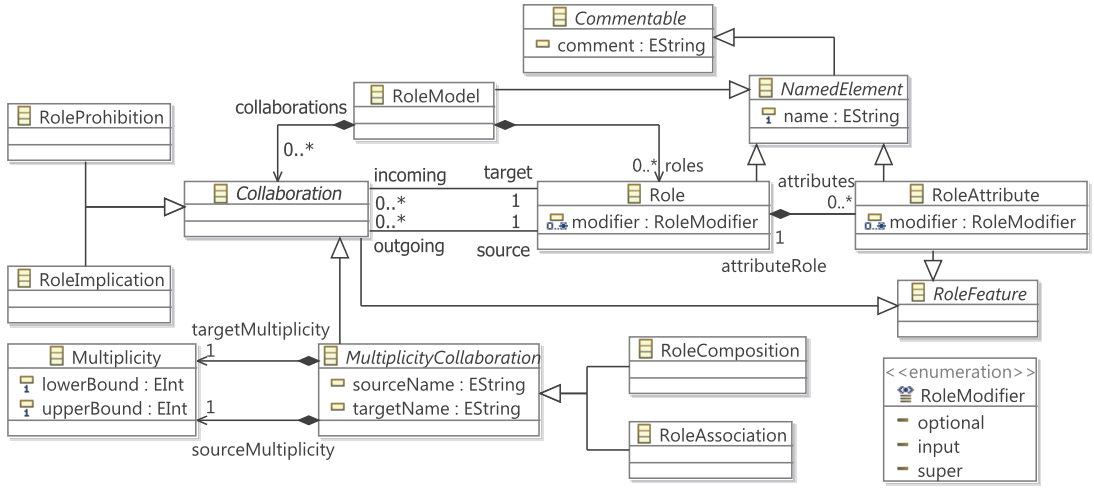


Abbildung 6.3: Metamodell zur Definition von Rollen, die von Elementen in einem Modell-Refactoring gespielt werden

element dieses Metamodells ist `RoleModel`. Diese Metaklasse ist eine Unterklasse von `NamedElement`. Diese Beziehung drückt aus, dass für ein *Role Model* stets ein Name angegeben werden muss. Dieser sollte das Modell-Refactoring aussagekräftig benennen. Des Weiteren ist jedes `NamedElement` Unterklasse von `Commentable`, mit dessen Attribut `comment` ein Kommentar zur näheren Erläuterung angegeben werden kann. Weitere Vererbungsbeziehungen zu `NamedElement` werden nicht mehr explizit erwähnt, da sie immer die gerade erläuterten Bedeutungen der Namensgebung und der damit verbundenen Kommentierung innehaben.

Ein *Role Model* besitzt zwei Kompositionsbeziehungen. Zum einen kann es beliebig viele Rollen enthalten, was mit der Metaklasse `Role` modelliert wurde. Für eine Rolle können mehrere `modifier` disjunkt angegeben werden. Diese dienen dazu, für das mit dem *Role Model* zu definierende Modell-Refactoring wichtige Eigenschaften festzulegen. Zu diesem Zweck enthält die Aufzählung `RoleModifier` folgende drei Elemente, die bei Verwendung die nachstehende Bedeutung haben.

input Zur Festlegung, dass eine Rolle als Eingabe für ein Modell-Refactoring dient.

super Besitzt eine Rolle diesen `RoleModifier`, wird ausgedrückt, dass auch Superklassen einer Metaklasse diese Rolle spielen dürfen.

optional Damit kann angegeben werden, dass eine Rolle von geringer Bedeutung ist und nicht zwingend einen Spieler im Ziel-Metamodell benötigt.

Des Weiteren kann ein *Role Model* mehrere Attribute besitzen, was mit der Kompositionsbeziehung zur Metaklasse `RoleAttribute` modelliert wurde. Mit einem Rollen-Attribut kann ausgedrückt werden, dass eine Rolle eine Eigenschaft besitzt, die sich im Kontext eines Modell-Refactorings ändern kann. Beispielsweise ändert sich bei *Rename Method* der Name der Methode. Für ein Rollen-Attribut können wiederum `optional` und `input` als `RoleModifier` gesetzt werden, womit einem Attribut die gleiche Bedeutung wie oben beschrieben zukommt. Außerdem ist `RoleAttribute` eine der beiden Unterklassen von `RoleFeature`. Diese abstrakte Metaklasse wurde aus Gründen der Wiederverwendung von Code in der Implementierung eingeführt und hat konzeptionell für das *Role Model* keine weitere Bedeutung. Darauf wird im Kapitel 7.3 noch einmal eingegangen.

Die andere Kompositionsbeziehung, die von einem *Role Model* ausgeht, ist die zur Metaklasse `Collaboration`. Diese ist neben *Role* das zweite wichtige Konzept, um Rollen und ihre Kollaborationen zu modellieren. Wie in Abbildung 6.3 zu sehen, existieren außerdem zwei bidirektionale Beziehungen zu einer Rolle. Dabei muss eine `Collaboration` jeweils genau eine Rolle als `source` und `target` referenzieren, wodurch eine Kollaborations-Beziehung zwischen zwei Rollen modelliert wird. Diese Relationen zwischen den beiden Metaklassen `Role` und `Collaboration` sind bidirektional, damit die Rollen selbst auch ihre eingehenden und ausgehenden Kollaborationen kennen.

Da die Metaklasse `Collaboration` abstrakt ist, werden Unterklassen benötigt, um konkrete Kollaborationen zu modellieren. Dazu sei an dieser Stelle nochmals an die Abbildung 6.2 des Kapitels 6.1 erinnert. Die dort dargestellten Kollaborationen *Role Implication* und *Role Prohibition* stehen exakt für die entsprechenden Metaklassen aus

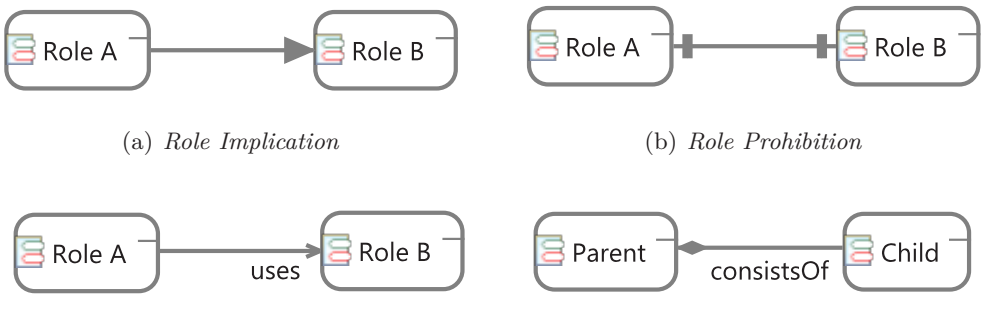
Abbildung 6.3. Mit einer `RoleProhibition` wird demnach ausgedrückt, dass sich zwei Rollen gegenseitig ausschließen, also keine Metaklasse des Ziel-Metamodells im Kontext eines Modell-Refactorings beide Rollen spielen darf. Im Gegensatz dazu gibt eine `RoleImplication` an, dass eine Rolle auch eine andere voraussetzt. Das heißt, dass die Metaklasse, die diejenige Rolle spielt, von der die *RoleImplication* ausgeht, auch gleichzeitig die andere Rolle spielen muss.

Neben den beiden zuvor erläuterten Kollaborationen existieren zwei weitere, die jeweils von der abstrakten Metaklasse `MultiplicityCollaboration` erben. Diese wird dafür verwendet, neben der Modellierung der Kollaboration zwischen zwei Rollen auch Multiplizitäten anzugeben, wie auch schon in Abbildung 6.2 zwischen *Parent* und *Child* zu sehen war. Zu diesem Zweck besitzt eine *Multiplicity Collaboration* für Quelle und Ziel jeweils eine Kompositionsbeziehung zur Metaklasse `Multiplicity`. Damit müssen dann die obere und die untere Grenze der zusammenwirkenden Rollen angegeben werden. Außerdem sind für eine *Multiplicity Collaboration* die Namen der Kollaborations-Enden anzugeben. Somit kann über eine derartige Beziehung mittels des Namens zur jeweiligen Quell- oder Ziel-Rolle navigiert werden. In der Art seiner Unterklassen von `MultiplicityCollaboration` unterscheidet sich jedoch dieses Metamodell von dem von Riehle und Gross aus Abschnitt 6.1. In dem hier vorgestellten wird statt unidirektionalem und bidirektionalem *Role Use* nur die Metaklasse `RoleAssociation` eingeführt. Damit wird modelliert, dass eine Rolle eine andere kennt und mit dieser direkt zusammenwirkt. Wird die umgekehrte Kollaborations-Richtung auch benötigt, so ist eine entgegengesetzte *Role Association* zu modellieren.

Die letzte Kollaborations-Art ist die `RoleComposition`. Diese war im Metamodell von Riehle und Gross gar nicht vorgesehen, stellt hier konzeptionell jedoch eine wichtige Erweiterung dar. War es vorher nicht möglich, Eltern-Kind-Kollaborationen zwischen Rollen zu modellieren, ist dies in diesem Konzept mit einer *Role Composition* möglich. Möchte man beispielsweise ausdrücken, dass eine Rolle nur existieren kann, wenn eine andere vorhanden ist, so entspricht diese Kollaboration einer Art Komposition, wie sie schon aus UML-Klassenmodellen bekannt ist. In einer anderen Sichtweise kann diese Kollaboration als eine komposite Rolle beschrieben werden, die aus unabhängigen Unter-Rollen zusammengesetzt wird, um einen gemeinsamen Zweck zu verfolgen. Diese Betrachtungsweise ähnelt sehr dem Muster der Werkzeug-Komposition der *Tools & Materials*-Metapher, wie sie von Riehle und Züllighoven in [RZ95] vorgestellt wurde.

Damit wurde das Metamodell zur Spezifikation von Rollen und Kollaborationen der teilnehmenden Elemente eines Modell-Refactorings definiert und erläutert. Um die zuvor beschriebenen Kollaborationen zu veranschaulichen, wird an dieser Stelle ein in der Implementierung umgesetzter graphischer Editor vorweggenommen. Auf diesen wird in Kapitel 7.2 nochmals näher eingegangen, seine Erwähnung bereits in diesem Kapitel erweist sich hier zum besseren Verständnis als vorteilhaft. Abbildung 6.4 zeigt dazu alle im zuvor definierten Metamodell erläuterten Rollen-Kollaborationen und wie sie im graphischen Editor umgesetzt wurden. Dabei wurde weitestgehend versucht, sich an der Notation von Riehle und Gross zu orientieren.

Mit dem in diesem Kapitel vorgestellten Metamodell ist es nun möglich, *Role Models* zu definieren. Mit Hilfe des schon erwähnten Editors kann man dies besonders komfortabel



(a) *Role Implication* (b) *Role Prohibition*

(c) *Role Association* (d) *Role Composition*

Abbildung 6.4: Mögliche Kollaborationen zwischen Rollen eines Modell-Refactorings

handhaben. Ein derartig modelliertes *Role Model* enthält dann alle Rollen, die von den an einem Modell-Refactoring teilnehmenden Modell-Elementen gespielt werden können. So ist ein *Role Model* das dynamische Pendant zum statischen GenericMT der Triskell-Gruppe. Das bedeutet, dass ein *Role Model* als temporäres Meta-Metamodell im Kontext eines generischen Modell-Refactorings betrachtet werden kann. Demzufolge ist es damit möglich, die teilnehmenden Elemente eines Modell-Refactorings sehr viel feingranularer zu definieren, als es das GenericMT tut. Außerdem können die durch die modellierten Kollaborationen repräsentierten Bedingungen (wie beispielsweise eine *Role Prohibition*) statisch im Ziel-Metamodell geprüft werden.

Die Vorstellung des hier zu definierenden Metamodells ist nun vollständig. Der erste in Kapitel 6.2 benötigte Teil wurde damit umgesetzt. Im nachfolgenden Abschnitt wird der zweite Teil konzipiert, mit dem die in einem Modell-Refactoring durchzuführenden Schritte auf Grundlage eines *Role Models* spezifiziert werden können.

6.2.2 Metamodell zur Spezifikation der Schritte eines Modell-Refactorings

Nun gilt es, die durchzuführenden Schritte eines Modell-Refactorings unabhängig von den Ziel-Metamodellen zu spezifizieren. Die Grundlage eines Modell-Refactorings ist ein vorher definiertes *Role Model*. Um die einzelnen Schritte generisch spezifizieren zu können, darf dies nur auf den im *Role Model* definierten Rollen und Kollaborationen durchführbar sein. Dieser Mechanismus wird jedoch noch von keiner der aktuell existierenden M2M-Transformationssprachen, wie beispielsweise ATL[2] oder QVT [OMG07], unterstützt. Außerdem wurden vorhandene M2M-Sprachen primär dazu entworfen, ein Modell in ein anderes zu transformieren, wobei beide unterschiedliche Metamodelle besitzen. Endogene Transformationen sind demnach nicht der primäre Verwendungszweck solcher Sprachen. Da Modell-Refactorings Transformationen in einem einzigen Modell beschreiben, sind diese auch endogen. Aus diesem Grund wird in diesem Kapitel ein Metamodell erläutert, mit dem es möglich ist, die Schritte eines Modell-Refactorings anhand

[2] `http://www.eclipse.org/atl/`

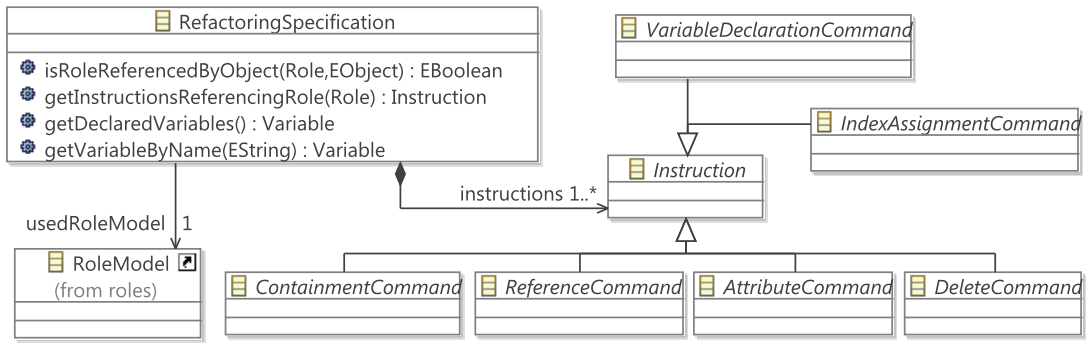


Abbildung 6.5: Grundsätzlicher Aufbau der *Refactoring Specification*

eines vorher modellierten *Role Models* zu spezifizieren. Dieses Metamodell ist gleichzeitig als endogene Modell-Transformations-Sprache zu sehen, mit der auf Basis von Rollen die durchzuführenden Schritte von Modell-Refactorings definiert werden können.

Wie in Kapitel 3.3 erläutert, konnten vier primitive Muster identifiziert werden, aus denen sich Modell-Refactorings dann zusammensetzen lassen. Diese vier Muster sind namentlich die Erzeugung, die Verschiebung, die Änderung oder die Entfernung von Elementen. Infolgedessen müssen mindestens diese Primitive im Metamodell reflektiert werden. Der Übersichtlichkeit halber werden im Folgenden immer nur Ausschnitte aus dem gesamten Metamodell zur Spezifikation der Einzelschritte abgebildet und erläutert. Die Zusammensetzung aller Teile bildet dann das vollständige Metamodell.

Abbildung 6.5 zeigt den grundsätzlichen Aufbau des Metamodells. Die Metaklasse `RefactoringSpecification` stellt für dieses Metamodell das Wurzelelement dar. Dort ist zu erkennen, dass eine *Refactoring Specification* genau ein vorher modelliertes *Role Model* referenziert. Damit wird die Verbindung zwischen diesen beiden Modellen hergestellt und die Teilschritte können anhand der im referenzierten *Role Model* definierten Rollen spezifiziert werden. Die dynamische Semantik der Metaklasse `RefactoringSpecification` wird zusätzlich durch die vier definierten Operationen festgelegt. Mit ihnen wird die Interpretation einer *Refactoring Specification* vereinfacht, worauf in Kapitel 7.3 noch näher eingegangen wird. Des Weiteren besitzt eine *Refactoring Specification* eine Kompositions-Beziehung zur abstrakten Metaklasse `Instruction`. Mit dieser Beziehung können die Transformations-Anweisungen eines Modell-Refactorings modelliert werden. Hier ist schon zu erkennen, dass die zuvor genannten primitiven Anweisungen auf dieser Ebene des Metamodells noch nicht direkt Verwendung finden. Lediglich die Metaklasse `DeleteCommand` entspricht dem Entfernen von Modell-Elementen. Dieser Teil des Metamodells wird in Abschnitt 6.2.2.4 detailliert erläutert. Da die primitiven Anweisungen des Erzeugens und des Bewegens von Modell-Elementen sehr eng verwandt sind, wurden diese hinter der abstrakten Metaklasse `ContainmentCommand` gekapselt. Diese wird in Abschnitt 6.2.2.3 näher erläutert. Das Ändern von Elementen bezieht sich in der Metamodellierung nicht nur auf Attribute, sondern, wie schon in Kapitel 3.3 ausgeführt, auch auf Referenzen. Zu diesem Zweck wurden die Metaklas-

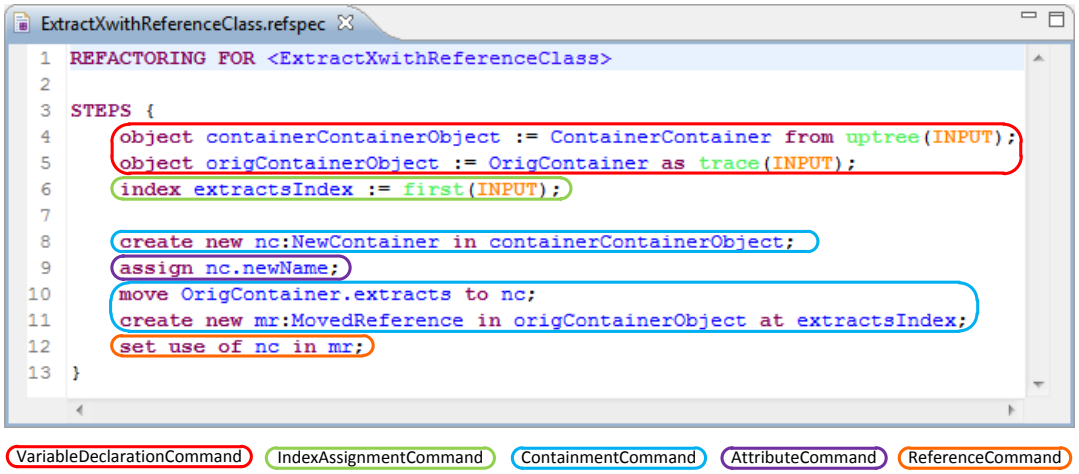


Abbildung 6.6: Beispiel-Refactoring, dargestellt im textuellen Editor

sen `ReferenceCommand` und `AttributeCommand` in das Metamodell aufgenommen. Diese werden in Abschnitt 6.2.2.5 vorgestellt.

Allen voran steht jedoch die Deklaration von Variablen. Diese werden beispielsweise benötigt, um während der Interpretation der generischen Spezifikation eines Modell-Refactorings auf das eigentliche Objekt, welches eine konkrete Rolle spielt, zugreifen zu können, oder aber um die Navigation von einer als `input` markierten Rolle zu einer anderen und den Zugriff auf diese mittels einer Variablen zu ermöglichen. Das bedeutet, dass Variablen von den meisten der möglichen Anweisungen benutzt werden können, weshalb zuerst in Abschnitt 6.2.2.1 gezeigt wird, wie der Teil zur Modellierung von Variablen im Metamodell spezifiziert wurde, für den die Metaklasse `VariableDeclarationCommand` existiert. Außerdem kann es für das Bewegen von Modell-Elementen vonnöten sein anzugeben, an welche konkrete Position sie zu verschieben sind. Dafür wurde die Metaklasse `IndexAssignmentCommand` eingeführt, die in Abschnitt 6.2.2.2 erläutert wird.

Bevor die einzelnen Teilstrukturen aus Abbildung 6.5 erläutert werden, werden hier eine Beispiel-Transformation und der textuelle Editor zur Spezifikation der Teilschritte von Modell-Refactorings vorweggenommen. Dieser Editor und auch die Transformation werden im Kapitel 7.3 detailliert erläutert. Da das in diesem Kapitel vorgestellte Metamodell sehr komplex ist, dient Abbildung 6.6 dem genaueren Verständnis der im Folgenden beschriebenen Teilstrukturen des Metamodells. In dieser Grafik sind die Anweisungen, die den abstrakten Metaklassen aus Abbildung 6.5 entsprechen, farbig hervorgehoben.

6.2.2.1 Modellieren von Variablen

Variablen sind auch in einer *Refactoring Specification* von entscheidender Bedeutung. Wie oben schon erwähnt, kann eine Rolle im *Role Model* als Eingabe-Rolle markiert werden. Damit wird ausgedrückt, dass ein Modell-Element, welches diese Rolle spielt, als Eingabe-Parameter für ein Modell-Refactoring fungiert. Alle anderen Teilnehmer,

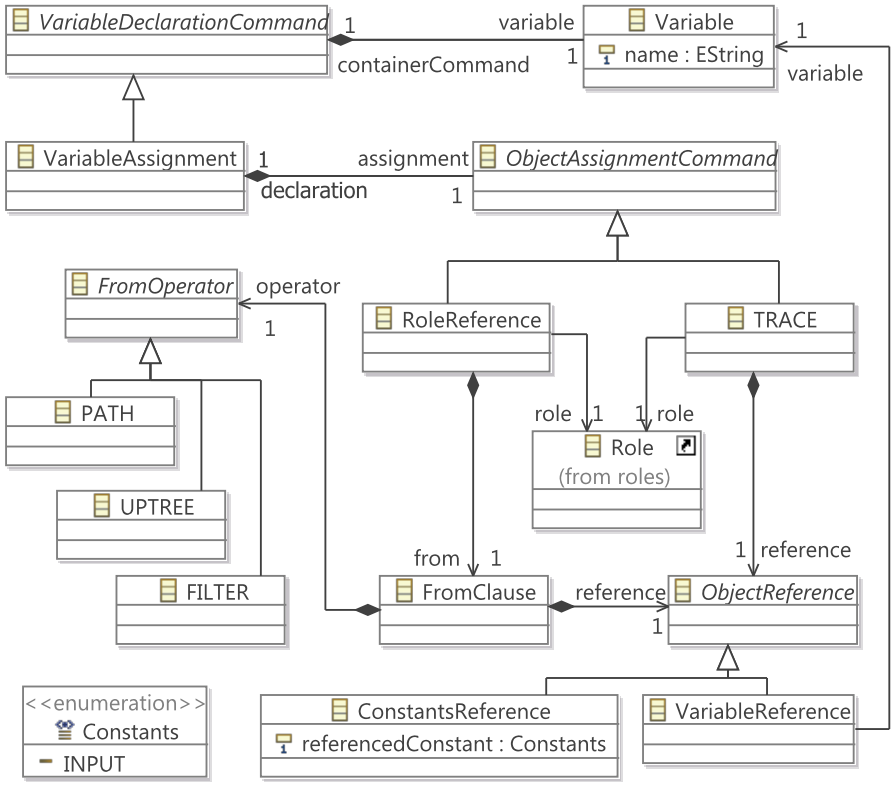


Abbildung 6.7: Modellierung von Variablen in der *Refactoring Specification*

die keine Eingabe-Rolle spielen, müssen jedoch trotzdem in der *Refactoring Specification* zugänglich sein. Das heißt, dass die eigentliche Transformation des Modell-Refactorings natürlich nicht auf den Rollen, sondern stets auf den konkreten Modell-Elementen einer Instanz des jeweiligen Ziel-Metamodells erfolgt. Demnach müssen die Teilschritte zwar anhand der modellierten Rollen spezifiziert werden, aber Platzhalter für die später zu transformierenden Modell-Elemente werden trotzdem benötigt. Diese Aufgabe übernehmen die Variablen. Mit ihnen ist es möglich, ausgehend von den Elementen, die eine Eingabe-Rolle spielen, alle konkreten Modell-Elemente, die andere Rollen spielen, auf Variablen abzubilden. Dazu dient der in Abbildung 6.7 dargestellte Ausschnitt aus dem Metamodell einer *Refactoring Specification*.

In dieser Abbildung ist zu sehen, dass von so einer Anweisung immer genau eine Variable erzeugt wird. Diese wird eindeutig durch ihren Namen identifiziert. Da `VariableDeclarationCommand` abstrakt definiert ist, wird eine konkrete Unterklasse benötigt. Um einer Variablen etwas zuzuweisen, sind in einer Zuweisung jeweils eine linke Seite, in Gestalt der Variablen, sowie eine rechte Seite, mit dem Ausdruck des gewünschten Wertes für die Variable, vonnöten. Die rechte Seite wird hier von der Metaklasse `VariableAssignment` und der unären Kompositions-Beziehung zur abstrakten

Metaklasse `ObjectAssignmentCommand` verkörpert. Zu dieser gibt es genau zwei konkrete Unterklassen, mit denen der Variablen Modell-Elemente in einer jeweils anderen Art und Weise zugewiesen werden können.

Die eine Möglichkeit ist die Nutzung der Metaklasse `TRACE`. Diese besitzt genau eine Referenz zu einer vorher modellierten Rolle. Darüber hinaus existiert eine Kompositions-Beziehung zu einer `ObjectReference`, mit der auf in einem konkreten Modell vorkommenden Elemente verwiesen werden kann. Dies kann zum einen mit der Metaklasse `VariableReference`, welche auf eine schon definierte Variable verweist, geschehen, zum anderen aber auch durch eine referenzierte Konstante aus der Aufzählung `Constants`. Diese enthält `INPUT`, um alle Elemente anzusprechen, die Eingabe-Rollen spielen. Doch was genau kann nun mit `TRACE` erreicht werden? Diese Anweisung wird dazu benutzt, um zuerst die Eltern-Elemente aller, mittels der `ObjectReference` aufgelösten, Elemente zu ermitteln. Danach werden genau diejenigen Elemente beibehalten, die Spieler der von `TRACE` referenzierten Rolle sind. Damit kann also eine Art Filter für Eltern-Elemente angewendet werden.

Die andere Möglichkeit ist die Nutzung der Metaklasse `RoleReference`. Damit wird auch, wie bei `TRACE`, eine Rolle des *Role Models* referenziert. Ausgehend von dieser Rolle können mit `RoleReference` konkrete Modell-Elemente ermittelt werden, die in unterschiedlicher Art und Weise von der referenzierten Rolle erreichbar sind. Dazu existiert die Kompositions-Beziehung zur Metaklasse `FromClause`, mit der wiederum auf schon aufgelöste Modell-Elemente in einer `ObjectReference` verwiesen wird. Des Weiteren muss ein `FromOperator` angegeben werden, mit dem die aufgelösten Modell-Elemente verschieden weiterverarbeitet werden können. Der einfachste aller Operatoren ist `FILTER`. Wird dieser Operator benutzt, so dient die referenzierte Rolle als Filter für die aufgelösten Modell-Elemente. Es bleiben dadurch also nur die Elemente übrig, die auch Spieler der Rolle sind. Der Operator `PATH` hingegen kann benutzt werden, um von den gegebenen Modell-Elementen, entlang der im *Role Model* modellierten Pfade ihrer gespielten Rollen, genau die Elemente zu erreichen, die Spieler der referenzierten Rolle sind. Damit ist es möglich, entlang modellierter Pfade zu navigieren, ohne sich direkt auf die spezifizierten Kollaborationen zwischen den Rollen beziehen zu müssen. Der dritte Operator wird von der Metaklasse `UPTREE` verkörpert. Mit diesem Operator wird zuerst für jedes der aufgelösten Elemente der Pfad bis zum Wurzelelement des Modells bestimmt. Danach wird die Schnittmenge aller Pfade bestimmt. Es darf demnach nur noch ein Pfad übrig bleiben, der in allen anderen enthalten ist. Entlang dieses Pfades wird nun das Element ermittelt, welches, von unten beginnend, als erstes die von `RoleReference` verwiesene Rolle spielt.

Zusammenfassend kann gesagt werden, dass Variablen konkrete Modell-Elemente auflösen, die im Kontext eines Modell-Refactorings die dafür modellierten Rollen spielen. Auf die definierten Variablen kann dann in anderen Anweisungen zurückgegriffen werden.

6.2.2.2 Modellieren von Indizes

Wie zuvor schon erwähnt, kann es manchmal notwendig sein anzugeben, wo ein Modell-Element beim Verschieben genau positioniert werden soll. Dafür wird mit dem hier

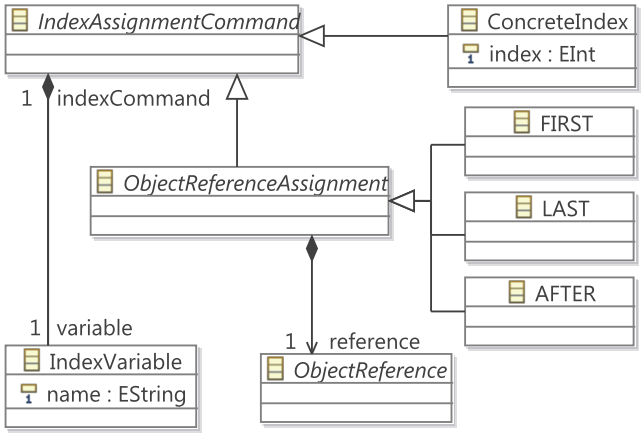


Abbildung 6.8: Modellierung von Indizes in der *Refactoring Specification*

erläuterten Metamodell der *Refactoring Specification* eine weitere Art von Variablen definiert. Dies geschieht mit dem in Abbildung 6.8 dargestellten Teil des Metamodells.

Analog zum `VariableDeclarationCommand` wird hier mit der Metaklasse `IndexAssignmentCommand` eine `IndexVariable` erzeugt. Diese wird wieder durch einen Namen eindeutig identifiziert. Um einen absoluten Index anzugeben, kann eine Instanz von `ConcreteIndex` benutzt werden, mit deren Attribut `index` eine natürliche Zahl als gewünschte Position definiert wird. Um einen Index in Abhängigkeit von schon aufgelösten Modell-Elementen zu spezifizieren, wird die Metaklasse `ObjectReferenceAssignment` verwendet, welche wiederum eine Kompositions-Beziehung zu `ObjectReference` besitzt. Diese wurde im vorangegangenen Abschnitt schon erläutert, weshalb hier nicht mehr darauf eingegangen wird. Mit Instanzen von `FIRST` beziehungsweise `LAST` kann dann der kleinste, respektive der größte, Index der mit `ObjectReference` aufgelösten Modell-Elemente ermittelt werden und der `IndexVariable` zugewiesen werden. Außerdem bestimmt `AFTER` den um Eins höheren Index des letzten mittels `ObjectReference` verwiesenen Elementes.

Mit diesem Teil des Metamodells der *Refactoring Specification* werden Indizes in Variablen abgelegt, wenn die Position von zu verschiebenden Modell-Elementen wichtig ist. Damit wurden die zwei Arten von möglichen Variablen erläutert und können nun in den nachfolgenden Abschnitten zur Definition der einzelnen Teilschritte verwendet werden.

6.2.2.3 Erzeugen und Verschieben von Elementen

Mit der Einführung der Variablen können nun die Anweisungen zur Definition der Einzelschritte eines Modell-Refactorings erläutert werden. Dazu zählen das zuvor schon erwähnte Erzeugen sowie das Verschieben von Elementen. Um dies zu realisieren, besitzt das Metamodell der *Refactoring Specification* den in Abbildung 6.9 dargestellten Ausschnitt. Wie in dieser Abbildung zu sehen, kann jede dieser beiden Anweisungen auf eine `IndexVariable` verweisen. Damit kann ein Modell-Element entweder an einer

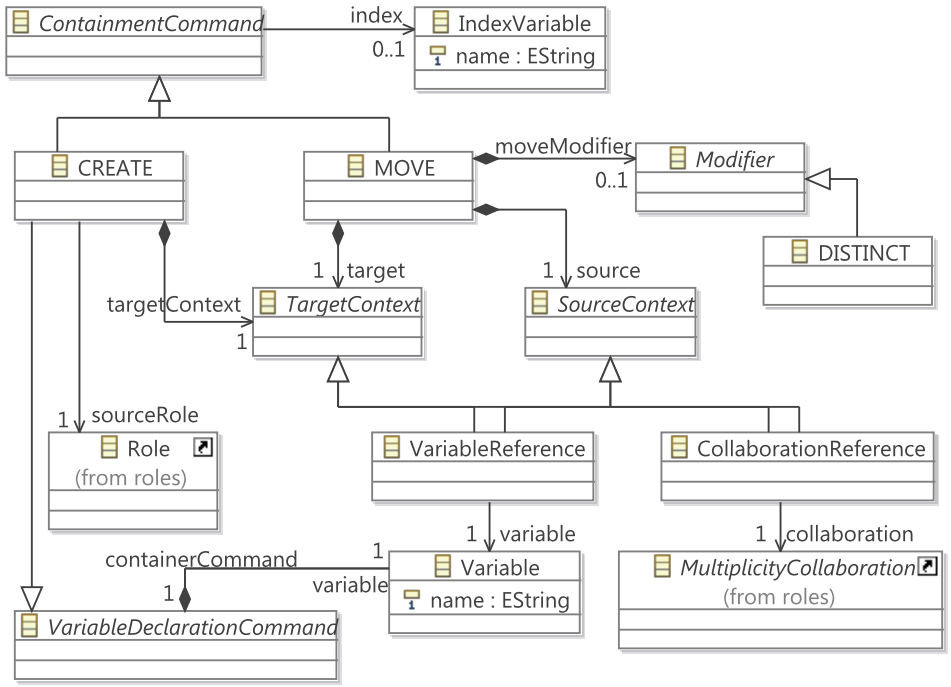


Abbildung 6.9: Erzeugen und Verschieben in der *Refactoring Specification*

bestimmten Position erzeugt oder dorthin verschoben werden.

Zur Erzeugung wird die Metaklasse `CREATE` benutzt. Diese verweist auf eine vorher definierte Rolle, womit ausgedrückt wird, dass ein neues Element der Metaklasse des Ziel-Metamodells erzeugt werden soll, die auf die referenzierte Rolle abgebildet wurde. Um später auf das zu erzeugende Modell-Element zugreifen zu können, erbt `CREATE` von `VariableDeclarationCommand`. Somit wird sichergestellt, dass für das neu erzeugte Element gleichzeitig auch eine Variable deklariert wird, auf die dann in folgenden Anweisungen zugegriffen werden kann. Außerdem besitzt `CREATE` eine Kompositions-Beziehung zur abstrakten Metaklasse `TargetContext`. Damit wird angegeben, wo genau das zu erzeugende Element erstellt werden soll, das heißt als Kind welches anderen Elementes. Dazu kann zum einen die in Abschnitt 6.2.2.1 schon erläuterte Metaklasse `VariableReference` benutzt werden, womit das in einer Variablen referenzierte Modell-Element verwendet wird, wenn dessen gespielte Rolle eine *Role Composition* zur Rolle des erzeugten Elementes hält. Zum anderen kann aber auch die Metaklasse `CollaborationReference` als Ziel der Erzeugung genutzt werden, um eine konkrete ausgehende Kollaboration einer im *Role Model* definierten Rolle anzugeben. Das bedeutet, dass das Ziel der ausgehenden *Role Composition* der damit referenzierten Rolle eine Rolle sein muss, die vom erzeugten Element gespielt wird.

Im Gegensatz dazu kann mittels `MOVE` ein schon existierendes Element an eine andere

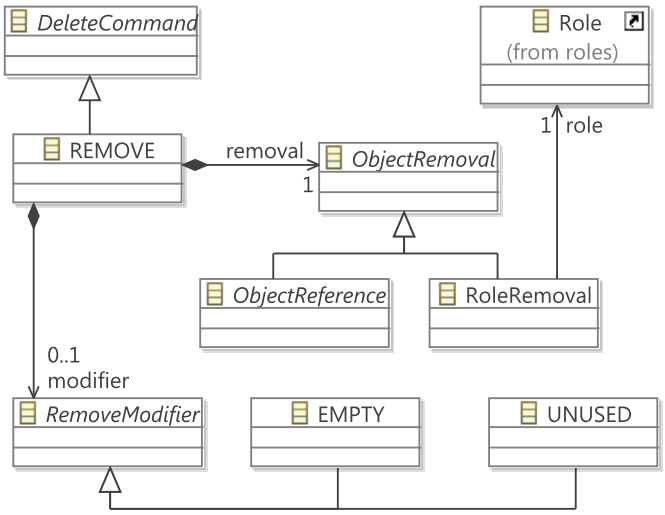


Abbildung 6.10: Entfernen von Elementen in der *Refactoring Specification*

Position verschoben werden. Dazu muss mit der abstrakten Metaklasse `SourceContext` angegeben werden, um welches Element es sich handeln soll. In Abbildung 6.9 ist zu sehen, dass sowohl `VariableReference` als auch `CollaborationReference` Unterklassen von `SourceContext` sind. Demnach können auch hier wieder Variablen oder konkrete Kollaborationen des *Role Models* referenziert werden. Als Ziel der Verschiebung dient, analog zu `CREATE`, die Metaklasse `TargetContext`, die zuvor schon erläutert wurde. `MOVE` besitzt allerdings noch eine Besonderheit. Optional ist hier ein `Modifier` anzugeben, mit dem die Verschiebung beeinflusst werden kann. So wird mit `DISTINCT` erreicht, dass aus der Menge der zu verschiebenden Elemente keine zwei gleichen erzeugt werden. Beispielsweise können in mehreren Klassen eines UML-Diagramms gleichzeitig beliebige Attribute zur Durchführung von *Pull Up Attribute* gewählt werden. In jeder einzelnen Klasse sind die Attribute einmalig. Nun kann es jedoch vorkommen, dass zwei Klassen genau dasselbe Attribut besitzen. Würde man `DISTINCT` nicht verwenden, hätte die neue Klasse zwei gleiche Attribute. In diesem Fall wird mit `DISTINCT` erreicht, dass das Attribut in der neuen Klasse nur einmal vorkommt.

Damit wurden das Erzeugen und das Verschieben von Modell-Elementen erläutert. Die Modellierung dieser Anweisungen erfolgt einzig in Abhängigkeit vom *Role Model*. Im nachfolgenden Abschnitt wird nun das Entfernen von Elementen beschrieben.

6.2.2.4 Entfernen von Elementen

Zum Spezifizieren von Anweisungen zur Löschung etwaiger Modell-Elemente dient die in Abbildung 6.10 dargestellte abstrakte Metaklasse `DeleteCommand`. Einzige Unterklasse davon ist `REMOVE`, mit deren Hilfe die mit der Metaklasse `ObjectRemoval` zu eliminierenden Elemente anzugeben sind. Auch hier müssen wieder die konkreten Modell-Elemente

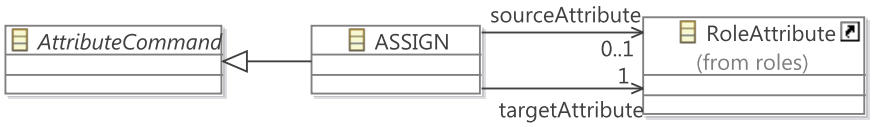


Abbildung 6.11: Ändern von Attributen in der *Refactoring Specification*

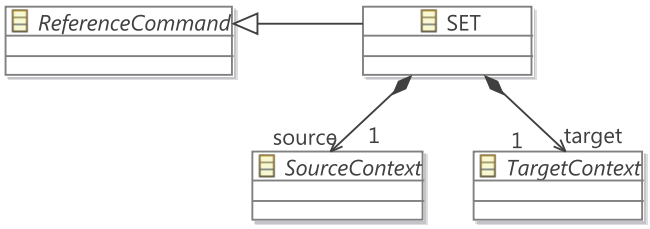


Abbildung 6.12: Ändern von Referenzen in der *Refactoring Specification*

ermittelt werden. Dazu dient zum einen die schon in Abschnitt 6.2.2.1 erläuterte Metaklasse `ObjectReference`, auf die hier nicht näher eingegangen wird. Zum anderen kann mittels der Metaklasse `RoleRemoval` eine Rolle referenziert werden. Im Ergebnis der Verwendung dieser Klasse werden die Modell-Elemente entfernt, die im Kontext des Modell-Refactorings die referenzierte Rolle spielen. Zudem kann auch hier wieder das Entfernen mit Verwendung der Metaklasse `RemoveModifier` beeinflusst werden. Sollen Elemente beispielsweise nur entfernt werden, wenn sie leer sind, also keine weiteren Unter-Elemente besitzen, so kann man `EMPTY` verwenden. `UNUSED` hingegen muss verwendet werden um auszudrücken, dass nur gelöscht werden soll, wenn die jeweiligen Modell-Elemente nicht von anderen referenziert werden.

Somit kann auf komfortable Art und Weise das Entfernen von Modell-Elementen gesteuert werden. Der Teil zum Ändern von Modell-Elementen wird nun im nachfolgenden Abschnitt beschrieben.

6.2.2.5 Änderung von Elementen

Als letztes primitives Muster muss nun noch das Ändern von Modell-Elementen realisiert werden. Dazu muss wie in Kapitel 3.3 zwischen Attributen und Referenzen in Modellen unterschieden werden. Abbildung 6.11 zeigt den Teil zur Änderung von Attributen aus dem Metamodell der *Refactoring Specification* und Abbildung 6.12 zeigt den Teil zur Änderung von Referenzen zwischen Modell-Elementen. Um Attribute zu ändern existieren genau zwei Möglichkeiten. Die Metaklasse `ASSIGN` besitzt dazu eine optionale Referenz auf ein in einem *Role Model* definiertes `RoleAttribute`. Wird an dieser Stelle ein Rollen-Attribut referenziert, so bedeutet dies, dass der Wert dieses Attributs der neue Wert des obligatorisch referenzierten Attributs sein wird. Es leuchtet ein, dass dies

nur möglich ist, wenn die Typen der Attribute, auf die die Rollen-Attribute abgebildet wurden, einander entsprechen. Wird nicht auf ein `sourceAttribute` verwiesen, so erhält das `targetAttribute` den Wert, der von dem das Modell-Refactoring durchführenden Benutzer angegeben wird. Dieser Wert muss auch im Wertebereich des abgebildeten Attributs gültig sein.

Das Ändern von Referenzen geschieht mit der Metaklasse `SET`. In Abbildung 6.12 ist zu sehen, dass diese jeweils genau eine Quelle und ein Ziel besitzt. Die dazu verwendeten Metaklassen `SourceContext` und `TargetContext` wurden in Abschnitt 6.2.2.3 schon im Detail erläutert. Es kann zusammenfassend gesagt werden, dass auch hier wieder bereits aufgelöste Modell-Elemente zugleich als Quelle und Ziel dienen. Mit `SET` kann so eine Referenz-Beziehung zwischen Modell-Elementen verändert werden. So werden entsprechend einer im *Role Model* definierten *Role Association* die Elemente der Quelle bei den Elementen des Ziels referenziert.

Damit ist die Vorstellung des Metamodells *Refactoring Specification* abgeschlossen. Mit der hier vorgestellten Konzeption wurde erreicht, dass die einzelnen Schritte eines Modell-Refactorings nur in Abhängigkeit von einem *Role Model* definiert zu werden brauchen. Damit ist die Grundlage für die geforderte Generizität gelegt, da keine Abhängigkeiten zu den Ziel-Metamodellen bestehen. Ein Modell-Refactoring kann mit den bis hierher vorgestellten Metamodellen vollständig generisch spezifiziert werden. Um es aber in konkreten Metamodellen anwenden zu können, wird im nächsten Abschnitt das Metamodell zum Abbilden von Rollen auf ein gewünschtes Ziel-Metamodell erläutert.

6.2.3 Metamodell zur Abbildung von Rollen auf ein Ziel-Metamodell

Die beiden zuvor erläuterten Metamodelle erlauben es, Modell-Refactorings generisch zu spezifizieren. Dies allein reicht allerdings noch nicht aus, um sie auch in einem konkreten Metamodell anwenden zu können. Wie in Abschnitt 6.2 in der Vorbetrachtung schon diskutiert, wird ein Mechanismus benötigt, mit dem man die in einem *Role Model* definierten Rollen nun auch auf die gewünschten Ziel-Metamodelle abbilden kann. Ein Vorteil einer solchen Abbildung ist es, dass die Bedingungen, die mittels der Kollaborationen zwischen Rollen spezifiziert wurden, dann schon statisch im Ziel-Metamodell geprüft werden können. Besteht beispielsweise zwischen zwei Rollen eine *Role Prohibition*, so kann diese Bedingung im Ziel-Metamodell validiert werden, indem ermittelt wird, auf welche Metaklassen beide Rollen abgebildet wurden, und anschließend verglichen wird, ob beide disjunkt sind.

Um sicherzustellen, dass nur MOF-konforme Ziel-Metamodelle unterstützt werden, dürfen in dem hier vorgestellten Metamodell neben den Metaklassen, die im Abschnitt 6.2.1 definiert wurden, nur Metaklassen des EMOF implementierenden Ecore referenziert werden. Abbildung 6.13 zeigt das hierfür entwickelte Metamodell. Das Wurzelelement ist die Metaklasse `RoleMappingModel`. Diese verweist auf die in Ecore enthaltene Metaklasse `EPackage`. Da Metamodelle immer in einem `EPackage` definiert werden, wird mit dieser Referenz die Verbindung zum gewünschten Ziel-Metamodell hergestellt. In einem *Role Mapping Model* können gleichzeitig mehrere *Role Models* unterschiedlicher Modell-Refactorings auf ein Metamodell abgebildet werden. Aus diesem Grund können

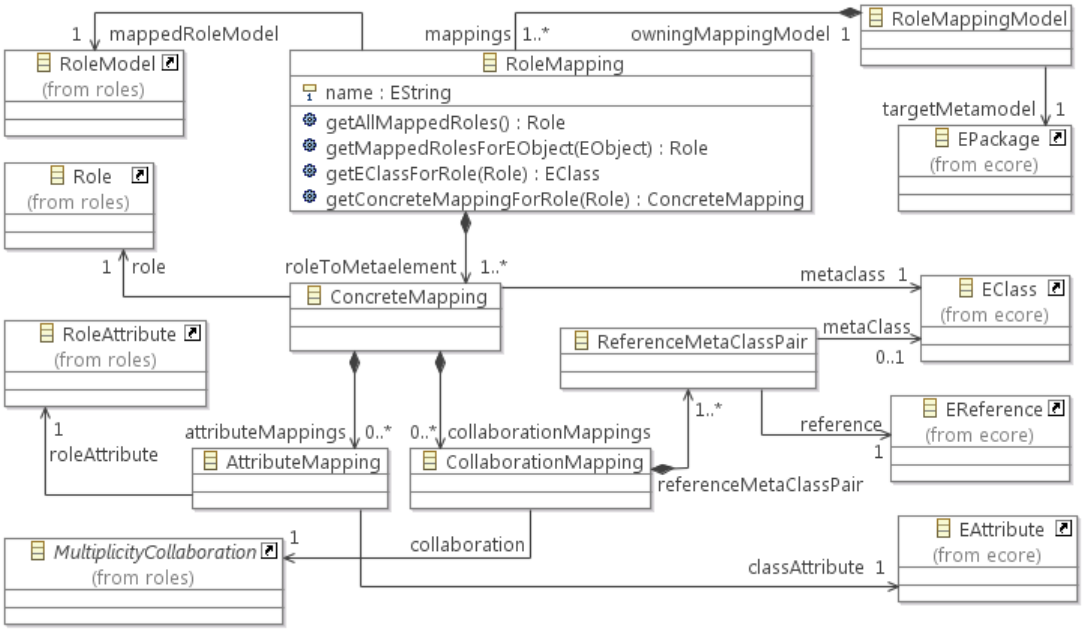


Abbildung 6.13: Metamodell zum Abbilden eines *Role Models* auf ein Ziel-Metamodell

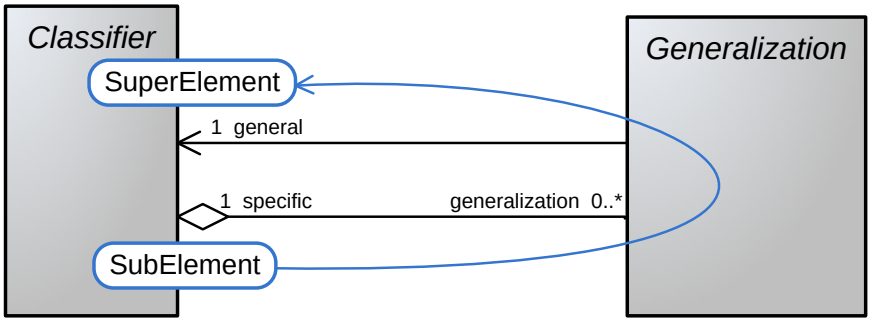
in einem *Role Mapping Model* mehrere `RoleMapping`s definiert werden. Ein `RoleMapping` dient dazu, die Rollen genau eines *Role Models* auf die Metaklassen des zuvor referenzierten Metamodells abzubilden. Aus diesem Grund muss in einem `RoleMapping` auf ein *Role Model* verwiesen werden. Außerdem besitzt die Metaklasse `RoleMapping` einen Namen, der gleichzeitig als spezifischer Name des generischen Modell-Refactorings im angegebenen Ziel-Metamodell gilt. Die im `RoleMapping` definierten Operationen dienen der Interpretation eines Modell-Refactorings im Ziel-Metamodell und werden in Abschnitt 7.3 näher erläutert. Um nun eine konkrete Rolle auf eine konkrete Metaklasse abzubilden, besitzt `RoleMapping` eine Kompositions-Beziehung zu `ConcreteMapping`. Letztere Metaklasse referenziert zu diesem Zweck genau eine Rolle aus dem *Role Model* und genau eine `EClass` aus Ecore. Damit wird der referenzierten Rolle eine Metaklasse im Ziel-Metamodell zugeordnet. Außerdem können, wie in Abbildung 6.3 gezeigt, in einem *Role Model* auch Attribute für Rollen definiert werden. Ist dies geschehen, müssen diese im `ConcreteMapping` auch auf Attribute der Metaklasse abgebildet werden. Dazu dient die Metaklasse `AttributeMapping`, mit der einem `EAttribute` aus Ecore ein `RoleAttribute` aus dem *Role Model* zugewiesen wird. Bis hierhin können nun schon Rollen und auch Rollen-Attribute auf Metaklassen und ihre Attribute abgebildet werden. Es kann jedoch vorkommen, dass von einer Metaklasse zu einer anderen mehrere Beziehungen existieren. Dies war beispielsweise in Abbildung 5.3 auf Seite 35 zu sehen. Dort besitzt die Metaklasse `Generalization` sowohl eine Referenz `general` als auch eine Referenz `specific` zur Metaklasse `Classifier`. Um derartige Mehrdeutigkeiten zu vermei-

den und eindeutig angeben zu können, welche Kollaboration welcher Beziehung zwischen Metaklassen entspricht, müssen nun noch die im *Role Model* modellierten Kollaborationen auf Relationen im Metamodell abgebildet werden. Dies geschieht mit der Metaklasse `CollaborationMapping`, mit der zum einen auf eine `MultiplicityCollaboration` verwiesen wird. Um jedoch größtmögliche Generizität zu garantieren, kann eine Kollaboration nicht nur auf eine Relation einer Metaklasse zu einem direkten Nachbarn abgebildet werden. Diese Beziehung würde ein direktes Zusammenwirken zweier Metaklassen repräsentieren. Um aber auch indirektes Zusammenwirken zu ermöglichen, kann mit dem Metamodell aus Abbildung 6.13 eine Rollen-Kollaboration auch auf einen Pfad im Ziel-Metamodell abgebildet werden. Damit wird erreicht, dass eine Kollaboration zwischen zwei Rollen dem Zusammenwirken zweier Klassen mit Zwischenklassen entsprechen kann. Derartige Zwischenklassen sind im Kontext des Modell-Refactorings dann irrelevant, werden aber benötigt, um das indirekte Zusammenspiel zweier Metaklassen zu realisieren. Zu diesem Zweck existiert zum anderen die 1:n-Komposition zwischen `CollaborationMapping` und `ReferenceMetaClassPair`, mit der die vorher referenzierte `MultiplicityCollaboration` auf die hier verwiesene `EReference` aus Ecore abgebildet wird. Des Weiteren kann optional für jedes `ReferenceMetaClassPair` zusätzlich eine `EClass` referenziert werden. Dies ist notwendig, wenn beispielsweise eine `EReference` zu einer abstrakten Metaklasse führt. Mit dem Verweis auf eine konkrete `EClass` kann somit die gewünschte konkrete Unterklasse angegeben werden, um eventuell den Pfad darüber noch weiter zu spezifizieren. Diese Beziehung zu `ReferenceMetaClassPair` ist geordnet, das heißt, mit mehreren `EClass-EReference`-Paaren wird ein Pfad von einer Metaklasse zu einer anderen über die angegebenen Referenzen definiert.

Damit wurde das Metamodell eines *Role Mapping Models* vollständig erläutert. Es wird einerseits dazu benutzt, um Rollen eines *Role Models* auf Metaklassen des Ziel-Metamodells abzubilden. Andererseits können Kollaborationen komplexen Pfaden im Ziel-Metamodell zugewiesen werden. Dies soll hier an einem Beispiel gezeigt werden. Es sei dazu abermals auf Abbildung 5.3 verwiesen, in der die Metaklasse `Classifier` über die Relation `generalization` zur gleichnamigen Metaklasse navigieren kann. Von dort aus gelangt man über `general` wieder zurück zum `Classifier`. Diese Beziehung steht für die Superklassen, die von einer Unterklasse geerbt und demnach referenziert werden. Derartige Relationen kommen in nahezu jedem Metamodell vor. In einem *Role Model* könnte diese Kollaboration, wie in Abbildung 6.14(a) dargestellt, modelliert werden. Da Kollaborationen mittels des hier vorgestellten Metamodells auf Pfade abgebildet werden können, kann dieses einfache *Role Model* den beschriebenen Relationen zwischen `Classifier` und `Generalization` zugewiesen werden. Das Resultat ist in Abbildung 6.14(b) zu sehen. Zur Verdeutlichung sind die Rollen und ihre *Role Association* blau markiert.

Mit den drei in diesem Kapitel erläuterten Metamodellen wurde eine Möglichkeit geschaffen, Modell-Refactorings generisch zu spezifizieren und anschließend auf beliebige Ziel-Metamodelle anzuwenden. Mit diesem Konzept können Modell-Refactorings über Metamodell-Grenzen hinaus wiederverwendet werden. Außerdem konnten so der Vorteil der Generizität des Triskell-Ansatzes und der Vorteil der exakten Kontrolle von *EMF Refactor* und vom *Operation Recorder* vereint werden. Durch dieses Konzept ergibt sich

(a) Beispiel-Kollaboration *Role Association*



(b) Mapping auf dieselbe Metaklasse entlang eines Pfades

Abbildung 6.14: Abbildung einer *Role Association* auf einen Pfad zwischen Metaklassen

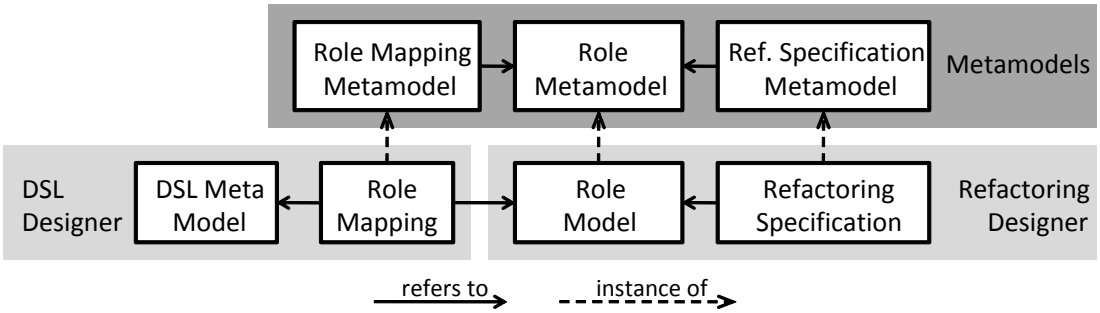


Abbildung 6.15: Übersicht über die Beziehungen aller Modelle und der Stakeholder

neben dem DSL-Designer ein neuer Stakeholder: der Refactoring-Designer. Dieser definiert das Modell-Refactoring generisch und teilt dem DSL-Designer dann die enthaltenen Rollen mit. Da dieser das Ziel-Metamodell (also das Metamodell seiner DSL) bis ins kleinste Detail kennt, relativiert sich der vom Team des *Operation Recorder* vorgebrachte Einwand, ein Refactoring-Designer muss sich im jeweiligen Metamodell auskennen. Mit dem hier vorgestellten Ansatz konnten die Experten von Modell-Refactorings erfolgreich von den DSL-Experten getrennt werden. Die Abbildung 6.15 dient einem zusammenfassenden Überblick über die Zusammenhänge aller Modelle und ihrer Metamodelle sowie der Stakeholder. Dort sind die drei Metamodelle dunkel und die konkreten Modelle sowie das Ziel-Metamodell grau hinterlegt. Des Weiteren ist zu erkennen, dass der DSL-Designer lediglich ein *Role Mapping Model* erstellen muss, um ein Modell-Refactoring, das vom Refactoring-Designer modelliert wurde, für seine DSL verfügbar zu machen. Mit diesem Überblick wird der Kern des hier zu entwickelnden Konzeptes abgeschlos-

sen. Im nachfolgenden Kapitel wird die Spezifizierung der Vor- und Nachbedingungen diskutiert, mit denen angegeben werden kann, unter welchen Umständen ein Modell-Refactoring überhaupt durchgeführt werden darf.

6.3 Vor- und Nachbedingungen

In Anforderung 6 wurde erläutert, dass Vor- und Nachbedingungen eines Modell-Refactorings von großer Bedeutung sind, um den Zustand eines Modell näher zu beschreiben, bevor es refaktoriert werden kann. Sind diese Bedingungen nicht erfüllt, darf ein Modell-Refactoring nicht durchgeführt werden [HA08]. Alle in Kapitel 5 vorgestellten Ansätze ermöglichen in der einen oder anderen Art und Weise die Spezifikation von Vorbedingungen, die von Opdyke als Voraussetzung für den Erhalt der Semantik formuliert wurden [Opd92]. Roberts hat später argumentiert, dass Code-Refactorings meistens nicht isoliert auftreten, sondern oft in einer Sequenz durchgeführt werden. So bereitet ein Refactoring den Weg für das darauf folgende, weshalb ersteres auch die Vorbedingungen des zweiten sicherstellen muss. Damit begründet Roberts nicht nur die Wichtigkeit von Vorbedingungen, sondern auch die Wichtigkeit von Nachbedingungen, bereiten doch die Nachbedingungen des einen schon die Vorbedingungen des anderen vor und vereinfachen unter Umständen so die Analysen zur Beibehaltung des Verhaltens [Rob99].

Vorbedingungen definieren meist solche strukturelle Eigenschaften des zu refaktorierenden Modells, die direkt darin abgelesen werden können. Beispielsweise kann *Extract Method* nur durchgeführt werden, wenn keine existierende Methode den Namen und die Signatur der neu hinzukommenden trägt. Nachbedingungen hingegen treffen Aussagen entweder über strukturelle Eigenschaften, die vor dem Modell-Refactoring noch nicht verfügbar waren, oder aber Aussagen über nicht-strukturelle Eigenschaften des Modells. Dazu zählen vor allem Angaben zur Semantik beziehungsweise zum Verhalten, wie beispielsweise die Durchführung von *Extract Method* in einem Modell eines Systems mit hohen Echtzeit-Anforderungen. In den Nachbedingungen könnte dann zum Beispiel angegeben werden, dass sich die Ausführung der Methode, in der die verschobenen Anweisungen enthalten waren, um maximal zwei Millisekunden verzögern darf. Auf den Aspekt der Bewahrung des Verhaltens wird im Kapitel 6.4 nochmals eingegangen.

Da in dem vorgestellten Konzept Modell-Refactorings aber generisch spezifiziert werden und dann auf gewünschte Ziel-Metamodelle abgebildet werden, muss hier eine Unterscheidung der Vor- und Nachbedingungen vorgenommen werden. Zum einen müssen Bedingungen sprachspezifisch definiert werden, das bedeutet auf der Grundlage des Metamodells, in dem Modell-Refactorings angewendet werden sollen. Dies kann mit dem weit verbreiteten Standard der OCL vorgenommen werden, wie es beispielsweise auch im *Operation Recorder* gehandhabt wird [BLS+09]. Außerdem ist die OCL in vielen wissenschaftlichen Arbeiten als Methode zur Formulierung von Bedingungen für Metamodelle anerkannt [ZLG05, vdSJM07, TMM08, MMBJ09, BSW+09]. Dazu werden OCL-Ausdrücke auf Metamodell-Ebene spezifiziert, die in gewissem Maße auch Garantie für die Erhaltung des Verhaltens übernehmen können [SPTJ01], allerdings nur solange, wie das Verhalten strukturell im Metamodell, mittels sogenannter *Well-Formedness Rules*

(WFRs), beschreibbar ist. Eine andere Art der sprachspezifischen Bedingungen sind solche, die sowieso im Metamodell der DSL spezifiziert wurden. Auf diese Weise können Metaklassen mit OCL-Ausdrücken annotiert werden, wodurch auch hier WFRs definiert werden. Auch die Bedingungen, die global in einem Metamodell angegeben wurden, müssen im lokalen Kontext eines Modell-Refactorings danach weiterhin erfüllt sein.

Des Weiteren müssen neben den sprachspezifischen Bedingungen noch generische Bedingungen differenziert werden. Diese sind demnach unabhängig vom gewünschten Ziel-Metamodell und deshalb in jeder Sprache, in der ein Modell-Refactoring angewendet wird, zu erfüllen. Dies gilt sowohl für die Vorbedingungen als auch für die Nachbedingungen. Da das Zusammenwirken der teilnehmenden Elemente eines Modell-Refactorings in diesem Ansatz mit einem *Role Model* spezifiziert wird, müssen OCL-Bedingungen demnach nicht auf einem Metamodell, sondern auf Rollen und deren Kollaborationen definiert werden. Darüber hinaus sind konventionelle OCL-Interpreter dann nicht mehr einsetzbar, da sie nicht in der Lage sind, OCL-Ausdrücke in einem *Role Model* mit gleichzeitiger Berücksichtigung des *Role Mapping Models* in das gewünschte Ziel-Metamodell, auszuwerten. Ein derartiger Interpreter muss demnach nicht nur die Strukturen im *Role Model* untersuchen, sondern diese auch auf die abgebildeten Metaklassen und Pfade übertragen. Da die Implementierung eines Rollen-basierten OCL-Interpreters nicht trivial und auch nicht primäres Ziel der vorliegenden Arbeit ist, wird die Angabe von Bedingungen im Kapitel 7 nicht umgesetzt. Allerdings wird diesbezüglich im Kapitel 8 nochmals ein kleiner Ausblick gegeben.

Die Beibehaltung der Semantik von Modellen ist eng mit der Angabe von Vor- und Nachbedingungen verwoben. Das damit einhergehende Verhalten ist grundsätzlich eine Eigenschaft von Modellen, die schwer zu beweisen ist. Im folgenden Kapitel wird dieser wichtige Aspekt des Modell-Refactorings diskutiert.

6.4 Bewahrung der Semantik

In Kapitel 2.3 wurde schon erwähnt, dass die Semantik einer DSL beziehungsweise eines Metamodells nicht als absolut angesehen werden kann. Grundsätzlich gilt, dass eine Sprache nicht nur aus ihren Konstruktionsregeln (Metamodell), die angeben, wie darin Sätze (Modell) gebildet werden können, sondern auch aus der vom DSL-Designer beabsichtigten Bedeutung besteht. Diese muss in der Spezifikation einer DSL auch enthalten sein, damit das Verständnis der Sprache kommuniziert werden kann. Die Ursache davon ist, dass die Bedeutung eines Begriffes oder eines Konzeptes von jedem Menschen subjektiv verstanden und bewertet wird, in Abhängigkeit von den Erfahrungen, die eine Person in der Umgebung dieses Konzeptes schon gesammelt hat [Kle08, S. 132f]. Aus diesem Grund kann die Bedeutung eines Metamodells auf dem einfachsten Weg verbal erklärt oder in Textform festgehalten werden, um sie anderen Menschen mitzuteilen. Die neue Herausforderung, die beim generischen Modell-Refactoring zutage tritt, ist die, dass, im Gegensatz zu den Code-Refactorings aus Kapitel 3.1, die Semantik der Ziel-Metamodelle nicht vorhergesehen werden kann und demnach auch nicht generisch anzugeben ist, da einem Modell-Refactoring in jedem Ziel-Metamodell eine vollständig andere Bedeutung

zukommt. Des Weiteren muss die Semantik eines Metamodells formal spezifiziert sein, damit sie entweder nach einem Modell-Refactoring automatisiert überprüft werden oder ein Modell-Refactoring schon vorher, anhand der *Refactoring Specification*, für das Ziel-Metamodell ausgeschlossen werden kann, da es dafür nicht Semantik-erhaltend ist. Diese Hürden konnten in der Forschung bisher noch nicht überwunden werden [HA08], da es zum einen keinen einheitlichen Formalismus zum Beschreiben der Semantik gibt [MDDB+03], und zum anderen das Beweisen von semantischen Invarianten sehr schwer ist [Rob99, MRG09].

Einen pragmatischen Ansatz stellt das Testen vor und nach dem Modell-Refactoring dar, wie es auch schon von Fowler et al. in [FBB+99] motiviert wurde. Dabei werden vor und nach jedem Modell-Refactoring Tests durchlaufen und die Ergebnisse beider Durchläufe miteinander verglichen [MTM08]. Voraussetzung dafür ist allerdings, dass man eine möglichst alle Aspekte abdeckende *Test Suite* besitzt. Stimmen die von beiden Testläufen erzeugten Ausgaben überein, besteht eine hohe Chance, dass das Verhalten beibehalten wurde [MTM07]. Ein entscheidender Nachteil dieses Ansatzes ist, dass so die Durchführungsdauer des Modell-Refactorings deutlich steigen kann.

Andere Ansätze werden von Kleppe in [Kle08] beschrieben. Beispielsweise kann die Semantik eines Metamodells in eine für den Empfänger verständliche Sprache umgeschrieben werden. Bezogen auf den Menschen kann das bedeuten, dass eine Sprache in eine andere übersetzt wird (beispielsweise Deutsch in Spanisch). Für die automatisierte Überprüfung der Bewahrung des Verhaltens eines Modells nach einem Modell-Refactoring bedeutet dies, dass die Semantik in einer bekannten formalen Sprache, wie beispielsweise Java, beschrieben werden muss. Damit kann dann eindeutig spezifiziert werden, was unter einem Ziel-Metamodell zu verstehen ist, und wie es sich verhält. Der Nachteil dieses Ansatzes ist allerdings, dass die Beschreibung der Semantik mit einer konkreten Programmiersprache implementiert werden muss und somit nur schwer wiederzuverwenden ist, da sie von einer konkreten Umgebung abhängt.

Neben diesen Ansätzen wird auf weiteren Gebieten zur Formalisierung von Semantik, beziehungsweise auf dem des automatischen Beweisens der Beibehaltung des Verhaltens von Modell-Refactorings, geforscht [vKCKB05, RLK+08, BK10]. Diese Ansätze gehen einen sehr viel mathematischeren Weg. Dadurch ist das Verhalten zwar prinzipiell formal definierbar, jedoch wird die Spezifikation sehr komplex, wodurch das automatische Beweisen nicht vereinfacht wird. Außerdem werden sehr hohe Anforderungen an den DSL-Designer gestellt, was sich für ihn negativ auswirkt.

Durch diese kurze Vorstellung vorhandener Möglichkeiten, die Semantik formal zu spezifizieren, wird deutlich, dass es keinen einheitlichen Formalismus gibt. Es ist wichtig zu verstehen, dass der DSL-Designer die Aufgabe hat, die Bedeutung seiner Sprache formal zu beschreiben, da diese in einem generischen Modell-Refactoring nicht vorausgesehen werden kann. Das bedeutet insbesondere, dass mittels des Ansatzes der Definition eines *Role Models*, der Erstellung der *Refactoring Specification* auf Basis der Rollen und der anschließenden Abbildung der Rollen auf das Ziel-Metamodell mittels des *Role Mapping Models* lediglich strukturelle Transformationen von Modellen generisch definiert werden können. Es liegt dann am DSL-Designer, auf welche Bereiche seines Metamodells ein Modell-Refactoring abgebildet wird, und ob das Verhalten in diesem Kontext für ihn

ausreichend erhalten bleibt. Die semantische Wohlgeformtheit von Modellen kann mit OCL-Ausdrücken relativ einfach definiert werden; die Frage, ob dies ausreicht, muss jeder DSL-Designer selbst für sich beantworten. Zudem ist auch denkbar, dass Änderungen des Verhaltens in Kauf genommen werden können, wenn vor einem Modell-Refactoring klar identifiziert werden kann, inwiefern sich die Änderungen auswirken [MTM07, MRG09].

Dieses komplexe und noch junge Thema der Bewahrung des Verhaltens von Modell-Refactorings birgt eine Fülle an Spielraum, aufgrund der fehlenden Standardisierung, und auch einen hohen Diskussionsbedarf. In diesem Kapitel konnte nur ein kurzer Abriss darüber dargestellt und verdeutlicht werden, dass dieser Bereich des Modell-Refactorings viel Raum für weitere Untersuchungen lässt. Deshalb wird die Beibehaltung der Semantik im Kapitel 7 nicht berücksichtigt, jedoch enthält Kapitel 8 einen abschließenden Ausblick darüber.

Im nachfolgenden Kapitel wird nun der letzte wichtige Aspekt des generischen Modell-Refactorings betrachtet. Dieser beinhaltet die horizontale und vertikale Konsistenz, die nach der in dieser Arbeit gegebenen Definition garantiert werden muss.

6.5 Horizontale und vertikale Konsistenz

Die horizontale und vertikale Konsistenz wurde in keinem der in Kapitel 5 vorgestellten Ansätze berücksichtigt. In all diesen Ansätzen werden nur einzelne Modelle refaktoriert, wodurch darauf verweisende Modelle unberührt bleiben. Man stelle sich beispielsweise ein Klassen-Modell der UML vor. Des Weiteren existiert ein separates Sequenz-Modell, in dem auf eine Klasse des ersten Modells verwiesen wird. Es handelt sich hierbei also um zwei unterschiedliche Modelle. Wird nun die referenzierte Klasse umbenannt, befindet sich das Sequenz-Modell nach dem Modell-Refactoring in einem inkonsistenten Zustand, wenn man nur am Modell lokale Modifikationen durchführt, wie es bei allen drei vorgestellten Ansätzen der Fall ist. Ein anderes Beispiel ist die Anpassung der Repräsentation eines Modells, wenn das Modell selbst refaktoriert wird. So muss sich der Name einer Klasse im Klassen-Diagramm auch ändern, wenn sich der Name im Klassen-Modell geändert hat. Bei diesen Beispielen handelt es sich um horizontale Konsistenz, die nicht gewahrt wird, da alle Modelle den gleichen Abstraktionsgrad haben und sich auf MOF-Ebene M1 befinden.

Als zweites Beispiel wird an dieser Stelle noch einmal auf den Ausschnitt des UML-Metamodells in Abbildung 5.3 auf Seite 35 verwiesen. Angenommen, die Metaklasse `Class` wird in einem Modell-Refactoring zu `Klasse` umbenannt. In diesem Fall ändert sich also die Regel, mit der Klassen in einem Klassen-Modell als Instanz des UML-Metamodells erstellt werden können. Eine kleine Änderung mit enormen Ausmaßen, da sich alle bisher erzeugten Klassen-Modelle nach dem Modell-Refactoring in einem inkonsistenten Zustand befinden, denn sie sind nicht mehr konform zu ihrem Metamodell. Dabei handelt es sich um vertikale Konsistenz, befinden sich doch die in Verbindung stehenden Modelle (Metamodell und seine Instanzen) auf den unterschiedlichen MOF-Abstraktionsstufen M2 und M1.

Die horizontale Konsistenz wird von den meisten Ansätzen, wie auch bei den hier

vorgestellten zu sehen, außer Acht gelassen [MRG09]. Die vertikale Konsistenz wird in der Literatur sehr häufig lediglich nur mit der Model Driven Architecture (MDA) in Verbindung gebracht [MVG06, MTM07, Hub08]. Vertikale Konsistenz wird dort oft nur als die Konsistenz zwischen einem UML-Modell und dem daraus generierten Code bezeichnet [vGSMD03, MTM08]. Bei Änderungen in der höheren Ebene wird der darunter liegende Code einfach neu generiert. Wie jedoch am obigen Beispiel gezeigt, ist vertikale Konsistenz sehr viel mehr als nur das.

Die horizontale Konsistenz ist schon aus dem Bereich des Code-Refactorings bekannt, denkt man beispielsweise an Klassen in einer Vererbungshierarchie. Wird eine Klasse einer höheren Stufe umbenannt, so werden die Verweise auf diese in der erbenden Klasse automatisch aktualisiert. Ebenso werden Objekt-Deklarationen angepasst, wenn deren Typen der umbenannten Klasse entsprechen. Deshalb ist es unverständlich, dass das Thema der horizontalen und vertikalen Konsistenz in bisherigen Ansätzen so wenig Beachtung erfuhr.

Diesem Problem der Konsistenz wird in dem hier vorgestellten Konzept zuerst auf philosophischer Ebene begegnet. Wie in Kapitel 2.3 schon erklärt, ist es durchaus möglich, Programmiersprachen auf die Ebene von Modellierungssprachen anzuheben, da für sie auch Metamodelle spezifiziert werden können, wie es mit JaMoPP gezeigt wurde. In Abschnitt 2.2.2 wurde zudem schon erläutert, dass Metamodelle ausdrucksstärker als kontextfreie Grammatiken sind. Aus diesem Grund kann angenommen werden, dass für jede Programmiersprache ein Metamodell erstellt werden kann. Dadurch verlässt man die technologische Trennung zwischen Code und Modell und braucht nur noch von Modellen zu sprechen. Aus diesem Blickwinkel wird klar, dass viele für die horizontale und vertikale Konsistenz wichtige Informationen schon in den Modellen enthalten sind. In jedem Modell ist gespeichert, welches andere Modell es referenziert. Mit dieser Information kann schon ein Teil des formulierten Problems gelöst werden, indem nach durchgeführten Modell-Refactorings einfach alle referenzierten Modelle dahingehend überprüft werden, ob sie Verweise auf die veränderten Elemente besitzen. Ist dies der Fall, werden diese angepasst. Was nun noch fehlt, ist die andere Richtung: Wird ein Modell refaktoriert, so müssen auch die Modelle überprüft werden, die darauf verweisen. Das Problem hierbei besteht jedoch darin, nicht die direkten, sondern die inversen Referenzen des refaktorierten Modells zu finden. Direkte Referenzen können einfach aus dem Modell selbst abgelesen werden und bedürfen keiner weiteren Überlegungen. Inverse Referenzen hingegen sind nicht auf triviale Weise zu ermitteln. Als Lösung wird in dieser Arbeit ein Index-Mechanismus vorgeschlagen. Dieser nimmt alle Modelle der Modellierungs-Umgebung des Anwenders auf und kann dann transparent zum Nutzer alle Referenzen auflösen. Inverse Referenzen sind auf diese Art im Index gespeichert. Anschließend muss dieser nur in Abhängigkeit vom refaktorierten Modell nach dessen inversen Referenzen befragt werden. Damit wird dem Problem auf der Ebene der Implementierung begegnet, was im Kapitel 7.6 auch entsprechend umgesetzt wird.

Durch die Betrachtung jedes Artefakts als Modell konnte die horizontale und vertikale Konsistenz durch deren Stärken des Graphen-Charakters in Angriff genommen werden. Die horizontale Konsistenz kann wie oben beschrieben gewahrt werden. Direkte Referenzen sind hierbei im Modell abzulesen und inverse Referenzen werden über einen

Index ermittelt. Die Frage der vertikalen Konsistenz hingegen kann nicht so einfach beantwortet werden. Mit der zuvor erläuterten Wahrung der horizontalen Konsistenz mit Hilfe eines Indexes kann die vertikale Konsistenz nur sichergestellt werden, wenn es sich um dynamische Instanziierung eines Metamodells handelt. Das heißt, dass ein Modell auf MOF-Ebene M1 beispielsweise nur dann automatisch an eine Umbenennung einer seiner Metaklassen auf M2-Ebene angepasst werden kann, wenn sich beide in derselben Modellierungs-Umgebung befinden. Ist dies nicht der Fall, würden zwei verschiedene Indizes existieren: einer für die Modellierungs-Umgebung des Metamodells und ein zweiter für dessen Instanzen. Demzufolge müsste eine Möglichkeit geschaffen werden, wie eine Verbindung zwischen beiden hergestellt werden kann. Diese Überlegungen liegen aber außerhalb des Fokus dieser Arbeit, weshalb an dieser Stelle darauf verzichtet wird.

Der nachfolgende Abschnitt enthält abschließend eine Zusammenfassung und eine Bewertung der Konzeption des generischen Modell-Refactorings.

6.6 Zusammenfassung

Ausgehend von den in Kapitel 4 formulierten Anforderungen und den in Kapitel 5 identifizierten Stärken und Schwächen angrenzender Ansätze, wurde in diesem Kapitel das Konzept des Rollen-basierten generischen Modell-Refactorings vorgestellt. Aus der detaillierten Erläuterung anderer Ansätze gingen zwei Fragen hervor, die mit dem Konzept dieser Arbeit beantwortet werden mussten. Es war zum einen zu klären, wie Modell-Refactorings unabhängig vom Ziel-Metamodell zu spezifizieren sind. Als Grundlage der Beantwortung dieser Frage diente der Triskell-Ansatz aus Kapitel 5.1. Zum anderen galt es zu beantworten, wie trotz der Generizität die zu refaktorierenden Strukturen exakt kontrolliert werden können. Hierzu orientierte man sich an den Ansätzen von *EMF Refactor* und *Operation Recorder*. Des einen Stärke ist des anderen Schwäche. Deshalb lag die Herausforderung darin, die Vorteile von beiden zu verschmelzen.

In diesem Kapitel wurde zur Lösung dieser Probleme und zur Beantwortung der Fragen ein Rollen-basierter Ansatz konzipiert. Dieser fußt auf der Theorie der Rollen, wie sie von Reenskaug, Riehle und Gross eingeführt und in Kapitel 6.1 vorgestellt wurden. Daraus entwickelte sich die Notwendigkeit, im Gegensatz zum Triskell-Ansatz, die teilnehmenden Elemente eines Modell-Refactorings nicht statisch, sondern immer abgegrenzt im Kontext desselben zu betrachten. In Abschnitt 6.2.1 wurde dafür das Metamodell vorgestellt, mit dem es möglich ist, Rollen und deren Kollaborationen zu modellieren. Ein *Role Model*, welches alle nötigen Elemente eines Modell-Refactorings beschreibt, ist unabhängig von den Metamodellen, in denen ein Modell-Refactoring angewendet werden soll, und bildet so die Grundlage für die geforderte Generizität.

Ein *Role Model* allein reicht jedoch noch nicht, um auch spezifizieren zu können, welche einzelnen Schritte in einem Modell-Refactoring durchzuführen sind. Dafür wurde in Abschnitt 6.2.2 das Metamodell der *Refactoring Specification* erläutert, mit dem es möglich ist, nur in Abhängigkeit von definierten Rollen aus einem *Role Model*, die Teilschritte anzugeben. Ein derartiges Modell muss dann als Eingabe der ausführenden Komponente in der Implementierung dienen. Darauf wird in Kapitel 7.5 im Detail eingegangen.

Als ein noch zu lösendes Problem erwies sich die Frage, wie man in einer *Refactoring Specification* die zu refaktorierenden Elemente ansprechen kann, um konkrete Befehle auf ihnen spezifizieren zu können. Da aufgrund der höchsten Priorität der Generizität keine Abhängigkeiten zu den Ziel-Metamodellen bestehen durften, fiel die Wahl auf Variablen. Mit ihnen war es möglich, zu einer *Refactoring Specification* lokale Objekte zu definieren, die zur Ausführungszeit dann im entsprechenden Metamodell aufgelöst werden. Mit den bis hierhin vorgestellten Metamodellen konnte die Frage der Generizität positiv beantwortet werden.

Zur Beantwortung der Frage nach der exakten Kontrolle der zu refaktorierenden Strukturen wurde in Abschnitt 6.2.3 das Metamodell zur Abbildung eines *Role Models* auf ein Ziel-Metamodell vorgestellt. Damit müssen nur noch den Metaklassen die definierten Rollen und Kollaborationen zugewiesen werden. Als Besonderheit ist es zudem möglich, nicht nur Kollaborationen zwischen Rollen auf Relationen zwischen Metaklassen, sondern auch auf Pfade zwischen Metaklassen abzubilden. Dies ist die Voraussetzung für die Möglichkeit der Abbildung von *Role Models* auf möglichst viele verschiedene Metamodelle. Im Allgemeinen ist nicht davon auszugehen, dass alle *gleiche* Strukturen haben, weshalb die Abbildung auf Pfade nur noch *ähnliche* Strukturen erfordert. Wichtig zu erwähnen ist, dass ein *Role Mapping Model* zwar die Abbildungen mehrerer *Role Models* enthalten kann, jedoch stets nur auf ein und dasselbe Metamodell. Somit ist es insbesondere möglich, dasselbe *Role Model* auf verschiedene Bereiche im selben Metamodell abzubilden, was der generische Triskell-Ansatz nicht hergibt.

Anschließend wurde in Kapitel 6.3 die Notwendigkeit der Angabe von Vor- und Nachbedingungen diskutiert. Nur wenn alle Bedingungen erfüllt sind, darf ein Modell-Refactoring schlussendlich auch durchgeführt werden. Des Weiteren beinhaltet Kapitel 6.4 die Diskussion der Bewahrung des Verhaltens eines Modells nach dem Modell-Refactoring. Beide Aspekte des Modell-Refactorings bedürfen allerdings für sich einer intensiven Untersuchung vorhandener Ansätze und einer daraus resultierenden Lösung. Da dies nicht primär Gegenstand dieser Arbeit ist, wird diesbezüglich in Kapitel 8 ein Ausblick gegeben.

Schließlich konnte im Kapitel 6.5 argumentiert werden, dass sich mit der Betrachtung von Artefakten als Modelle die Probleme der horizontalen und vertikalen Konsistenz sehr viel einfacher lösen lassen, als es bisher in anderen Arbeiten diskutiert wurde. Die Bewahrung der Konsistenz kann mit einem geeigneten Index-Mechanismus für Modell-Referenzen gelöst werden und wird in Kapitel 7.6 detailliert erläutert.

Damit konnten die Hürden der Generizität und der gleichzeitigen exakten Kontrolle der zu refaktorierenden Strukturen überwunden werden. Die Vor- und Nachbedingungen sowie die Beibehaltung der Semantik erfordern noch weitergehende Forschung. Auf die hier vorgestellte Art und Weise des Rollen-basierten generischen Modell-Refactorings konnten somit alle Anforderungen aus Kapitel 4, die nicht von der Implementierung abhängen, erfüllt werden. Für Anforderung 5 und 6 muss auf zukünftige Arbeiten verwiesen werden.

7 Implementierung und Evaluation

In Kapitel 6 wurde das Konzept zur generischen Spezifikation von Modell-Refactorings auf der Basis von Rollen erläutert. In diesem Kapitel werden nun die prototypische Umsetzung sowie die Evaluation im Kontext von EMFText vorgestellt. Die Wahl eines geeigneten Modellierungs-Frameworks fällt an dieser Stelle nicht schwer. Die Umsetzung erfolgt auf der Grundlage von Eclipse und dem schon des öfteren erwähnten Eclipse Modeling Framework (EMF). Der Grund dafür ist, dass das im EMF benutzte Meta-Metamodell Ecore vollständig den Standard EMOF abbildet und maßgeblich zu dessen Weiterentwicklung beigetragen hat [SBPM08, S. 40]. Außerdem ist EMF in der breiten Gemeinschaft der MDSD akzeptiert und konnte sich dort mit Ecore als Quasistandard etablieren [MFJ05, BEK+06b, MTM07, TMM08].

Des Weiteren ist EMF integraler Bestandteil der quelloffenen Plattform Eclipse, die zum einen sehr leistungsstark als IDE ist und zum anderen sehr gute Werkzeuge und Mechanismen bereitstellt. Mit diesen kann man eigene Plugins entwickeln, die ihrerseits wiederum als Werkzeug in Eclipse eingebunden werden können. Auf dieser Grundlage wurden zahlreiche Eclipse-Plugins entwickelt, die in ihrer Gesamtheit ein Framework zur Spezifikation und Ausführung generischer Modell-Refactorings bilden. Dieses Framework wird *Refactory* genannt. Die Architektur von *Refactory* und das Zusammenwirken der darin enthaltenen Plugins wird in Abbildung 7.1 dargestellt. Bevor jedoch die Bestandteile im Einzelnen vorgestellt werden, wird im nachfolgenden Kapitel zunächst das EMF erläutert, um einen Überblick darüber zu geben, worin dessen Stärken liegen und wie Modellierer bei der Arbeit mit EMF unterstützt werden können.

7.1 Eclipse Modeling Framework

Das EMF ist das Framework von Eclipse, mit dem Modellierungs-Werkzeuge bereitgestellt werden, die auf einfache Art und Weise MOF-konforme Metamodelle entwickeln können, auf deren Grundlage man dann konkrete Modelle instanziieren kann. Außerdem bietet EMF Möglichkeiten der Modell-Persistenz, der Notifikation und Validierung nach Modell-Änderungen sowie eine ausgereifte reflektierende Programmierschnittelle (API) an.

Das Meta-Metamodell zur Erzeugung von anderen Metamodellen ist Ecore. Wie schon erläutert, ist Ecore die Referenzimplementierung von EMOF und enthält im Wesentlichen Metaklassen zur Definition von Klassen, Assoziationen zwischen ihnen, Attributen und Datentypen zum Erstellen eigener Metamodelle. Dabei dient Ecore selbst als die erste Möglichkeit, ein Metamodell zu entwickeln. Darüber hinaus kann ein Metamodell aus einem XML-Schema, aus annotiertem Java-Code oder aus einem Klassen-Modell der UML generiert werden. Modelle der drei letztgenannten Mittel befinden sich gemäß

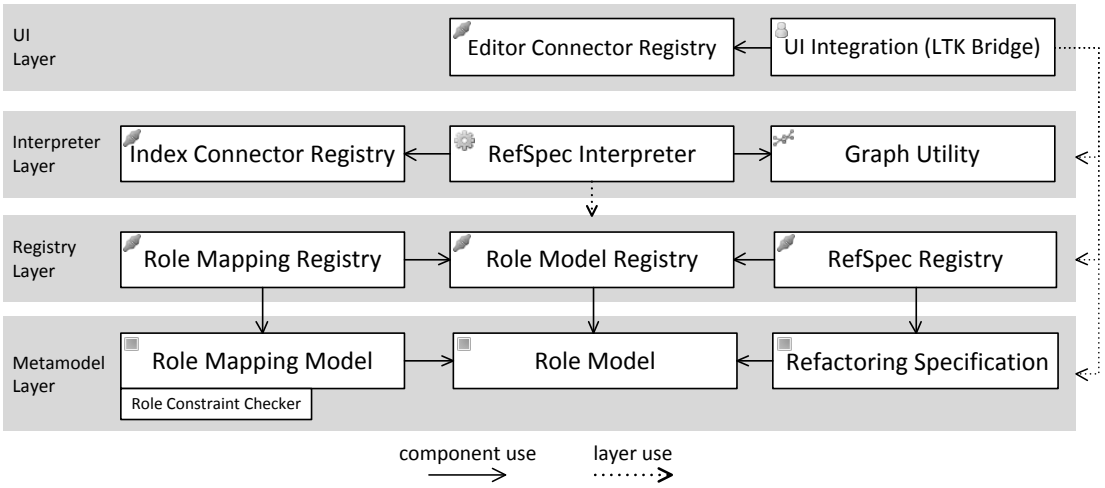


Abbildung 7.1: Architektur des Frameworks *Refactory*

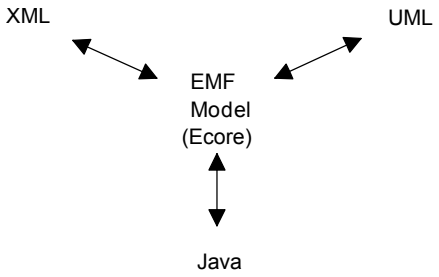


Abbildung 7.2: Ecore als Unifikator zwischen allen Varianten zur Entwicklung von Metamodellen nach [SBPM08, S. 14]

Abbildung 2.1 auf Seite 8 jeweils auf MOF-Ebene M1, weshalb sie nicht direkt als Metamodell genutzt werden können. Durch Ecore können sie jedoch durch Generierung in ein Metamodell auf Ebene M2 angehoben werden. Infolgedessen wirkt Ecore, wie in Abbildung 7.2 zu sehen, als Unifikator zwischen allen vier Varianten der Entwicklung von Metamodellen.

Aus diesen vier Möglichkeiten der Erstellung eines Metamodells resultiert der Fakt, dass mit EMF die Einstiegshürde in die MDSD sehr niedrig ist. Wurde einmal ein Metamodell mit EMF erstellt, so kann dafür die korrespondierende API generiert werden, wodurch der programmatische Zugriff auf Modelle beträchtlich erleichtert wird. Die generierten APIs der Metamodelle *Role Model*, *Refactoring Specification* und *Role Mapping Model*, sowie die reflektierende API von Ecore selbst sind im Abschnitt 7.5 die Grundlage der Implementierung des Interpreters zur Ausführung von Modell-Refactorings. Des Weiteren ist es mit EMF möglich, für jedes Metamodell einen Baum-Editor zu generieren. Dies ist schnell durchgeführt und bietet dem Anwender eine erste Möglichkeit, Modelle zu erstellen. Wie in Kapitel 2.4 schon erläutert, stellt EMFText ein Framework

dar, mit dem für ein Metamodell eine textuelle Syntax definiert werden kann. Dies wird auch basierend auf mit EMF entwickelten Metamodellen durchgeführt, womit der Zusammenhang von EMF und EMFText hergestellt ist. Wie die Definition einer konkreten Syntax im Detail vorzunehmen ist, wird in den Kapiteln 7.2, 7.3 und 7.4 gezeigt. Des Weiteren kann auf Basis von EMF und dem Graphical Modeling Framework (GMF)[1] ein graphischer Editor generiert werden, mit dem es möglich ist, Modelle in einem Diagramm zu entwickeln. Ein Beispiel für einen solchen Editor wurde in Abbildung 6.4 zur Modellierung von *Role Models* gezeigt. Auch darauf wird im nachfolgenden Abschnitt im Detail eingegangen.

Anhand von EMF-basierten Modellen können nun Transformationen durchgeführt werden. Dies kann mit einschlägigen M2M-Sprachen, aber auch, wie schon erwähnt, auf Grundlage der generierten API erfolgen. An den zuvor genannten Beispielen der unterschiedlichen Modell-Repräsentationen ist noch einmal zu erkennen, wie wichtig es ist, dass Modell-Refactorings nur auf den Modellen arbeiten und unabhängig von deren Repräsentation sind. Die Grundlage für den weiteren Verlauf der Implementierung bietet nun die Umsetzung des *Role Models* im folgenden Kapitel.

7.2 Role Model

Die Grafik in Abbildung 6.3 zeigt das Metamodell *Role Model*, wie es mit dem grafischen Ecore-Editor modelliert wurde. Wie zuvor schon erwähnt, können daraus ein Baum-Editor beziehungsweise ein graphischer Editor generiert werden. Mit Hilfe von EMFText ist außerdem ein textueller Editor generierbar. Der Vorteil der textuellen Erstellung von Modellen liegt vor allem darin, dass die Repräsentation vom Menschen leichter lesbar ist, und dass sie sehr einfach in ein Versionsverwaltungs-System eingebracht werden kann. Dadurch ist, im Gegensatz zur Serialisierung als XML-Artefakt, ein textueller Vergleich von verschiedenen Versionen desselben Modells einfacher möglich. Die Differenz zweier Modelle ist in XML nur schwer les- und nachvollziehbar, wohingegen die textuellen Unterschiede besser verständlich sind. Aus diesen Gründen wurde in dieser Arbeit nicht der Baum-Editor für die Erstellung von *Role Models*, sondern eine textuelle Syntax mit EMFText für das entwickelte Metamodell gewählt. Das Listing 7.1 zeigt, wie diese für *Role Models* festgelegt wurde.

Listing 7.1: Konkrete Syntax für *Role Models*

```
1  SYNTAXDEF rolestext FOR <http://www.emftext.org/language/roles>
2  START RoleModel
3
4  TOKENS{
5    DEFINE ML_COMMENT $'/*'.*'*/'$;
6
7    DEFINE ROLE_MODIFIER $'optional'|'input'|'super'$;
8    DEFINE UPPER_IDENTIFIER $('A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
9    DEFINE LOWER_IDENTIFIER $('a'..'z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
10   DEFINE NUMBER $('0')|('-1')|('*')|(('1'..'9')('0'..'9')*)$;
11 }
```

[1] www.eclipse.org/gmf

```
12
13 RULES {
14   RoleModel ::= (comment[ML_COMMENT])?
15     "RoleModel" name[UPPER_IDENTIFIER] "{" roles* collaborations* "}";
16
17   Role ::= (comment[ML_COMMENT])? ( modifier[ROLE_MODIFIER]* )?
18     "ROLE" name[UPPER_IDENTIFIER] ("(" attributes ("," attributes)* ")")? ";";
19
20   RoleAttribute ::= (comment[ML_COMMENT])?
21     modifier[ROLE_MODIFIER]* name[LOWER_IDENTIFIER];
22
23   RoleProhibition ::= source[UPPER_IDENTIFIER] "|-|" target[UPPER_IDENTIFIER] ";";
24
25   RoleImplication ::= source[UPPER_IDENTIFIER] "->" target[UPPER_IDENTIFIER] ";";
26
27   RoleAssociation ::= source[UPPER_IDENTIFIER] sourceName[LOWER_IDENTIFIER]?
         sourceMultiplicity
28     "--" target[UPPER_IDENTIFIER] targetName[LOWER_IDENTIFIER]? targetMultiplicity
           ";";
29
30   RoleComposition ::= source[UPPER_IDENTIFIER] sourceName[LOWER_IDENTIFIER]?
         sourceMultiplicity
31     "<>-" target[UPPER_IDENTIFIER] targetName[LOWER_IDENTIFIER]?
          targetMultiplicity  ";";
32
33   Multiplicity ::= "[" lowerBound[NUMBER] ".." upperBound[NUMBER] "]";
34 }
```

Begonnen wird mit dem Schlüsselwort `SYNTAXDEF`, gefolgt von der gewünschten Dateierweiterung für textuelle *Role Models* (hier `rolestext`). Danach wird hinter `FOR` der Identifikator für das gewünschte Metamodell angegeben. Die zweite Zeile enthält die Metaklasse, die als Startregel für die unten definierten Regeln fungieren soll. Wie zu erwarten, wird dazu die Metaklasse `RoleModel` benutzt. Mit dem Abschnitt `TOKENS` werden textuelle Einheiten als absolute Zeichenreihen oder relativ mittels regulärer Ausdrücke definiert. Die erste Zeile gibt an, wie Kommentare repräsentiert werden. Diese werden, angelehnt an Java, in `/*` und `*/` eingeschlossen und können ein *Role Model*, Rollen und Rollen-Attribute näher erläutern, da diese Unterklassen von `Commentable` sind.

Der Abschnitt `RULES` beinhaltet den wichtigsten Teil der textuellen Syntax, mit dem eine kontextfreie Grammatik für das Metamodell definiert wird. Dort muss für jede konkrete Metaklasse eine Regel angegeben werden, deren Struktur an die Extended Backus–Naur Form (EBNF) erinnert. Der Name der Metaklasse steht auf der linken Seite der Regel und alle strukturellen Eigenschaften dieser Metaklasse, wie Attribute oder Referenzen, müssen auf der rechten Seite vorkommen. In doppelten Anführungszeichen eingeschlossene Zeichenketten definieren Schlüsselwörter, die genauso in einem textuellen Modell an der angegebenen Position vorkommen müssen. Das Fragezeichen besagt, dass der davor angegebene Teil der Regel optional ist. Beispielsweise wurde das Attribut `comment` der Metaklasse `Commentable` nicht als obligatorisch modelliert, weshalb dieses in der ersten Regel zu `RoleModel` als optional markiert wurde. Wie dort schon zu erkennen ist, werden Präzedenzen mittels öffnender und schließender Klammern angegeben. In eckigen Klammern hingegen wird zum einen definiert, wie Attribute von Metaklassen textuell spezifiziert werden, und zum anderen, wie referenzierende Assoziationen anzugeben sind. Der Name eines *Role Models* wurde im obigen Listing mit

`name[UPPER_IDENTIFIER]` angegeben. Das bedeutet, dass der Name mit dem oben definierten regulären Ausdruck `UPPER_IDENTIFIER` zu setzen ist. Dieser beinhaltet alle Zeichenreihen, die mit einem Großbuchstaben beginnen und außerdem Buchstaben, Ziffern und den Unterstrich beinhalten. Referenzierende Assoziationen hingegen sind Beziehungen zwischen Metaklassen, bei denen es sich nicht um Kompositionen handelt. Ein Beispiel dafür ist der rechte Teil der Regel `RoleProhibition`. Diese Metaklasse besitzt, wie in Abbildung 6.3 zu sehen, eine Referenz namens `source` auf `Role`. Da Rollen-Namen in der zweiten Regel auch mit `UPPER_IDENTIFIER` identifiziert werden, wird die Referenz auf die Quell-Rolle mit `source[UPPER_IDENTIFIER]` angegeben. Im Gegensatz zum Fragezeichen für optionale Teile markiert ein Stern Strukturen, die beliebig oft vorkommen dürfen, in Fällen, bei denen es sich um strukturelle Eigenschaften einer Metaklasse handelt, die auch als solche definiert wurden. Ein Beispiel dafür ist die Angabe von `roles*` der Regel `RoleModel`. Dort ist zu erkennen, dass mit einem Stern Kompositions-Beziehungen zwischen Metaklassen definiert werden. Dies bedeutet insbesondere, dass mit der Angabe des Namens einer Komposition die Verbindung einer Regel zur Regel der korrespondierenden Metaklasse hergestellt wird.

Aus dieser konkreten Syntax für das Metamodell von *Role Models* kann nun mit EMFText ein textueller Editor generiert werden. Dieser enthält eine *Outline*, die das Modell in Baum-Form abbildet. Außerdem ermöglicht der Editor *Syntax Highlighting*, *Code Completion*, sowie das Navigieren zwischen Code, mit Hilfe der in Kapitel 2.4 erwähnten Resolver. Um abschließend eine Vorstellung vom generierten Editor und der textuellen Modellierung von *Role Models* zu bekommen, sind in Abbildung 7.3 der Editor und seine Outline für ein Beispiel dargestellt. Dort können die von Listing 7.1 erläuterten Regeln an einem konkreten Beispiel nachvollzogen werden.

Das in Abbildung 7.3 dargestellte *Role Model* heißt *Extract X with Reference Class* und beschreibt die Rollen, die für das Modell-Refactoring benötigt werden, mit dem *Extract Method* generisch umgesetzt werden kann. Bevor die einzelnen Bestandteile des abgebildeten *Role Models* näher erläutert werden, ist in Abbildung 7.4 der schon erwähnte graphische Editor für *Role Models* dargestellt. Dieser wurde umgesetzt, um dem Refactoring-Designer ein zweites Werkzeug zur Verfügung zu stellen, mit dem *Role Models* auf einfache Weise definiert werden können. Dieser wurde leicht an Klassen-Diagramme der UML angelehnt, wodurch die Einstiegshürde weiter gesenkt wird. Dieses Diagramm beschreibt dasselbe Modell wie Abbildung 7.3. Der graphische Editor wurde mit GMF erstellt und ist nahtlos mit dem textuellen Editor verbunden. Das bedeutet, dass Änderungen im graphischen Editor direkt in das textuelle Modell propagiert werden und umgekehrt.

Doch nun sollen die Rollen aus den beiden Abbildungen näher erläutert werden. Da *Extract Method*, wie in Kapitel 3.3 schon erläutert, abstrakt übertragbar ist, wird eine Rolle benötigt, die der ursprünglichen Methode entspricht. Diese Rolle entspricht `OrigContainer` in den Abbildungen 7.3 und 7.4. Von dieser geht eine *Role Composition* zur Rolle `Extract` aus, welche die zu verschiebenden Anweisungen repräsentiert. Außerdem wird ein neuer Container benötigt, der für die neu zu extrahierende Methode steht. Dieser Zusammenhang wird von `NewContainer` realisiert, welcher ebenfalls eine *Role Composition* zu `Extract` hält, da in diesen die ursprünglichen Elemente verschoben

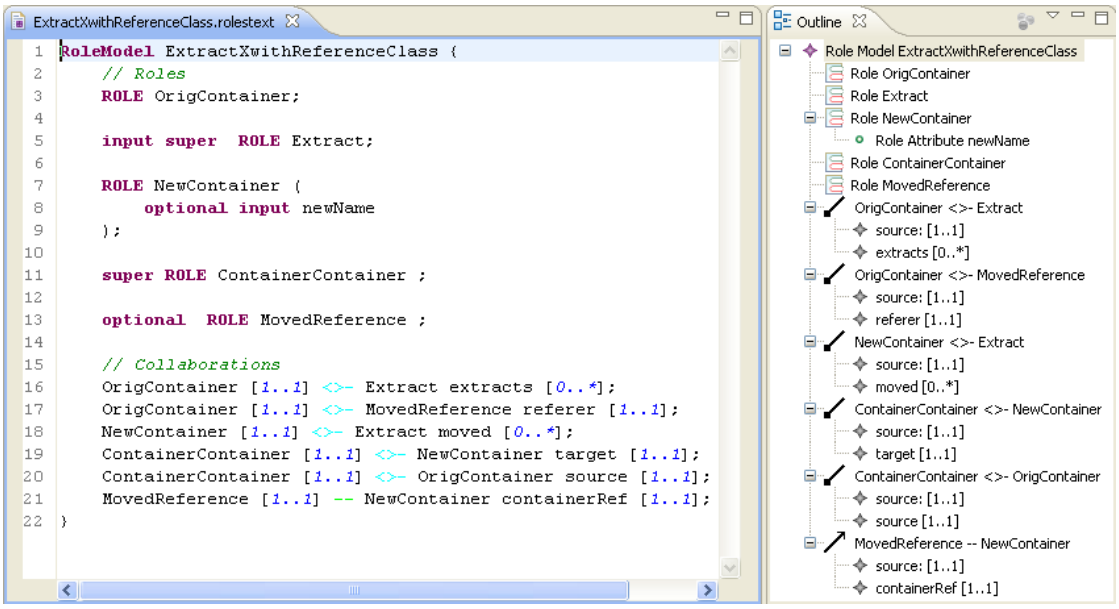


Abbildung 7.3: Mit EMFText generierter Editor mit Outline für *Role Models*

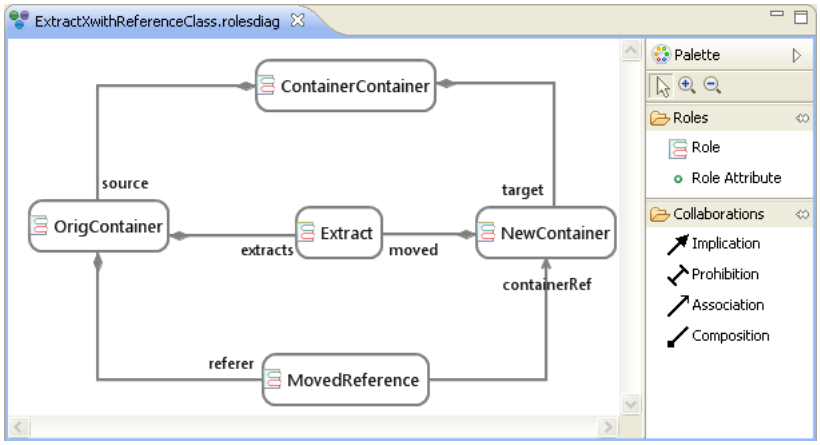


Abbildung 7.4: Graphischer Editor für *Role Models*

werden müssen. Zudem muss an der ursprünglichen Position der Elemente, die `Extract` spielen, eine Referenz auf den neu angelegten Container gesetzt werden. Im konkreten Fall von *Extract Method* aus den Programmiersprachen entspricht diese Beziehung dem Hinzufügen des Methoden-Aufrufs. Im *Role Model* existiert zu diesem Zweck eine *Role Composition* von `OrigContainer` zu `MovedReference`, die wiederum eine *Role Association* zu `NewContainer` hält. Schließlich wird ein Container benötigt, in dem die anderen beiden enthalten sind. In der Welt der objektorientierten Programmiersprachen ist dieser eine Klasse und wird hier von der Rolle `ContainerContainer` umgesetzt. Die bis hierher beschriebene Implementierung der Rollen-Funktionalität in *Refactory* ist in dem in Abbildung 7.1 dargestellten *Metamodel Layer* enthalten. Um es dem Refactoring-Designer nun zu ermöglichen, auf einfache Art und Weise neue *Role Models* in der Eclipse-Plattform verfügbar zu machen, wurde eine *Role Registry* umgesetzt, die sich im *Registry Layer* befindet und jetzt erläutert wird.

Um Erweiterungspunkte zu setzen, bietet die Eclipse-Plattform den Mechanismus der *Extension Points* an. Damit können deklarativ eigene Erweiterungspunkte definiert werden, an denen andere Plugins ihre Erweiterungen spezifizieren. Der Erweiterungspunkt der *Role Registry* enthält nur ein Attribut, über das auf eine Datei verwiesen werden kann, die ein *Role Model* enthält. Um den Zugriff auf registrierte *Role Models* zu vereinfachen und diese auch programmatisch zu registrieren, wurde der Erweiterungspunkt in die Funktionalität einer *Registry* gekapselt. Der Zugriff darauf wird über das Interface aus Listing 7.2 vorgenommen.

Listing 7.2: Interface für den Zugriff auf die *Role Registry*

```
1  public interface IRoleModelRegistry {
2
3    public static IRoleModelRegistry INSTANCE = new BasicRoleModelRegistry();
4
5    public Collection<RoleModel> getAllRegisteredRoleModels();
6
7    public void registerRoleModel(RoleModel roleModel) throws
         RoleModelAlreadyRegisteredException;
8
9    public RoleModel getRoleModelByName(String name);
10
11   public void updateRoleModel(RoleModel model);
12 }
```

Wie dort zu erkennen ist, enthält dieses Interface Methoden zur Abfrage aller registrierten *Role Models* sowie eines konkreten *Role Models*, das über seinen Namen angesprochen wird. Außerdem kann ein *Role Model* registriert und ein vorhandenes auch aktualisiert werden. Beim ersten Zugriff auf die *Role Registry* sammelt diese alle registrierten Erweiterungen des zuvor beschriebenen Erweiterungspunktes ein. Auf diese kann dann über das Interface `IRoleModelRegistry` zugegriffen werden. Ein weiterer Vorteil der *Role Registry* ist der, dass das Ermitteln der Registrierungen des Erweiterungspunktes vollkommen transparent gehalten ist. Dadurch kann darauf auch außerhalb des Eclipse-Kontextes zugegriffen werden, beispielsweise um Unit-Tests durchzuführen, die außerhalb einer Eclipse-Umgebung laufen.

Damit ist die Vorstellung der Implementierung des zentralen unteren Teils aus Abbildung 7.1 abgeschlossen. Im nun folgenden Kapitel wird die Umsetzung der *Refactoring Specification* erläutert, mit der Modell-Refactorings dann schon vollständig generisch spezifiziert werden können.

7.3 Refactoring Specification

Um die Erstellung einer *Refactoring Specification* für den Refactoring-Designer so einfach wie möglich zu halten, wurde für das Metamodell, dessen einzelne Bestandteile in Kapitel 6.2.2 erläutert wurden, eine textuelle Syntax definiert. Dies befähigt den Designer dazu, die Teilschritte des durchzuführenden Modell-Refactorings in wenigen Zeilen zu definieren. Mit einem generierten Baum-Editor wäre dies sehr viel aufwändiger. Außerdem ist man es von anderen Modell-Transformations-Sprachen gewöhnt, die Modifikationen in textueller Form zu spezifizieren. Das nachfolgende Listing 7.3 enthält die konkrete Syntax der *Refactoring Specification*, in der für jede nicht abstrakte Metaklasse eine textuelle Regel definiert wird.

Listing 7.3: Konkrete Syntax für die textuelle *Refactoring Specification*

```
1  SYNTAXDEF refspec FOR <http://www.emftext.org/language/refactoring_specification>
2  START RefactoringSpecification
3
4  TOKENS{
5    DEFINE INTEGER$('1'..'9')('0'..'9')*|'0'$;
6    DEFINE CONSTANTS $'INPUT'$;
7    DEFINE UPPER_IDENTIFIER $('A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
8    DEFINE LOWER_IDENTIFIER $('a'..'z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*$;
9    DEFINE DOT_NOTATION $($ + UPPER_IDENTIFIER + $|$ + LOWER_IDENTIFIER + $)$ + $'.'
         $ + LOWER_IDENTIFIER;
10 }
11
12 RULES{
13
14   RefactoringSpecification ::= "REFACTORING" "FOR" usedRoleModel['<','>']
15     "STEPS" "{" (instructions ";") + "}" ;
16   // Modellieren von Variablen
17   VariableAssignment ::= "object" variable ":=" assignment ;
18   Variable ::= name[LOWER_IDENTIFIER] ;
19   VariableReference ::= variable[LOWER_IDENTIFIER] ;
20   RoleReference ::= role[UPPER_IDENTIFIER] "from" from ;
21   TRACE ::= role[UPPER_IDENTIFIER] "as" "trace" "(" reference ")" ;
22   FromClause ::= operator "(" reference ")" ;
23   UPTREE ::= "uptree" ;
24   PATH ::= "path" ;
25   FILTER ::= "filter" ;
26   ConstantsReference ::= referencedConstant[CONSTANTS] ;
27   // Modellieren von Indizes
28   IndexVariable ::= name[LOWER_IDENTIFIER] ;
29   FIRST ::= "index" variable ":=" "first" "(" reference ")" ;
30   LAST  ::= "index" variable ":=" "last" "(" reference ")" ;
31   AFTER  ::= "index" variable ":=" "after" "(" reference ")" ;
32   ConcreteIndex ::= "index" variable ":=" index[INTEGER] ;
33   // Erzeugen und Verschieben von Elementen
34   CREATE ::= "create" "new" variable ":" sourceRole[UPPER_IDENTIFIER]
35     "in" targetContext ("at" index[LOWER_IDENTIFIER])? ;
```

```
36   MOVE ::= "move" source "to" target ("at" index[LOWER_IDENTIFIER])? (moveModifier
         )? ;
37   DISTINCT ::= "distinct" ;
38   CollaborationReference ::= collaboration[DOT_NOTATION] ;
39   // Entfernen von Elementen
40   REMOVE ::= "remove" (modifier)? removal ;
41   RoleRemoval ::= role[UPPER_IDENTIFIER] ;
42   UNUSED ::= "unused" ;
43   EMPTY ::= "empty" ;
44   // Aenderung von Elementen
45   ASSIGN ::= "assign" (sourceAttribute[DOT_NOTATION] "for" )? targetAttribute[
         DOT_NOTATION] ;
46   SET ::= "set" "use" "of" source "in" target ;
47 }
```

Der Übersichtlichkeit halber wurde der Abschnitt `OPTIONS` hier nicht mehr mit einbezogen. In den mit `//` gekennzeichneten Kommentaren ist jeweils der Name des Abschnittes aus Kapitel 6.2.2 angegeben.

An dieser Stelle wird nicht mehr im Detail auf die Bestandteile einer konkreten Syntax eingegangen. Dies ist im Zusammenhang mit Listing 7.1 in Kapitel 7.2 schon erfolgt. Allerdings sind die Regeln von `CollaborationReference` und `ASSIGN` noch einmal zu erwähnen. Wie in Kapitel 2.4 schon erläutert, wird die statische Semantik mit Hilfe von Resolvern umgesetzt. Diese werden anhand einer konkreten Syntax in EMFText automatisch für jede referenzierende Metaklasse generiert – so auch für `ASSIGN` und `CollaborationReference`. Im obigen Listing ist für beide Regeln zu erkennen, dass mittels der Punkt-Notation entweder eine Kollaboration oder aber Rollen-Attribute referenziert werden. Da aus diesen beiden Regeln insgesamt drei sich ähnelnde Resolver generiert werden, wurde der gemeinsame Code in einen abstrakten Resolver ausgelagert und der spezifische Code jeweils mittels einer *Template Method* in den konkreten Resolvern umgesetzt [GHJV04, S. 366ff]. Dies ist der Grund, warum `Collaboration` und `RoleAttribute` Unterklassen der Metaklasse `RoleFeature` sind, welche in Abschnitt 6.2.1 für dieses Kapitel zurückgestellt wurde.

Die Erklärung der vier in Abbildung 6.5 dargestellten Operationen der *Refactoring Specification* ist bisher auch noch offen. Wie in Abschnitt 6.2.2 schon erwähnt, dienen diese in der späteren Interpretation einer *Refactoring Specification* der Vereinfachung des Zugriffes auf Variablen und Anweisungen. In der folgenden Auflistung werden alle Operationen genannt und kurz erläutert.

isRoleReferencedByObject(Role,EObject):EBoolean Zur Überprüfung, ob die übergebene Rolle im ersten Parameter von dem Objekt des zweiten Parameters referenziert wird. Der zweite Parameter ist üblicherweise ein Element einer *Refactoring Specification*.

getInstructionsReferencingRole(Role):Instruction Alle Anweisungen einer *Refactoring Specification* werden ermittelt, von denen die übergebene Rolle referenziert wird. Dabei wird intern die zuvor beschriebene Operation benutzt.

getDeclaredVariables():Variable Ermittelt alle in einer *Refactoring Specification* deklarierten Variablen.

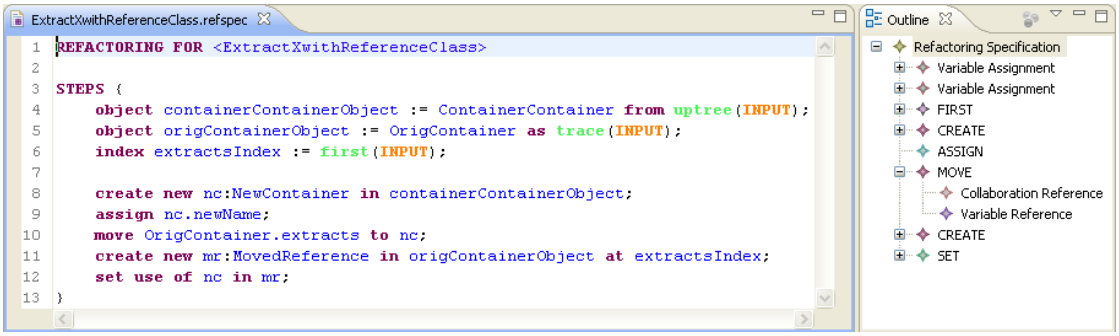


Abbildung 7.5: Graphischer Editor für *Role Models*

getVariableByName(EString):Variable Anhand des übergebenen Strings wird die dementsprechend benannte Variable zurückgegeben.

Neben den zuvor erläuterten Resolvern und Operationen der *Refactoring Specification* wurde mit EMFText außerdem wieder ein Editor generiert. Wie im vorigen Abschnitt 7.2 schon beschrieben, besitzt dieser alle Eigenschaften, die man als Benutzer einer DSL erwartet. In der Abbildung 7.5 ist der Editor mit seiner Outline für die im generischen Modell-Refactoring *Extract X with Reference Class* durchzuführenden Schritte, die sich an *Extract Method* orientieren, dargestellt. Zum besseren Verständnis sei nochmals auf Abbildung 7.3 verwiesen, in der alle für dieses Modell-Refactoring nötigen Rollen, Attribute und Kollaborationen dargestellt sind. Zuerst werden hier das `containerContainerObject` und das `origContainerObject` ermittelt. Danach wird der Index des ersten Elementes der Eingabe bestimmt und in der Variablen `extractsIndex` abgelegt. Die nachfolgenden fünf Anweisungen definieren die im Modell-Refactoring durchzuführenden Umstrukturierungen. Dort wird mit der Erzeugung des Modell-Elementes begonnen, das die Rolle `NewContainer` spielt. Diesem wird in der nächsten Zeile ein Name zugewiesen, der wiederum während der Durchführung vom Anwender erfragt wird. Danach werden die Spieler der Rolle `Extract` in den neuen Container verschoben. Anschließend wird das Element erzeugt, das der Rolle `MovedReference` zugewiesen wurde. Als Position wird dabei der zuvor ermittelte Index des ersten Elementes verwendet. Abschließend beinhaltet die letzte Zeile das Setzen der Referenz auf den neu erzeugten Container. Da die Rolle `MovedReference` als optional markiert wurde, werden die letzten beiden Anweisungen nur durchgeführt, wenn diese Rolle auch auf eine Klasse im Ziel-Metamodell abgebildet wurde. Damit ist der Teil der *Refactoring Specification*, der sich in Abbildung 7.1 im *Metamodel Layer* befindet, vollständig. Darüber hinaus wurde auch hier eine *RefSpec Registry* implementiert, um den Refactoring-Designer bei der Bereitstellung seiner generischen Modell-Refactorings zu unterstützen. Diese befindet sich im *Registry Layer* und wird im Folgenden erläutert.

Die Umsetzung erfolgte analog zur *Role Registry* aus Kapitel 7.2. Über einen zur Verfügung gestellten Erweiterungspunkt müssen Erweiterungen lediglich eine Datei anmelden, in der eine *Refactoring Specification* spezifiziert wurde. Auch hier wurde der Erweiterungspunkt wieder in eine *Registry* gekapselt, welche den programmatischen Zu-

griff erlaubt. Im nachfolgenden Listing 7.4 ist das zugehörige Interface dargestellt.

Listing 7.4: Interface für den Zugriff auf die *RefSpec Registry*

```
1  public interface IRefactoringSpecificationRegistry {
2
3    public static final IRefactoringSpecificationRegistry INSTANCE = new
         BasicRefactoringSpecificationRegistry();
4
5    public RefactoringSpecification getRefSpec(RoleModel roleModel);
6
7    public Collection<RefactoringSpecification> getAllRefSpecs();
8
9    public void registerRefSpec(RefactoringSpecification refSpec) throws
         RefSpecAlreadyRegisteredException;
10
11   public void updateRefSpec(RefactoringSpecification refSpec);
12
13 }
```

Damit wird nicht nur die Abfrage aller registrierten *Refactoring Specifications*, sondern auch die zu einem *Role Model* gehörende *Refactoring Specification* ermöglicht. Außerdem gestattet das Interface die Registrierung und die Aktualisierung. Mit der *RefSpec Registry* ist auch der in Abbildung 7.1 rechts unten dargestellte Teil vollständig beschrieben. Im nächsten Kapitel wird nun die Umsetzung des *Role Mapping Models* erläutert, wodurch dann der Grundstein für die Interpretation von Modell-Refactorings gelegt ist.

7.4 Role Mapping Model

In einem *Role Mapping Model* werden nun die in einem *Role Model* modellierten Rollen und Kollaborationen auf die Metaklassen eines konkreten Metamodells abgebildet. Dieses Modell muss von einem DSL-Designer erstellt werden, da nur er wissen kann, welche Modell-Refactorings für welche Strukturen seiner DSL sinnvoll sind. Auch für das in Abbildung 6.13 dargestellte Metamodell wurde eine textuelle Syntax mit EMFText umgesetzt, mit der die Abbildung von *Role Model* auf Ziel-Metamodell sehr bequem durchführbar ist. Die textuelle Repräsentation wurde für dieses Modell gewählt, da so der Benutzer mit Hilfe der implementierten Resolver und der damit verbundenen *Code Completion* leicht auf die *Role Model Registry* und die in der Eclipse-Plattform verfügbaren Metamodelle zugreifen kann.

Listing 7.5: Konkrete Syntax für *Role Mapping Models*

```
1  SYNTAXDEF rolemapping FOR <http://www.emftext.org/language/rolemapping>
2  START RoleMappingModel
3
4  TOKENS {
5    DEFINE IDENT $('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'-') + ('.'('A'..'Z'|'a'..'z
         '|'0'..'9'|'_'|'-')+)*$;
6  }
7
8  RULES {
9
10   RoleMappingModel::= "ROLEMODELMAPPING" "FOR" targetMetamodel['<','>'] mappings+;
11
```

```
12    RoleMapping ::= name['"','"'] "maps" mappedRoleModel['<','>'] "{"
13      roleToMetaelement+  "}";
14
15    ConcreteMapping ::= role[IDENT] ":=" metaclass[IDENT] ("(" attributeMappings
          ("," attributeMappings)* ")")?
16      ("{" collaborationMappings collaborationMappings* "}")? ";";
17
18    CollaborationMapping ::= collaboration[IDENT] ":=" referenceMetaClassPair ("->"
          referenceMetaClassPair)* ";";
19
20    ReferenceMetaClassPair ::= reference[IDENT] (":" metaClass[IDENT])?;
21
22    AttributeMapping ::= roleAttribute[IDENT] "->" classAttribute[IDENT];
23  }
```

Im Vergleich zu den dargestellten Metaklassen in Abbildung 6.13 enthält obiges Listing nur sechs Regeln. Der Grund dafür ist, dass nur solche Regeln für Metaklassen definiert werden, die auch in dem entsprechenden Metamodell definiert sind. Die in Abbildung 6.13 auf der linken und rechten Seite dargestellten Metaklassen befinden sich jedoch physisch im jeweiligen Metamodell vom *Role Model*, respektive von Ecore. Die entsprechenden Instanzen werden dann wieder mit Hilfe der Resolver aufgelöst.

Im Abschnitt 6.2.3 wurden außerdem die vier Operationen der Metaklasse `RoleMapping` angesprochen. Diese dienen dem einfacheren Zugriff auf Rollen und ihnen zugewiesene Metaklassen für den Prozess der Interpretation eines Modell-Refactorings. Nachfolgend werden die Operationen genannt und kurz erläutert.

getAllMappedRoles():Role Ermittelt alle Rollen, die auf Metaklassen abgebildet wurden. Diese Funktionalität wird benötigt, da optionale Rollen nicht abgebildet zu werden brauchen.

getMappedRolesForEObject(EObject):Role Anhand der Metaklasse des übergebenen Objekts werden alle Rollen ermittelt, die von dieser Metaklasse im Kontext eines Modell-Refactorings gespielt werden.

getEClassForRole(Role):EClass Es wird die Metaklasse zurückgegeben, die Spieler der übergebenen Rolle ist.

getConcreteMappingForRole(Role):ConcreteMapping Ein `RoleMapping` enthält mehrere Instanzen von `ConcreteMapping`, in denen jeweils eine Rolle auf eine Metaklasse abgebildet wird. Mit dieser Operation wird die eine Rolle abbildende Instanz ermittelt.

Damit ist der in Abbildung 7.1 dargestellte Teil *Role Mapping Model* aus dem *Metamodel Layer* vollständig. Da Kollaborationen zwischen Rollen aber Bedingungen zwischen Metaklassen modellieren, müssen diese auch in einer Abbildung von *Role Model* auf Metamodell geprüft werden. Dazu dient die Komponente *Role Constraint Checker*. Diese wurde abgegrenzt, um die interne Implementierung einfach anpassen zu können.

Der *Role Constraint Checker* wird immer nach dem Auflösen aller Referenzen mittels der Resolver aufgerufen, damit sichergestellt ist, dass keine ungültigen Referenzen existieren. Geprüft werden Kollaborationen wie *Role Implication* und *Role Prohibition*.

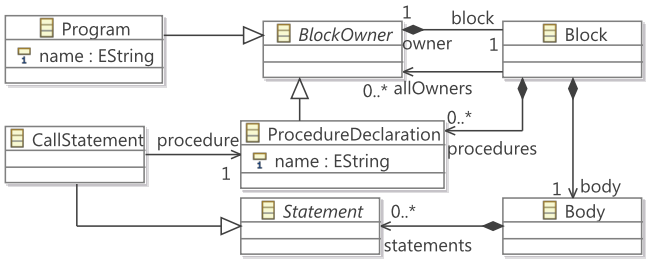


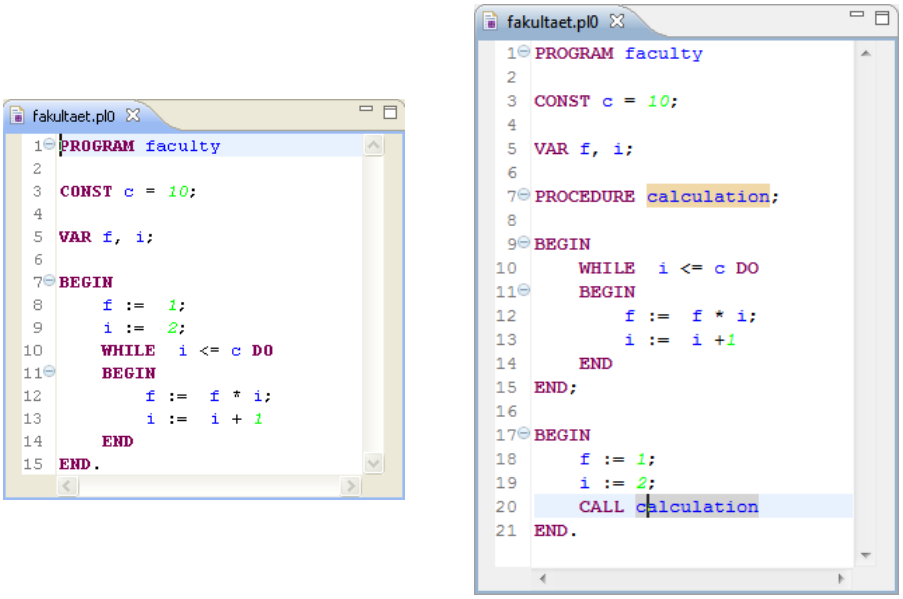
Abbildung 7.6: Ausschnitt aus dem Metamodell von PL/0

Dazu werden vom *Role Constraint Checker* die den Rollen zugewiesenen Metaklassen abgefragt und anschließend überprüft, ob diese den modellierten Kollaborationen entsprechen. Beispielsweise dürfen keine zwei Rollen, die mit einer *Role Prohibition* in Verbindung stehen, von derselben Metaklasse gespielt werden. Umgekehrt impliziert eine *Role Implication* zwischen zwei Rollen, dass diese von derselben Metaklasse zu spielen sind.

Als Beispiel für ein *Role Mapping Model* wird zuerst eine kleine Beispiel-DSL vorgestellt, da eine Abbildung sonst nur schwer verständlich ist. Dazu wurde die Sprache PL/0 gewählt, die von Niklaus Wirth in seinem Buch [Wir86] als Beispielsprache benutzt wird, um den vollständigen Prozess der Erstellung eines Compilers zu durchlaufen. PL/0 ist eine an Pascal angelehnte prozedurale Programmiersprache, in der es nur den Typ der Ganzzahlen, sowie Kontrollstrukturen, Prozeduren und arithmetische Variablenzuweisungen gibt. Für diese Sprache wurde ein Metamodell entwickelt, von dem in Abbildung 7.6 nur der für das generische Modell-Refactoring *Extract X With Reference Class* benötigte Teil dargestellt ist. Auf Einzelheiten soll hier nicht näher eingegangen werden. Nähere Informationen sind dem Buch [Wir86] zu entnehmen.

Zur Erinnerung an die Syntax zeigt Abbildung 7.7(a) ein PL/0-Beispielprogramm, in dem die Fakultät von 10 berechnet wird. Der dort abgebildete Editor wurde ebenfalls mit EMFText auf Basis einer textuellen Syntax generiert, die exakt der Syntax von Wirth entspricht. Es wird deutlich, dass das generische Modell-Refactoring *Extract X With Reference Class* für PL/0 durchaus sinnvoll ist, da auch hier das Design verbessert werden kann, wenn eine Gruppe von Anweisungen in eine eigene Prozedur verschoben wird. Abbildung 7.7(b) zeigt beispielsweise die Auslagerung der Schleife in die Prozedur `calculation`.

Um dies zu ermöglichen, wurde, wie zuvor schon erläutert, aus der in Listing 7.5 dargestellten konkreten Syntax für das *Role Mapping Model* ein textueller Editor mit EMFText generiert. Dieser ist in Abbildung 7.8 zu sehen und enthält das *Role Mapping Model* von *Extract X With Reference Class* auf das PL/0-Metamodell. Diese Abbildung zeigt, dass das generische Modell-Refactoring *Extract X With Reference Class* im Metamodell von PL/0 *Extract Procedure* genannt wird. Dieser Name wird später auch in der Benutzerschnittstelle zur Ausführung des Modell-Refactorings angezeigt. Bei der Abbil-



(a) vor *Extract Procedure* (b) nach *Extract Procedure*

Abbildung 7.7: PL/0-Beispielprogramm *faculty*

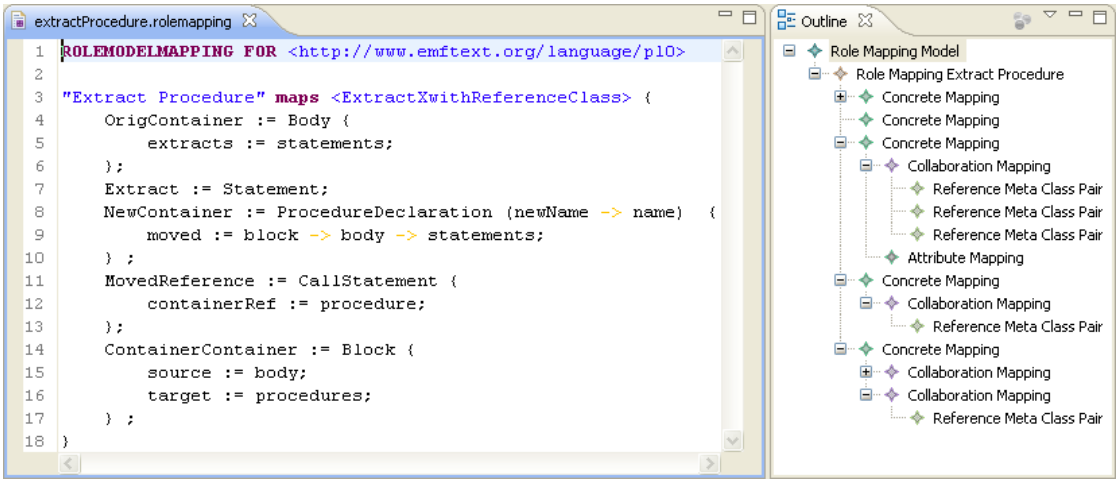


Abbildung 7.8: *Role Mapping Model* des generischen Modell-Refactorings *Extract X With Reference Class* auf das Metamodell von PL/0

Listing 7.6: Interface für den Zugriff auf die *Role Mapping Registry*

```
1  public interface IRoleMappingRegistry {
2
3    public static final IRoleMappingRegistry INSTANCE = new BasicRoleMappingRegistry
         ();
4
5    public Map<String, RoleMapping> getRoleMappingsForUri(String nsUri);
6
7    public List<RoleMapping> registerRoleMapping(RoleMappingModel roleMapping);
8
9    public void updateMappings(List<RoleMapping> mappingsToUpdate);
10
11   public Map<String, Map<String, RoleMapping>> getRoleMappingsMap();
12
13   public void registerPostProcessor(RoleMapping mapping, IRefactoringPostProcessor
         postProcessor);
14
15   public IRefactoringPostProcessor getPostProcessor(RoleMapping mapping);
16
17   public ImageDescriptor getImageForMapping(RoleMapping mapping);
18 }
```

dung der Rolle `NewContainer` auf die Metaklasse `ProcedureDeclaration` ist auch zu sehen, wie eine Kollaboration einem Pfad im Metamodell zugewiesen wird.

Zur Erinnerung an das verwendete *Role Model* sei nochmals auf Abbildung 7.4 verwiesen. Dort ist zu sehen, dass zwischen `NewContainer` und `Extract` eine *Role Composition* existiert. In Abbildung 7.8 ist zu sehen, dass diese Rollen auf `ProcedureDeclaration` und `Statement` abgebildet wurden. Deshalb reflektiert die Zuweisung `moved := block -> body -> statements;` den Pfad im Metamodell aus Abbildung 7.6, ausgehend von der Metaklasse `ProcedureDeclaration` zu `Statement`. Damit sind die Komponenten zur Definition von *Role Mapping Models* und der Überprüfung der Bedingungen aus dem *Metamodel Layer* in Abbildung 7.1 abgeschlossen. Nun wird die *Registry* beschrieben, die auch hier die Möglichkeit der Bereitstellung von *Role Mapping Models* und des Zugriffes auf diese sicherstellt.

Mit der Erstellung von *Role Mapping Models* sind prinzipiell alle Voraussetzungen für die Durchführung von Modell-Refactorings geschaffen. Um sie aber in der Eclipse-Plattform verfügbar zu machen, wurde wieder ein Erweiterungspunkt entworfen, an dem *Role Mapping Models* registriert werden können. Damit auf die *Role Mapping Registry* programmatisch zugegriffen werden kann, wurde die Funktionalität auch hier hinter einem Interface gekapselt. Dieses wird im Listing 7.6 dargestellt. Die ersten vier Methoden erinnern wieder an die anderen beiden *Registries*. Mit der Methode `getRoleMappingsForUri` können alle *Role Mapping Models* abgefragt werden, die für das Metamodell registriert wurden, das mit dem Parameter `nsUri` identifiziert wird. Zurückgegeben wird eine *Map*, deren Schlüssel die Namen der einzelnen *Role Mapping Models* sind. Außerdem können *Role Mapping Models* registriert und aktualisiert werden oder es kann die komplette *Registry* zurückgegeben werden.

Die Methoden `registerPostProcessor` und `getPostProcessor` ermöglichen die Umsetzung der Anforderung 3 von Seite 30. In dieser Anforderung wird die Notwendigkeit

Listing 7.7: Interface für Post-Prozessoren

```
1 public interface IRefactoringPostProcessor {
2   public IStatus process(Map<Role, List<EObject>> roleRuntimeInstanceMap,
        ResourceSet resourceSet, ChangeDescription change);
3 }
```

erläutert, neben der generischen Definition von Modell-Refactorings auch Sprachspezifika einzubinden. Der Grund dafür ist, dass ein generisches Modell-Refactoring nicht immer alle Belange, die im Kontext eines Ziel-Metamodells wichtig sind, umsetzen kann. Dazu zählt beispielsweise beim Modell-Refactoring *Extract Method* in Java, dass die verschobenen Anweisungen dahingehend untersucht werden müssen, ob diese auf in der ursprünglichen Methode lokal definierte Variablen zugreifen. In diesem Fall müssen der neuen Methode zusätzliche Parameter hinzugefügt werden, damit die lokalen Variablen beim Aufruf der neuen Methode übergeben werden können. Außerdem muss die extrahierte Methode als statisch definiert werden, wenn die ursprüngliche Methode auch statisch ist. Ist eine der zu verschiebenden Anweisungen eine Variablenzuweisung, auf die nachfolgend zugegriffen wird, so muss für die extrahierte Methode darüber hinaus ein Rückgabetyp spezifiziert werden, der dem der zugewiesenen Variablen entspricht.

Für diesen Zweck wurde in der *Role Mapping Registry* zusätzlich der Erweiterungspunkt *Post Processor* definiert. Damit können Erweiterungen registriert werden, die es ermöglichen, nach einem generischen Modell-Refactoring weitere sprachspezifische Umstrukturierungen durchzuführen. Um dies zu erreichen, muss einem konkreten *Role Mapping* eines Metamodells der Post-Prozessor zugewiesen werden können. Zu diesem Zweck muss eine Erweiterung den Identifikator des Metamodells sowie den Namen des konkreten *Role Mappings* angeben. Somit ist das *Role Mapping* eindeutig identifiziert und die Erweiterung muss zusätzlich noch auf eine Implementierung des Interfaces `IRefactoringPostProcessor` verweisen. Darin werden die sprachspezifischen Umstrukturierungen implementiert. Dieses Interface enthält nur eine Methode und ist im Listing 7.7 abgebildet. Anhand der darin übergebenen Parameter kann in der Implementierung auf die schon durchgeführten Änderungen, auf mit dem jeweiligen Modell in Beziehung stehende andere Modelle oder aber auf die den Rollen zur Laufzeit zugewiesenen Objekte zugegriffen werden. Der Parameter `roleRuntimeInstanceMap` enthält eine *Map*, mit der die Rollen zu den Objekten aufgelöst werden. Über `resourceSet` können alle Modell-Ressourcen ermittelt werden, die von dem durchgeführten Modell-Refactoring beeinflusst werden. Der Parameter `change` enthält eine Beschreibung der Änderungen, die bis zum Zeitpunkt des Aufrufs des Post-Prozessors durchgeführt wurden. So kann auf die bisherigen Änderungen reagiert und es können abhängig davon weitere Modifikationen durchgeführt werden. Als Beispiel wird in Listing 7.8 der Post-Prozessor für *Extract Method* in Java dargestellt. Der Übersichtlichkeit halber werden hier in der Methode `processMethodType` lediglich `void` als Rückgabetyp der neuen Methode gesetzt und diese als `static` deklariert. Es ist aber trotz der Kürze sehr gut zu erkennen, wie mittels der generierten API weitere sprachspezifische Umstrukturierungen durchzuführen sind.

Listing 7.8: Post-Prozessor für *Extract Method* in Java

```
1  public class JavaExtractMethodPostProcessor implements IRefactoringPostProcessor {
2
3    private ClassMethod newContainer;
4    private ClassMethod origContainer;
5
6    public IStatus process(Map<Role, List<EObject>> roleRuntimeInstanceMap,
         ResourceSet resourceSet, ChangeDescription change) {
7      newContainer = determineNewContainer(roleRuntimeInstanceMap);
8      origContainer = determineOrigContainer(roleRuntimeInstanceMap);
9      boolean result = processMethodType();
10     if(!result){
11       return new Status(IStatus.WARNING, Activator.PLUGIN_ID, "Couldn't␣determine␣
             the␣modifier");
12     }
13     return Status.OK_STATUS;
14   }
15
16   private Boolean processMethodType(){
17     Void voidType = TypesFactory.eINSTANCE.createVoid();
18     newContainer.setTypeReference(voidType);
19     List<AnnotationInstanceOrModifier> modifiers = origContainer.
           getAnnotationsAndModifiers();
20     for (AnnotationInstanceOrModifier modifier : modifiers) {
21       if(modifier instanceof Static){
22         Static staticModifier = ModifiersFactory.eINSTANCE.createStatic();
23         return newContainer.getAnnotationsAndModifiers().add(staticModifier);
24       }
25     }
26     return true;
27   }
28 }
```

Die Methoden `determineNewContainer` und `determineOrigContainer` sind in obigem Listing auch nicht dargestellt, da sie keine neuen Erkenntnisse hervorbringen. Sie iterieren lediglich über die Menge der Rollen der übergebenen `roleRuntimeInstanceMap` und geben die entsprechenden Laufzeit-Objekte des Modell-Refactorings zurück.

Die letzte in Listing 7.6 dargestellte Methode `getImageForMapping` hat Auswirkungen auf die spätere Integration in die Benutzerschnittstelle. Der Erweiterungspunkt der *Role Mapping Registry* ermöglicht es außer der Registrierung von *Role Mapping Models* und Post-Prozessoren, kleine Symbole für konkrete *Role Mappings* zu registrieren. Diese können dann in den Menüs, über die Modell-Refactorings aufzurufen sind, angezeigt werden. Somit erhöht sich der Wiedererkennungswert und Modell-Refactorings können schneller gefunden werden.

Damit ist die Vorstellung der Ebenen *Metamodel Layer* und *Registry Layer* aus Abbildung 7.1 vollständig. Mit den bis hierher vorgestellten Komponenten von *Refactory* ist es nun möglich, Modell-Refactorings generisch zu spezifizieren, diese auf konkrete Metamodelle abzubilden und sprachspezifische Erweiterungen zu integrieren. Was nun noch fehlt, ist die Ausführung. Wie dies vonstattengeht, erläutert der nächste Abschnitt.

7.5 Durchführung eines Modell-Refactorings durch Interpretation

Mit der bis hierhin erläuterten Umsetzung des in Kapitel 6 vorgestellten Konzepts ist die generische Spezifikation von Modell-Refactorings mit sprachspezifischen Erweiterungen möglich. Nun gilt es, derartige Spezifikationen auch in Modellen eines Metamodells durchzuführen. Prinzipiell kommen zwei Arten der Ausführung in Betracht: die Kompilation und die Interpretation. Erstere zeichnet sich durch die Übersetzung des generischen Modell-Refactorings im Kontext des jeweiligen Ziel-Metamodells in eine Sprache aus, die von der Ausführungsumgebung (dem Betriebssystem oder der virtuellen Maschine) ausgeführt werden kann. Das bedeutet, dass aus der generischen Spezifikation für ein Ziel-Metamodell ein eigenständiges Plugin generiert wird, in dem das Modell-Refactoring beispielsweise in Form von Java-Code ausgeführt werden kann. Der Vorteil dieses Vorgehens ist, dass die Ausführungsgeschwindigkeit sehr hoch ist, da ein Modell-Refactoring direkt von der Umgebung ausgeführt wird und keine weiteren Zwischenschritte benötigt werden. Dies bedeutet jedoch auch, dass für die Generierung des Kompilats Übersetzungsregeln benötigt werden, die festlegen, in welcher Art und Weise der Code erzeugt werden soll. Außerdem müssen zu Testzwecken immer erst die Kompilation durchgeführt, das daraus erzeugte Plugin der Eclipse-Plattform verfügbar gemacht und diese neu gestartet werden.

Bei der Interpretation handelt es sich um die schrittweise Ausführung einer *Refactoring Specification*. Das bedeutet, dass ein Interpreter die einzelnen definierten Schritte eines Modell-Refactorings direkt ausführen kann, ohne dass die Generierung von Code benötigt wird. Nachteilig kann sich bei diesem Vorgehen die geringere Ausführungsgeschwindigkeit auswirken. Der Grund dafür ist, dass zwischen dem zu refaktorierenden Modell und der *Refactoring Specification* immer die Komponente der Interpretation angesiedelt ist. Dadurch kann eine leichte Verzögerung eintreten. Entscheidender Vorteil allerdings ist, dass ein Modell-Refactoring mit der Fertigstellung der Spezifikation und eines konkreten *Role Mapping Models* direkt in der ausführenden Eclipse-Plattform gestartet werden kann. So ist es möglich, Modell-Refactorings sofort zu testen. Aus diesem Grund wird für die ausführende Komponente vorzugsweise die Interpretation gewählt, welche es nun zu erläutern gilt.

Bei der Implementierung des Interpreters erhält man von EMFText ein wenig Unterstützung. Dabei erzeugt EMFText für jede definierte konkrete Syntax auch eine Klasse, die eine Rahmen-Implementierung für einen abstrakten Interpreter darstellt. Diese enthält für jede Klasse des Metamodells eine Methode, die bei Bedarf dann nur noch überschrieben und den eigenen Wünschen angepasst zu werden braucht. Hierbei ist wichtig zu erwähnen, dass diese abstrakte Interpreter-Klasse keine Abhängigkeiten zu EMFText selbst besitzt, sondern als Bonus anzusehen ist, der weiterverarbeitet werden kann. Der Interpreter benutzt für die interne Verarbeitung der Anweisungen einen Stack. Dieser muss in Abhängigkeit von einer konkreten *Refactoring Specification* initialisiert werden, da darin die durchzuführenden Schritte spezifiziert sind. Im Falle des Refactoring-Interpreters wird der Stack nur mit Instanzen der Unterklassen der abstrakten Meta-

klasse *Instruction* (siehe Abbildung 6.5) gefüllt. Derart wird dann immer das im Stack am weitesten oben liegende Element interpretiert und vom Stack entfernt. Während der Interpretation wird außerdem ein Kontext entwickelt, in dem alle zu einer Variablen aufgelösten Objekte der Laufzeit gespeichert werden. Dadurch existiert während der Interpretation eines Modell-Refactorings in einem konkreten Modell ein Kontext, auf den stets zugegriffen und der in jedem Interpretations-Schritt verändert werden kann. Auf diese Weise können nun alle Anweisungen einer *Refactoring Specification* anhand der Abbildung der Rollen auf das jeweilige Ziel-Metamodell durchgeführt werden. Die generische Beschreibung der durchzuführenden konkreten Transformationen im jeweiligen zu refaktorierenden Modell wurde mit der reflektiven API von Ecore selbst umgesetzt. Diese gestattet es beispielsweise, anhand der in einem `ReferenceMetaClassPair` verwiesenen Referenz des Ziel-Metamodells, die im konkreten Modell der Referenz entsprechenden Elemente zurückzugeben. Auf diese Weise wird die Navigation entlang der so modellierbaren Pfade in dem Plugin *Graph Utility* vom *Interpreter Layer* aus Abbildung 7.1 umgesetzt.

Um die in einem Modell-Refactoring fehlenden Informationen, beispielsweise die Angabe von Namen für neue Elemente, zu integrieren, werden in einem ersten Probelauf alle anzugebenden Daten gesammelt, welche dann im eigentlichen Interpretations-Durchgang vom Nutzer erfragt werden. Ein weiterer Vorteil dieser Vorgehensweise ist es, dass so vor der eigentlichen Durchführung des Modell-Refactorings eine Vorschau auf die zu erwartenden Änderungen erfolgen kann.

Damit wurde in diesem Kapitel neben der Umsetzung des Plugins *Graph Utility* auch der *RefSpec Interpreter* gemäß der Abbildung 7.1 im *Interpreter Layer* erläutert. Wie in Kapitel 6.5 schon diskutiert, wird für die Wahrung der horizontalen und vertikalen Konsistenz ein Index-Mechanismus benötigt. Dieser wird im folgenden Abschnitt vorgestellt.

7.6 Index-Mechanismus zur Wahrung der Konsistenz

Zur Wahrung der horizontalen und vertikalen Konsistenz der Modelle, die auf das refaktorierte Modell verweisen oder von diesem referenziert werden, wird ein Index-Mechanismus benötigt. In Kapitel 6.5 wurde schon erläutert, dass das Auffinden der Verweise des refaktorierten Modells auf andere Modelle kein Problem darstellt, da diese Richtung mit EMF umgesetzt werden kann. Um das Problem der umgekehrten Richtung ebenfalls zu lösen, wurde hier die Anbindung von Indizes vorgesehen. Damit kann das Ermitteln der Modelle, die auf das refaktorierte verweisen, in die Implementierung des Indexes ausgelagert werden. Auf diese Weise ist es möglich, durch den Index alle Ressourcen zurückzugeben, die Abhängigkeiten zu einem beliebigen Element des refaktorierten Modells besitzen. Zu diesem Zweck wurde wieder ein Erweiterungspunkt definiert, an dem beliebige Index-Implementierungen registriert werden können. Eine Erweiterung muss dazu eine Realisierung des Interfaces `IndexConnector` aus Listing 7.9 liefern. Implementierungen der einzigen dort deklarierten Methode `getReferencingResources` müssen, wie beschrieben, alle Ressourcen zurückgeben, die auf den übergebenen Parame-

Listing 7.9: Interface zur Anbindung von Indizes

```
1 public interface IndexConnector {
2
3   public List<Resource> getReferencingResources(EObject referenceTarget);
4 }
```

ter `referenceTarget` verweisen. Bei `referenceTarget` handelt es sich um ein beliebiges Element des refaktorierten Modells.

Des Weiteren wurde zur Sicherstellung der Ermittlung möglichst aller referenzierenden Ressourcen eine *Registry* umgesetzt, an der mehrere *Index Connectors* registriert werden können. Diese dient außerdem dem programmatischen Zugriff des *RefSpec Interpreters* aus dem vorangegangen Kapitel, um alle referenzierenden Ressourcen ausfindig zu machen. Mit dem in Listing 7.10 dargestellten Interface wird auf die *Index Connector Registry* zugegriffen.

Listing 7.10: Interface der *Index Connector Registry*

```
1  public interface IndexConnectorRegistry {
2
3    public static IndexConnectorRegistry INSTANCE = new BasicIndexConnectorRegistry
         ();
4
5    public List<IndexConnector> getConnectors();
6
7    public void registerConnector(IndexConnector connector);
8
9    public List<Resource> getReferencingResources(EObject referenceTarget);
10 }
```

Mit Hilfe dieses Interfaces können *Index Connectors* sowohl zurückgegeben als auch registriert werden. Außerdem existiert hier wieder eine Methode `getReferencingResources`. Diese ermittelt kumulativ alle Ressourcen, die auf das refaktorierte Modell verweisen. Das heißt konkret, dass alle registrierten *Index Connectors* nach Referenzen gefragt werden und daraus dann die Vereinigungsmenge gebildet wird. Zum Zeitpunkt der Implementierung dieses Konzeptes konnte der Index *Sokan* aus dem Projekt Reuseware[2], in dem ein Framework zur invasiven Software-Komposition entwickelt wurde, erfolgreich angebunden werden. Doch auch andere Index-Implementierungen, wie beispielsweise das jüngst ins Leben gerufene Projekt *EMF Index*[3], können mit Hilfe der erläuterten *Index Connector Registry* integriert werden.

Auf diese Weise können alle direkt und invers referenzierten Ressourcen ermittelt und nach einem Modell-Refactoring automatisch angepasst werden. So wird die horizontale Konsistenz gesichert. Wird ein Metamodell dynamisch instanziiert, kann derart auch die vertikale Konsistenz garantiert werden, wie schon in Kapitel 6.5 diskutiert. Auf diese Weise konnte die Komponente *Index Connector Registry* aus Abbildung 7.1 erfolgreich umgesetzt werden und der *Interpreter Layer* ist nun vollständig. Schlussendlich fehlt

[2] `www.reuseware.org`
[3] `http://wiki.eclipse.org/EmfIndex`

jetzt noch die Anbindung von Modell-Refactorings an die Benutzerschnittstelle. Erläuterungen dazu folgen im nächsten Kapitel.

7.7 Kopplung an die UI

Ein wichtiges Kriterium für die Validierung des in dieser Arbeit entwickelten Konzepts ist die Akzeptanz des Anwenders von Modell-Refactorings. In Anforderung 12 auf Seite 32 wurde erläutert, dass sich die Modell-Refactorings möglichst nahtlos in die IDE integrieren und bezüglich der Benutzerführung an den schon vorhandenen Code-Refactorings orientieren müssen. In der Eclipse-Plattform, in der das Konzept implementiert wurde, werden Code-Refactorings mit Hilfe des Eclipse Language Toolkit (LTK)[4] umgesetzt. Dabei wird der Benutzer durch einen Wizard geführt, in dem alle fehlenden Benutzereingaben vorgenommen werden, sowie eine Vorschau über die vorzunehmenden Umstrukturierungen verfügbar ist. Zu jedem Zeitpunkt hat der Benutzer die Wahl, das Code-Refactoring abzubrechen, wodurch keine Änderungen durchgeführt werden, sowie im Wizard vor und zurück zu navigieren. Ein Dialog zum Refaktorieren von Code wird in Eclipse derart initiiert, dass zuerst die zu verändernden Code-Zeilen im jeweiligen Editor selektiert werden und anschließend das gewünschte Refactoring per Kontextmenü gestartet wird. Daraufhin öffnet sich der beschriebene Wizard. Diese Benutzerführung gilt demnach als Vorgabe, da der Anwender an das erläuterte Vorgehen gewöhnt ist und ihm nicht das Gefühl der Umstellung vermittelt werden soll. Aus diesen Gründen wurde die in den vorangegangenen Kapiteln erläuterte Implementierung vollständig auf Basis des LTK von Eclipse umgesetzt. Dazu wurde die Klasse `ModelRefactoring` als Erweiterung von `org.eclipse.ltk.core.refactoring.Refactoring` implementiert, in der Vor- und Nachbedingungen geprüft werden, sowie der Name eines Refactorings zurückgegeben wird. Dieser entspricht stets dem Namen, der in einem konkreten *Role Mapping* vergeben wurde, wie beispielsweise *Extract Procedure* am Beispiel von PL/0 aus Kapitel 7.4. Außerdem muss dort angegeben werden, wie die Umstrukturierung berechnet wird. Dazu dient `ModelRefactoringChange` als Erweiterung der Klasse `Change` aus dem Namensraum `org.eclipse.ltk.core.refactoring`. Diese implementiert außerdem das Interface `IModelCompareInputProvider`, welches sich im Namensraum `org.eclipse.emf.compare.ui` befindet, wodurch sichergestellt wird, dass ein Modell-Vergleich mit Hilfe des Frameworks *EMF Compare*[5] in der schon erwähnten Vorschau angezeigt werden kann. Die abstrakte Methode `perform`, die in `ModelRefactoringChange` implementiert werden muss, gibt außerdem wieder ein `Change`-Objekt zurück. Dieses Objekt beschreibt die Aktionen die erforderlich sind, um ein Refactoring rückgängig zu machen. Auf diese Art und Weise kann Anforderung 8 von Seite 31 erfüllt werden. Die Durchführung wurde in `perform` darüber hinaus so realisiert, dass sie in eine Transaktion eingebettet wird. Somit wird neben der Umkehrbarkeit auch Anforderung 7 der Atomarität umgesetzt.

Um den Wizard im Rahmen des LTK auszuführen, wurde die Klasse `ModelRefactoringWizard` als Erweiterung von `RefactoringWizard`, aus dem Namensraum `org.`

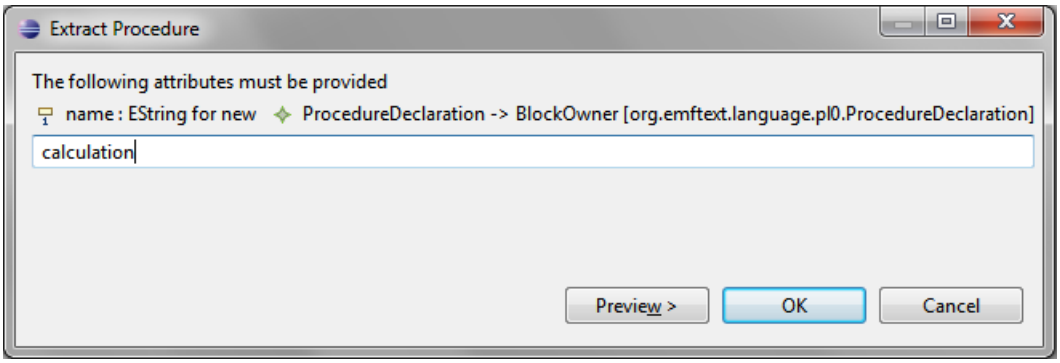
[4] `http://www.eclipse.org/articles/Article-LTK/ltk.html`
[5] `http://www.eclipse.org/modeling/emf/?project=compare`

Listing 7.11: Interface zur Registrierung zusätzlicher *Editor Connectors*

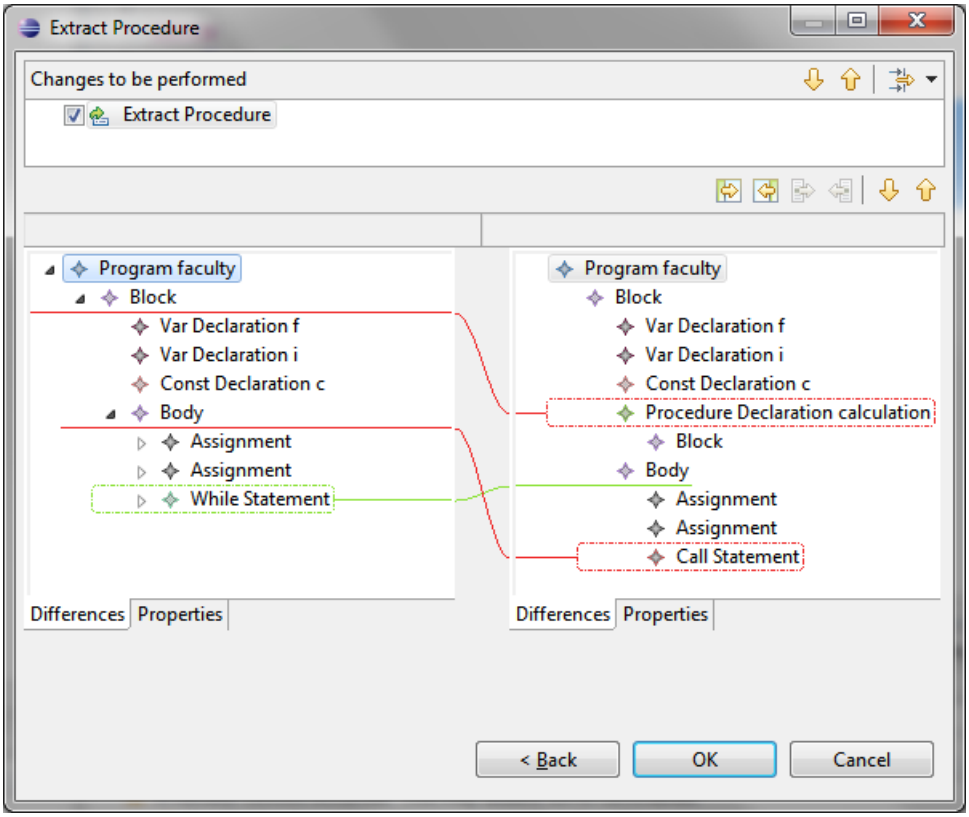
```
1  public interface IEditorConnector {
2
3    public boolean canHandle(IEditorPart editor);
4
5    public List<EObject> handleSelection(ISelection selection);
6
7    public TransactionalEditingDomain getTransactionalEditingDomain();
8
9    public void selectEObjects(List<EObject> objectsToSelect);
10 }
```

`eclipse.ltk.ui.refactoring`, umgesetzt. Der `ModelRefactoringWizard` wird nun mit einer Instanz von `ModelRefactoring` initialisiert, der wiederum ein Interpreter, wie er in Kapitel 7.5 erläutert wurde, und ein *Role Mapping Model* übergeben werden. Im `ModelRefactoringWizard` werden nun die einzelnen anzuzeigenden Seiten hinzugefügt. Diese werden dynamisch in Abhängigkeit von den im Probelauf der Interpretation ermittelten nötigen Benutzereingaben bestimmt. Die Eingabeseiten sind jeweils Unterklassen von `UserInputWizardPage` aus dem Namensraum `org.eclipse.ltk.ui.refactoring`. Abbildung 7.9(a) zeigt die Eingabeseite zu *Extract Procedure* von PL/0, in der der Name (hier `calculation`) der neu anzulegenden Prozedur vom Nutzer erfragt wird. Darüber hinaus zeigt Abbildung 7.9(b) die Vorschauseite von *Extract Procedure* mit der in Abbildung 7.9(a) dargestellten Vergabe des Namens `calculation` für die zu extrahierende neue Prozedur. Das Resultat dieses Modell-Refactorings wurde schon in Abbildung 7.7(b) dargestellt. Auf diese Art und Weise konnte die Umsetzung des generischen Modell-Refactorings mit dem LTK vollständig in Eclipse integriert werden. Somit ist sichergestellt, dass der Anwender mit einem den bekannten Code-Refactorings in Optik und Durchführung sehr ähnlichen Wizard ausgestattet wird.

Um abschließend noch die Unabhängigkeit bezüglich der Modell-Repräsentation der Anforderung 10 zu garantieren, muss überlegt werden, wie Editoren Selektionen verfügbar machen. Das Problem liegt dabei darin, dass anhand einer Selektion die damit ausgewählten Objekte eines Modells zurückgegeben werden müssen. Diese Art der Bereitstellung kann von Editor zu Editor sehr unterschiedlich sein. Mit EMFText generierte Editoren ermitteln beispielsweise die selektierten Modell-Elemente anhand einer internen Abbildung des Offsets in dem textuellen Modell auf das eigentliche Modell-Element. Grafische GMF-basierte Editoren hingegen geben die grafischen Elemente der Selektion zurück, aus denen anschließend über die API des grafischen GMF-Metamodells die Modell-Elemente manuell ermittelt werden müssen. Daran ist zu erkennen, dass es der Zugriffsarten viele gibt, die jeweils sehr spezifisch sein können. Aus diesem Grund wurde der Erweiterungspunkt *Editor Connector* geschaffen, über den Erweiterungen Implementierungen des Interfaces im Listing 7.11 registrieren können. Implementierungen können den Zugriff auf die Modell-Elemente damit sehr Editor-spezifisch umsetzen. Mit `canHandle` muss dazu ermittelt werden, ob ein registrierter *Editor Connector* mit dem übergebenen Editor umgehen, das heißt, ob dieser aus Selektionen des Editors Modell-Elemente ermitteln kann. Die Methode `handleSelection` führt dann die konkrete Auflösung der Modell-



(a) Eingabeseite



(b) Vorschauseite

Abbildung 7.9: Refactoring-Wizard von *Extract Procedure* für PL/0-Modelle

Elemente aus der übergebenen Selektion durch. Mit `getTransactionalEditingDomain` wird eine vom Editor bereitgestellte `TransactionalEditingDomain` zurückgegeben, die im Kontext von EMF-basierten Modell-Transformationen nötig ist, um sie als Transaktion durchführen zu können. GMF-basierte Editoren geben dafür eine spezielle Implementierung zurück, mit der außerdem die Synchronität zwischen dem eigentlichen Modell und seiner grafischen Repräsentation sichergestellt wird.

Die Methode `selectEObjects`, der als Parameter die in dem Modell-Refactoring veränderten oder neu hinzugekommenen Modell-Elemente übergeben werden, wird jeweils nach dem Modell-Refactoring aufgerufen. Implementierungen können anhand dieser Elemente selbst eine Selektion zur optischen Hervorhebung der durchgeführten Änderungen vornehmen. Dieses Verhalten ist beispielsweise aus dem Code-Refactoring *Extract Method* von Java bekannt, in dem danach die neu erzeugte Methode eine Selektion im Texteditor erhält. Dieses Verhalten kann so von einem *Editor Connector* nachempfunden werden.

Damit sind auch die verbliebenen Komponenten *UI Integration* und *Editor Connector Registry* aus Abbildung 7.1 umgesetzt und erläutert worden. Diese befinden sich im *UI Layer* der *Refactory*-Architektur, welcher auf alle darunter liegenden Ebenen zugreift. Zusätzlich konnten weitere Anforderungen aus Kapitel 4 realisiert werden, die vor allem für die Anwendung sehr wichtig sind. Mit den in diesem Kapitel erläuterten Implementierungen konnte validiert werden, dass das Konzept des generischen Modell-Refactorings auch praktisch umsetzbar ist. Bevor die Implementierung im Kapitel 7.9 evaluiert wird, beschreibt das nachfolgende Kapitel die für *Refactory* eingesetzte Testumgebung.

7.8 Testumgebung

Um die Qualität der implementierten Funktionalitäten zu sichern, müssen kontinuierlich Unit-Tests durchgeführt werden. Diese werden in *Refactory* mit dem Test-Framework JUnit[6] automatisch durchgeführt. Für jede Funktionalität wird ein Testfall implementiert, wodurch nach und nach eine vollständige Test-Suite entsteht. Jede einen Test implementierende Methode muss mit `@Test` annotiert werden, damit JUnit per *Reflection* erkennt, dass darin ein Testfall implementiert wurde. Nach jeder Änderung der Implementierung oder dem Hinzufügen neuer Funktionalitäten wird dann die Test-Suite aufs Neue durchlaufen, um sicherzustellen, dass keine Fehler hinzugekommen und die Abhängigkeiten zwischen den Komponenten valide sind. Auf diese Weise wird die Implementierung von *Refactory* selbst getestet. Allerdings muss auch eine Möglichkeit bereitgestellt werden, generisch definierte Modell-Refactorings im sprachspezifischen Kontext der Ziel-Metamodelle zu testen. Zu diesem Zweck wurde die Testumgebung um die Annotation `@InputData` erweitert. Eine Testmethode erhält neben `@Test` auch diese Annotation. An `@InputData` können dann beliebig viele Parameter vom Typ `String` übergeben werden, die jeweils für ein bestimmtes Muster der Dateinamen stehen, welche die Testdaten enthalten. Die mit dieser Testmethode zu testenden Modelle müssen im Namen stets die festgelegten Muster enthalten, damit erkannt werden kann, dass sie zu diesem Testfall

[6] `http://junit.sourceforge.net`

gehören. In der Praxis erfolgen die Benennung der Testdaten und die Initiierung der Annotation `@InputData` wie folgt. Angenommen, man möchte ein PL/0-Modell und ein UML-Modell mit derselben Testmethode testen. Dann können in der Testumgebung für beide Metamodelle jeweils eine Datei, in der das zu testende Modell enthalten ist, und eine Datei, welche das nach dem Modell-Refactoring erwartete Modell enthält, abgelegt werden. Die Eingabe-Modelle können beispielsweise so benannt werden: `IN_test.pl0` und `IN_test.uml`. Analog dazu ist folgende Benennung der erwarteten Modelle denkbar: `EXP_test.pl0` und `EXP_test.uml`. Das im Test refaktorierte Modell und das erwartete Modell können dann mit *EMF Compare* auf Übereinstimmung verglichen werden. Dabei ist zu erkennen, dass sowohl die Eingabe-Modelle als auch die erwarteten Modelle nach einem Muster benannt wurden, welche der neuen Annotation wie folgt übergeben werden: `@InputData({"IN_", "EXP_"})`. Aus dieser Annotation erstellt die hier erweiterte Testumgebung eine Test-Suite, die für jede Testmethode und für jede zusammengehörende Testdaten-Menge einen Testfall erstellt. Für das zuvor genannte Beispiel würden demnach zwei Testfälle generiert: einer für das PL/0-Modell und einer für das UML-Modell. Innerhalb der Testmethode kann dann unabhängig von den konkreten Daten über die definierten Muster auf die Modelle zugegriffen werden. Auf diese Weise ist es möglich, zu testende Modelle und darauf anzuwendende Modell-Refactorings automatisiert zu testen, indem einfach nur neue Testdaten nach den definierten Mustern abgelegt werden. So ist es nicht notwendig, für neue Testdaten manuell einen neuen Testfall zu implementieren, da dieser automatisch generiert wird.

In diesem Kapitel wurde ein Einblick in die Testumgebung gegeben, wie sie in dieser Arbeit umgesetzt und eingesetzt wurde. Dass die Implementierung von *Refactory* auch praktisch einsetzbar ist, wird nun im nachfolgenden Kapitel evaluiert.

7.9 Evaluation anhand von mehreren Metamodellen

In diesem Kapitel wird nun gezeigt, dass der Ansatz des generischen Modell-Refactorings auf Basis von Rollen auch tatsächlich den Grad der Wiederverwendbarkeit erhöht und ein Großteil der generisch definierten Modell-Refactorings ohne sprachspezifische Erweiterungen auskommt. Das Ziel der Evaluation ist es demnach zu zeigen, dass die spezifizierten Modell-Refactorings für möglichst viele Metamodelle wiederverwendet werden können und möglichst wenig Post-Prozessoren zu registrieren sind. Dazu werden im Folgenden alle *Role Models* der generisch definierten Modell-Refactorings kurz erläutert und anschließend wird in einer tabellarischen Übersicht gezeigt, auf welche Metamodelle sie abgebildet werden konnten. Um eine Vorstellung davon zu erlangen, wie generisch definierte Modell-Refactorings entstehen, wird zunächst die Vorgehensweise der Spezifikation von Modell-Refactorings erläutert.

Der Startpunkt des Vorgehens ist meist die Erkenntnis, dass ein schon bekanntes Code-Refactoring aus den einschlägigen, in Kapitel 3 vorgestellten Katalogen in anderen DSLs oder Metamodellen nützlich sein kann. Hat man diese Überzeugung erlangt, wird daraus ein generisches Modell-Refactoring abgeleitet. Dies geschieht durch Abstrahierung der teilnehmenden Elemente, woraus ein *Role Model* entsteht. Anschließend werden in der

Refactoring Specification die dafür notwendigen Schritte auf Basis der definierten Rollen spezifiziert. Das Ergebnis ist ein generisch definiertes Modell-Refactoring. Dieses kann nun auf die Ziel-Metamodelle abgebildet werden. Entspricht ein *Role Model* im Kontext einer DSL noch nicht vollständig den Vorstellungen des beabsichtigten Modell-Refactorings, kann das *Role Model* wieder als Grundlage für ein weiteres Modell-Refactoring dienen. Auf der anderen Seite ist außerdem die neue Definition von Modell-Refactorings möglich, die nicht auf anderen aufbauen. Die Menge der eigenen Modell-Refactorings wächst somit mit dem Vermögen des Refactoring-Designers, entweder vorhandene Refactorings zu abstrahieren oder neue zu erkennen, durch Identifikation von Mustern in der manuellen Umstrukturierung. Auf diese Weise entstehen nach und nach generisch definierte Modell-Refactorings. Nun werden die in dieser Arbeit erstellten Modell-Refactorings erläutert.

Insgesamt wurden neun generische Modell-Refactorings definiert. *Extract X With Reference Class* wird hier nicht mehr erläutert, da die daran beteiligten Rollen schon in Abbildung 7.4 gezeigt wurden und ihre Bedeutung Kapitel 7.2 zu entnehmen ist. Außerdem wurde in Abbildung 7.5 detailliert auf die in diesem Modell-Refactoring durchzuführenden Schritte eingegangen. Alle weiteren generischen Modell-Refactorings werden nun erklärt. Dabei ist nachfolgend ihr Name aufgelistet und dahinter ist der Verweis auf das entsprechende *Role Model* zu finden.

Rename X – Abbildung 7.10(a)
Rename X besitzt nur eine Rolle. `Nameable` besitzt ein Rollen-Attribut `name` und repräsentiert dabei eine Metaklasse, deren Name geändert werden soll. Dieses Refactoring entstand in Anlehnung an *Rename Method* und wird sehr häufig durchgeführt. Außerdem ist es das Paradebeispiel für die Wahrung der horizontalen und vertikalen Konsistenz, da beim Umbenennen zahlreiche Modelle betroffen sind.

Remove Empty Contained X – Abbildung 7.10(b)
Dieses Modell-Refactoring wird benutzt, um aus einem Container-Element andere Elemente zu entfernen, die selbst keine Unter-Elemente besitzen. Das *Role Model* besitzt dazu die Rollen `RemoveeContainer` und `Removee`, zwischen denen eine *Role Composition* existiert. In der *Refactoring Specification* wird für dieses Modell-Refactoring der in Abschnitt 6.2.2.4 erläuterte Modifizierer `EMPTY` benutzt.

Remove Unused Contained X – Abbildung 7.10(c)
Rename Unused Contained X ist aus *Remove Parameter* hervorgegangen und ist dem zuvor beschriebenen Modell-Refactoring sehr ähnlich. Dies ist an den gleichen *Role Models* zu erkennen. Der Unterschied besteht darin, dass mit diesem Modell-Refactoring nur Elemente entfernt werden, die nicht von anderen Modell-Elementen referenziert werden. Dazu wird in der *Refactoring Specification* der Modifizierer `UNUSED` verwendet.

Move X Loosely – Abbildung 7.10(d)
Soll ein Modell-Element innerhalb seines Containers an eine andere Position verschoben

werden, so wird *Move X Loosely* verwendet. Die Rolle `Container` besitzt dazu jeweils eine *Role Composition* zu den Rollen `Movee` und `Neighbor`. Erstere entspricht dem Element, das verschoben werden soll, wohingegen die zweite Rolle das neue Nachbar-Element repräsentiert. Um sicherzustellen, dass nur gleichartige Elemente neue Nachbarn sein können, wurde zwischen diesen beiden Rollen eine *Role Implication* modelliert. Dieses Modell-Refactoring ist in Anlehnung an *Move Method* entstanden, verschiebt Modell-Elemente aber lediglich innerhalb eines Containers.

Move X – Abbildung 7.10(e)
Mit diesem Modell-Refactoring können Elemente von einem Container in einen anderen verschoben werden. Den beiden Containern entsprechen demnach die Rollen `Source-Container` und `TargetContainer`. Beide besitzen eine *Role Composition* zur Rolle `Movee`, die für die zu verschiebenden Elemente steht. Die Verschiebung wird hier entlang eines Pfades durchgeführt, in dem der `SourceContainer` als Unter-Container des anderen Containers zu sehen ist. Deshalb wurde zwischen beiden Rollen eine *Role Association* von der Quelle zum Ziel modelliert. Vorbild dieses Modell-Refactorings war *Pull Up Member*.

Introduce Reference Class – Abbildung 7.10(f)
Das Modell-Refactoring *Introduce Reference Class* dient der Ableitung eines Modell-Elements in Abhängigkeit von anderen Elementen. Die Basis ist hier die Rolle `Derivee`, auf die eine *Role Association* von dem abgeleiteten, neu anzulegenden Element der Rolle `Derivation` verweist. Die Ableitung muss in dem Modell-Element erzeugt werden, welches die Rolle `DerivationContainer` spielt. Dieses Modell-Refactoring basiert nicht auf existierenden. Als Grundlage diente die Erkenntnis, dass man in Feature-Modellen, wie schon in Abschnitt 3.2.2 erläutert, oft *Constraints* für Features ableitet. In der nachfolgenden Tabelle 7.1 ist zu erkennen, dass dieses Modell-Refactoring auch in anderen Metamodellen Verwendung finden kann.

Extract X – Abbildung 7.10(g)
Dieses Modell-Refactoring ist dem schon erläuterten *Extract X With Reference Class* ähnlich und entstand in Anlehnung an *Extract Superclass*. Im Gegensatz zu *Extract X With Reference Class* wird hier jedoch die Rolle `MovedReference` nicht verwendet. Darüber hinaus existiert zwischen den Rollen `OrigContainer` und `NewContainer` eine *Role Association* vom ursprünglichen zum neu zu erzeugenden Container. Mit *Extract X* können so Elemente der Rolle `Extractee` von einem Container in einen neuen, auf den der ursprüngliche Container verweist, verschoben werden. Dadurch wird auch hier nach oben entlang eines Container-Pfades verschoben.

Extract Sub X – Abbildung 7.10(h)
Extract Sub X kann als das entgegengesetzte Modell-Refactoring von *Extract X* angesehen werden. Hier werden die Elemente der Rolle `Extractee` in einen neuen Container verschoben, der jedoch als Unter-Container des ursprünglichen Containers erstellt

wird. Deshalb besitzt die Rolle `OrigContainer` eine *Role Composition* zu den Rollen `NewContainer` und `Extractee`. Auch dieses Modell-Refactoring wurde in Anlehnung an *Extract Method* entwickelt.

Damit wurden alle generisch spezifizierten Modell-Refactorings erläutert. In der Tabelle 7.1 kann nun abgelesen werden, auf welche unterschiedlichen Metamodelle diese generischen Modell-Refactorings abgebildet werden konnten. Für jedes *Role Mapping* wurde ein im Kontext des jeweiligen Metamodells aussagekräftiger Name angegeben, der analog zu den Code-Refactorings schon auf die Bedeutung schließen lassen soll. Dieser ist in der Tabelle in der dritten Spalte angegeben. In der vierten Spalte wird außerdem ein Kreuz gesetzt, wenn für das jeweilige Metamodell und Modell-Refactoring ein Post-Prozessor implementiert werden musste.

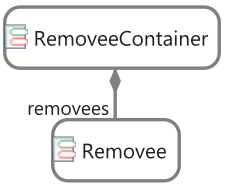
Die neun zuvor erläuterten generischen Modell-Refactorings wurden auf 16 verschiedene Metamodelle übertragen. Dabei wurden Sprachen verschiedener Komplexität verwendet. Beispielsweise besitzen die Metamodelle für UML, OWL (*Web Ontology Language*), Timed Automata (Erweiterung von Zustands-Automaten um Uhren) und Java sehr viele Metaklassen und Beziehungen zwischen ihnen. Sprachen wie Ecore, Concrete-Syntax (in EMFText verwendete Sprache zur Definition einer textuellen Syntax) oder BPMN (*Business Process Modeling Notation*) sind hingegen weniger komplex. Viele der Beispielsprachen, wie das schon erläuterte PL/0 oder Office und Forms, besitzen sehr wenige Metaklassen und sind als einfache Sprachen zu bewerten. Bei der Wahl der Sprachen wurde darauf geachtet, dass nicht nur objektorientierte Metamodelle Verwendung finden, sondern auch Sprachen dabei sind, die ein hohes Maß an Popularität (beispielsweise UML oder BPMN) genießen. Viele der in Tabelle 7.1 aufgelisteten Sprachen sind im schon erwähnten Syntax-Zoo[7] von EMFText zu finden. Metamodelle wie Ecore oder BPMN sind in Eclipse enthalten oder können sehr einfach installiert werden.

Wie in der Tabelle zu erkennen, konnten 9 generisch spezifizierte Modell-Refactorings auf 16 Metamodelle abgebildet werden. Insgesamt konnten aus 53 *Role Mapping Models* entsprechende Modell-Refactorings der unterschiedlichen Sprachen und Metamodelle erzeugt werden. Hervorzuheben ist, dass nur 6 Post-Prozessoren implementiert werden mussten, wodurch gezeigt ist, dass die Wiederverwendung von Modell-Refactorings einen großen Vorteil bringt. Außerdem konnte jedes generisch definierte Modell-Refactoring in mindestens zwei Metamodellen angewendet werden. Darüber hinaus ist in der Tabelle zu erkennen, dass einige Refactorings sogar mehrmals in ein und demselben Metamodell Verwendung finden, woran der generische Triskell-Ansatz scheiterte.

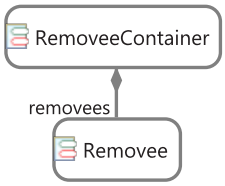
Zusammenfassend weist dieses Kapitel nach, dass das entwickelte Konzept erfolgreich umgesetzt werden konnte. Die Ergebnisse der Evaluation zeigen außerdem, dass die Wiederverwendung von generischen Modell-Refactorings in zahlreichen Metamodellen mit diesem Konzept sinnvoll ist und viele Vorteile bringt. So muss für ein konkretes Metamodell nur ein *Role Mapping Model* spezifiziert werden, wodurch ein generisch definiertes Modell-Refactoring darin dann Anwendung finden kann. Darüber hinaus ermöglicht das Framework *Refactory* die Durchführung von Modell-Refactorings, unabhängig von der

[7] `http://www.emftext.org/zoo`

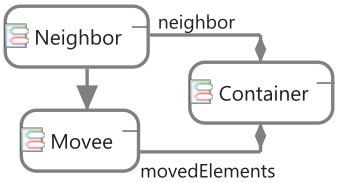
(a) *Rename X*



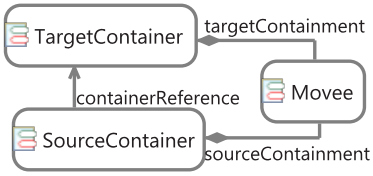
(b) *Remove Empty Contained X*



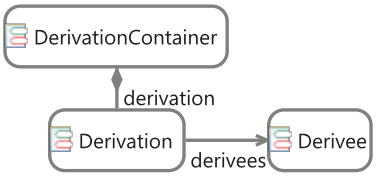
(c) *Remove Unused Contained X*



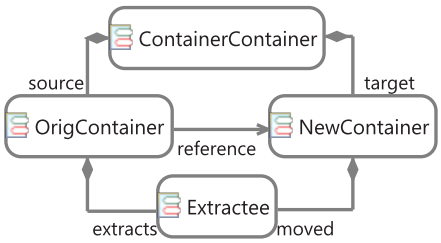
(d) *Move X Loosely*



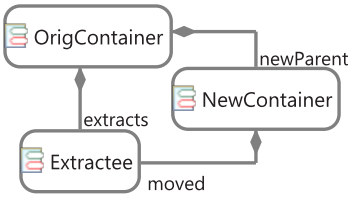
(e) *Move X*



(f) *Introduce Reference Class*



(g) *Extract X*



(h) *Extract Sub X*

Abbildung 7.10: Alle *Role Models* der umgesetzten generischen Modell-Refactorings

Tabelle 7.1: Auf Metamodelle abgebildete generische Modell-Refactorings

Generisher Name	Metamodel	Spezifischer Name	PP
Rename X	Java	Rename Element	
	Conference	Rename Conference Element	
	Forms	Rename Option	
	Office	Rename Employee	
	Office	Rename Office	
	OWL	Rename Element	
	PL/0	Rename Procedure	
	PL/0	Rename Declaration	
	PL/0	Rename Program	
	Roles	Rename Role Model Element	
	Sandwich	Rename Ingredient	
	Textadventure	Rename Room	
	Textadventure	Rename Door	
	TimedAutomata	Rename EElement	
	Ecore	Rename EElement	
	BPMN	Rename Identifiable	
	UML	Rename Element	
	ConcreteSyntax	Rename Token	
	Feature	Rename Feature	
Introduce Reference Class	Ecore	Derive Composite Interface	
	UML	Create Sub Interface	
	UML	Create Subclass	
	Feature	Introduce Constraint	
Extract X With Reference Class	Java	Extract Method	X
	Conference	Extract Track	
	Forms	Introduce Item Group	
	PL/0	Extract Procedure	
	SimpleGUI	Encapsulate In Panel	
	Textadventure	Move To New Room	
	BPMN	Extract Sub Process	X
	UML	Extract Composite State	X
	ConcreteSyntax	Extract Rule	X
Extract X	Ecore	Extract EEnum	
	Ecore	Introduce Parameter Object	
	Ecore	Extract Super Class	
	Ecore	Extract Interface From Features	X
	Ecore	Extract Interface From Operations	X
	UML	Extract Super Class	
	UML	Extract Interface	
Remove Empty Contained X	Java	Remove Empty Methods	
	UML	Remove Empty Superclass	
Move X Loosely	Office	Move Employee Next To	
	Forms	Move Group Next To	
Move X	PL/0	Pull Up Constant	
	Ecore	Pull Up Feature	
	UML	Pull Up Property	
	UML	Pull Up Operation	
	UML	Pull Up Operation To Interface	
	UML	Pull Up Operation To Super Interface	
Extract Sub X	Ecore	Extract Sub EPackage	
	UML	Extract Sub Package	
Remove Unused Contained X	Java	Remove Unused Parameters	
	UML	Remove Unused Classes	

Art der Repräsentation der Modelle. Viele Erweiterungspunkte in diesem Framework gestatten außerdem die Anpassung an weitere spezifische Anforderungen des Nutzers, wie beispielsweise die Anbindung weiterer *Index Connectors* oder *Editor Connectors*. Zudem existiert ein Wiki[8], in dem in Zukunft entwickelte Modell-Refactorings dokumentiert und gepflegt werden können. So schließt dieses Kapitel das Konzept und die Implementierung ab. Im nun folgenden letzten Kapitel werden die in dieser Arbeit erreichten Ziele noch einmal zusammengefasst und es wird ein Ausblick gegeben.

[8] `http://emftext.org/index.php/Refactoring:Catalog`

8 Zusammenfassung und Ausblick

In diesem letzten Kapitel der vorliegenden Arbeit werden nun die erreichten Ergebnisse und gewonnenen Erkenntnisse noch einmal zusammengefasst und reflektiert. Zudem erfolgt eine abschließende Bewertung. Ein Ausblick und Hinweise für zukünftige Arbeiten runden diese Arbeit ab.

8.1 Ergebnisse

Ziel dieser Arbeit war es, ein Konzept zu entwickeln, mit dem es möglich ist, Modell-Refactorings auf generische Art und Weise zu spezifizieren und damit in Modellen beliebiger DSLs und Metamodelle wiederzuverwenden. Der Ursprung des Begriffes *Modell-Refactoring* liegt hierbei in zwei Teilbereichen der Software-Entwicklung. Zum einen bildet die Technik des Code-Refactorings die Grundlage für diese Arbeit. Die dort umgesetzten Konzepte und entwickelten Refactoring-Kataloge dienten als Basis für die Untersuchung der Übertragbarkeit auf den zweiten erwähnten Teilbereich: die Modellgetriebene Software-Entwicklung. Somit war ein zu erreichendes Teilziel dieser Arbeit zu analysieren, wie das hinreichend erforschte Code-Refactoring auf Modelle und ihre Metamodelle angewendet werden kann. Dazu wurde in Kapitel 2 zunächst geklärt, was unter Refactoring allgemein zu verstehen ist, welche Vorteile die MDSD bringt und wo ihre formalen Wurzeln liegen. Anschließend wurde eine Brücke zwischen Code und Modellen geschlagen, wodurch das Modell als primäres Artefakt in der Software-Entwicklung angesehen werden kann. Danach erfolgte die begriffliche Definition des Modell-Refactorings und eine Vorstellung der Validierungs-Umgebung EMFText. Dort wurde außerdem darauf eingegangen, dass Modelle verschiedene Repräsentationen haben können, von denen die Durchführung von Modell-Refactorings unabhängig sein muss.

In Kapitel 3 erfolgte die Analyse vorhandener Refactoring-Kataloge. Als Grundlage dieser Kataloge dienten vor allem die Arbeiten von Opdyke [Opd92] und Fowler et al. [FBB+99]. Ziel dieser Analyse war es herauszufinden, ob und unter welchen Bedingungen Code-Refactorings auf Modelle übertragen werden können. Dazu wurde zunächst ein repräsentativer Querschnitt der existierenden Refactorings gegeben, der sowohl die am häufigsten benutzten als auch Refactorings unterschiedlicher Schwierigkeitsgrade beinhaltet. Aus dieser Analyse konnten dann primitive Muster abgeleitet werden, die in den einzelnen Refactorings immer wieder Verwendung finden. So wurden die Muster Erzeugen, Verschieben, Ändern und Entfernen von Elementen bestimmt. Diese Primitiven sind später direkt in die Konzeption mit eingeflossen. Abschließend wurden in Kapitel 3 vier Kategorien gebildet, in die dann vorhandene Refactorings eingeordnet wurden. Diese Kategorien sind aufsteigend nach ihrer Übertragbarkeit auf Modelle geordnet und heißen: spezifisch, objektorientiert, generell-programmatisch und generisch. Das bedeutet,

ein Code-Refactoring ist generisch, wenn es abstrakt und strukturell übertragbar ist. Es kann dann über Metamodell-Grenzen hinweg wiederverwendet werden. Die Umsetzbarkeit aller dieser Kategorien galt als eine wichtige Anforderung an diese Arbeit.

Die Kriterien der Untersuchung existierender Ansätze des Modell-Refactorings bilden die in Kapitel 4 aufgestellten funktionalen und nichtfunktionalen Anforderungen. Diese wurden auf Grundlage vorhandener Arbeiten definiert und an die hier zu erfüllenden Ziele angepasst. Sie dienten dann in Kapitel 5 der Bewertung vorhandener Ansätze, in denen die noch junge Thematik des Modell-Refactorings bearbeitet wurde. Dort bildeten sich drei Arten der Spezifikation von Modell-Refactorings heraus. Diese unterscheiden sich in der MOF-Ebene, auf der sie definiert werden. Im Resultat erwies sich der M3-Ansatz der französischen Triskell-Gruppe als sehr generisch, da mit Hilfe eines Meta-Metamodells (GenericMT) Modell-Refactorings unabhängig von den Ziel-Metamodellen spezifiziert werden können. Da das GenericMT aber nur objektorientierte Strukturen enthält, ist die Umsetzung aller entworfenen Kategorien nur schwer bis gar nicht möglich. Außerdem ist das GenericMT monolithisch, weshalb die starken Unterschiede, die in verschiedenen Metamodellen auftreten können, kaum darin reflektiert werden können. Darüber hinaus erwies sich die Technik der Adaption der Metamodelle auf das GenericMT als zu statisch, da unterschiedliche Strukturen nicht in ein und demselben Metamodell für Modell-Refactorings zur Verfügung gestellt werden können. Die Ansätze auf der MOF-Ebene M2 hingegen überwinden die letztgenannte Schwäche, indem Modell-Refactorings auf beliebigen Metamodell-Strukturen definierbar sind. Im M1-Ansatz werden Modell-Refactorings zwar anhand von konkreten Modellen spezifiziert, diese werden jedoch dann auf die Ebene des Metamodells propagiert, wodurch auch hier die zu refaktorierenden Strukturen sehr gut kontrolliert werden können. Der entscheidende Nachteil der Spezifikation auf M2- beziehungsweise auf M1-Ebene ist, dass die Modell-Refactorings nicht generisch und deshalb nicht in anderen Metamodellen wiederverwendbar sind. Die im Kapitel 5 gewonnenen Erkenntnisse flossen dann in die Konzeption mit ein.

In Kapitel 6 lag die Herausforderung der Konzeption des generischen Modell-Refactorings darin, die zwei wichtigsten Vorteile existierender Ansätze zu verschmelzen. Zum einen war dies die generische Spezifikation von Modell-Refactorings unabhängig von den Ziel-Metamodellen (im Gegensatz zu *EMF Refactor* und zum *Operation Recorder*). Zum anderen mussten die zu refaktorierenden Strukturen exakt kontrollierbar sein (im Gegensatz zum GenericMT). Zur Überwindung dieser beiden Hürden wurde in diesem Kapitel auf das Prinzip der Rollen zurückgegriffen. Diese wurden erstmals von Reenskaug erwähnt. Später wurde dann von Riehle und Gross ein informelles Metamodell für Rollen definiert. Rollen eignen sich dazu, Muster von Kollaborationen zwischen Objekten in verschiedenen Kontexten zu beschreiben. Daran scheitert beispielsweise das GenericMT, weil mit ihm lediglich ein statischer Kontext definiert wird. In diesem Kapitel wird weiter argumentiert, dass jedes Modell-Refactoring als abgegrenzter Kontext zu sehen ist, in dem jede Metaklasse eines Ziel-Metamodells stets eine andere Bedeutung haben kann. Aus diesen Überlegungen heraus ist somit das Konzept des *Role Models* zur generischen Definition aller an einem Modell-Refactoring teilnehmenden Modell-Elemente entwickelt worden. Dazu wurde das Metamodell für *Role Models* vorgestellt und erläutert. Darauf aufbauend, ist außerdem ein Metamodell für eine Transformationssprache entstanden,

mit der es möglich ist, die einzelnen durchzuführenden Schritte eines Modell-Refactorings auf Basis eines definierten *Role Models* zu spezifizieren. Mit der Rollen-basierten Definition von Modell-Refactorings konnte das Problem der dynamischen Generizität gelöst werden. Auf diese Art und Weise kann für jedes Modell-Refactoring mit Hilfe eines *Role Models* ein eigener Kontext definiert und auf dessen Grundlage die durchzuführenden Teilschritte spezifiziert werden. Um die zu refaktorierenden Strukturen kontrollieren zu können, wurde in diesem Kapitel außerdem das Metamodell *Role Mapping Model* vorgestellt, mit dem *Role Models* auf die jeweiligen Ziel-Metamodelle abgebildet werden können. Damit werden den Metaklassen, deren Instanzen an einem Modell-Refactoring teilnehmen können, die definierten Rollen zugewiesen. Darüber hinaus können die modellierten Kollaborationen zwischen den Rollen auf Relationen zwischen Metaklassen abgebildet werden. Eine Besonderheit liegt in der Abbildbarkeit von Kollaborationen auf Pfade. Hier wird von Rollen-Kollaborationen ein Rahmen vorgegeben, der aber in jedem Metamodell individuell umsetzbar ist. Mit dieser Technik kann ein *Role Model* auf beliebige Strukturen ein und desselben Metamodells abgebildet werden, was mit dem generischen Triskell-Ansatz auch nicht möglich ist. Um nach der Durchführung eines Modell-Refactorings zusätzlich sprachspezifische Änderungen des refaktorierten Modells, die im Kontext des jeweiligen Metamodells und des Modell-Refactorings wichtig sind, durchführen zu können, wurde außerdem die Technik der Post-Prozessoren eingeführt. Diese können für ein Metamodell, in Kombination mit einem konkreten *Role Mapping*, registriert und nach einem Modell-Refactoring ausgeführt werden. Des Weiteren wurden in diesem Kapitel die Angabe von Vor- und Nachbedingungen, die Bewahrung der Semantik sowie der horizontalen und vertikalen Konsistenz diskutiert. Die ersten beiden liegen außerhalb der näheren Betrachtung dieser Arbeit und für die Konsistenz wurde im Kapitel der Implementierung ein Index-Mechanismus erläutert.

In Kapitel 7 finden sich eine Beschreibung der Umsetzung des entwickelten Konzepts, sowie deren Evaluation. Dabei ist das Framework *Refactory* für die Plattform Eclipse entstanden, mit dem Refactoring-Designer Modell-Refactorings generisch spezifizieren können und DSL-Designer diese dann auf ihr konkretes Metamodell abbilden müssen. Für den Anwender fügt sich *Refactory* in den gewohnten Prozess der Umstrukturierung mittels bekannter Code-Refactorings ein. Dadurch ist sichergestellt, dass sich Nutzer nicht auf ein neues Vorgehen einstellen müssen. Um generisch definierte Modell-Refactorings durchführen zu können, wurde in diesem Kapitel der Interpreter erläutert, mit dem die Modell-Refactorings anhand der *Role Models* ausgeführt und mit Hilfe des *Role Mappings* in das entsprechende Ziel-Metamodell übersetzt werden. Des Weiteren wurden zahlreiche Erweiterungspunkte für *Refactory* definiert und erläutert, an denen beispielsweise neue Modell-Refactorings oder Implementierungen für eigene Editoren, zur Ermittlung der selektierten Modell-Elemente aus dem darin enthaltenen Modell, registriert werden können. Den Abschluss bildet die Evaluation von 9 generisch spezifizierten Modell-Refactorings und deren Wiederverwendung in 16 verschiedenen Metamodellen. Daraus sind insgesamt 53 Modell-Refactorings entstanden, womit gezeigt werden konnte, dass die Wiederverwendung von Modell-Refactorings über Metamodell-Grenzen hinweg sinnvoll und vor allem auch durchführbar ist.

Somit konnten in den Kapiteln der Konzeption und der Implementierung alle in Kapi-

tel 4 aufgestellten funktionalen und nichtfunktionalen Anforderungen umgesetzt werden. Ausnahmen bilden dabei die Angabe von Vor- und Nachbedingungen, die Bewahrung der Semantik sowie die Empfehlung von Modell-Refactorings. Bei diesen Anforderungen handelt es sich wiederum um komplexe Aufgabenstellungen, die außerhalb des Fokus dieser Arbeit lagen. Diesbezüglich wird unter anderem im nachfolgenden letzten Abschnitt noch ein Ausblick gegeben.

8.2 Ausblick

Zum Abschluss der vorliegenden Arbeit sollen hier weitere Hinweise gegeben werden, die als Ausgangspunkt für nachfolgende Arbeiten dienen können. Diese Ratschläge sind aus identifizierten Problemen, vor allem aus den erläuterten, nicht umgesetzten Anforderungen, hervorgegangen und basieren darüber hinaus auf Chancen, die sich aus einigen hier umgesetzten Konzepten und den verwendeten Technologien ergeben.

Vor- und Nachbedingungen In Kapitel 6.3 wurde die Notwendigkeit der Angabe von Vor- und Nachbedingungen für ein Modell-Refactoring erläutert. Diese Forderung steht im Einklang mit den von Roberts aufgestellten Anforderungen in [Rob99]. Mit Vor- und Nachbedingungen kann sichergestellt werden, dass ein Modell-Refactoring im aktuellen Zustand eines Modells überhaupt durchführbar ist. Können diese Bedingungen nicht erfüllt werden, so besteht immer die Gefahr, dass sich das Modell nach dem Refactoring in einem inkonsistenten Zustand befindet. Hierbei wurde zwischen sprachspezifischen und für ein Modell-Refactoring allgemeingültigen Bedingungen unterschieden. Bei beiden Arten sollte die Wahl des Standards zur Spezifikation von Bedingungen auf OCL fallen, wie auch schon in Kapitel 6.3 argumentiert. Sprachspezifische Bedingungen könnten dann mit OCL-Ausdrücken angegeben werden. So formulierte Invarianten könnten außerdem den kleinen Teilbereich bezüglich der Bewahrung der Semantik abdecken, in den die WFRs fallen. Demzufolge müsste in Zukunft untersucht werden, wie OCL in die Erstellung eines *Role Mapping Models* eingebunden werden kann, da dort die definierten Rollen eines Modell-Refactorings auf konkrete Metaklassen abgebildet werden. Allgemeingültige Bedingungen eines Modell-Refactorings müssten im Gegensatz dazu einzig auf einem definierten *Role Model* spezifiziert werden. Dies hätte zur Folge, dass zum Zeitpunkt der Interpretation der Bedingungen das *Role Mapping Model* herangezogen werden muss, um die OCL-Ausrücke im Kontext des konkreten Metamodells zu prüfen. Somit ist in künftigen Arbeiten zu untersuchen, wie ein Rollen-basierter OCL-Interpreter, der die formulierten Bedingungen zur Laufzeit in einem konkreten Metamodell auswertet, umzusetzen ist.

Bewahrung des Verhaltens Der Teil der Semantik, der mittels WFRs zu beschreiben ist, kann mit den zuvor beschriebenen Techniken der Definition von Vor- und Nachbedingungen mittels OCL geprüft werden. Werden diese von einem Modell-Refactoring nicht erfüllt, so kann der Anwender entscheiden, ob es durchgeführt werden soll. Wie jedoch in Kapitel 6.4 erläutert, reicht dies nicht immer aus, da jedes Metamodell und

darauf angewendete Modell-Refactorings stets eine andere Bedeutung haben. Aufgrund der fehlenden Standardisierung auf dem Gebiet des automatischen Beweisens der Beibehaltung der Semantik sollte in zukünftigen Arbeiten untersucht werden, wie die Bedeutung von Modellen formal spezifiziert werden kann, ohne den DSL-Designer dabei zu überfordern. Wurden die Grundlagen dafür gelegt, kann dann analysiert werden, ob es möglich ist, schon nach der Definition eines generischen Modell-Refactorings zu entscheiden, ob die Semantik erhalten bleibt oder nicht. Es ist jedoch davon auszugehen, dass die Beibehaltung der Semantik frühestens im Kontext eines konkreten Metamodells überprüft werden kann, also nach Erstellung eines *Role Mapping Models*. Darüber hinaus ist auch die Überprüfung zur Laufzeit denkbar, da erst dann der vollständige Kontext eines konkreten Modells zur Verfügung steht. Daraus ergeben sich für die Überprüfung des Modell-Verhaltens abermals die drei Kategorien, auf welcher MOF-Ebene die Beibehaltung am effektivsten überprüft werden kann.

Model Smells und Empfehlung von Modell-Refactorings In Kapitel 3.1 wurde der Begriff des *Bad Smells* genannt, mit dem in vorhandenem Code vorkommende „schlechte Gerüche“ bezeichnet werden, die schlechtes Design indizieren können. Als Beispiel ist hier *Duplicated Code* zu nennen. Solche Indikatoren werden auf dem Gebiet des Modell-Refactorings *Model Smells* genannt [MTM07]. Wenn es möglich ist, *Model Smells* zu definieren und diese in Verbindung mit Modell-Refactorings zu setzen, dann kann damit auch die Empfehlung von Modell-Refactorings vorangetrieben werden [MVG06, MTR07]. Auf diese Weise könnten Modelle automatisiert nach *Model Smells* durchsucht und dementsprechend Modell-Refactorings gezielt empfohlen werden, um das Design kontrolliert zu verbessern. Hier sind auch Überlegungen des automatischen Modell-Refactorings denkbar, wobei diese ohne den Dialog mit einem Anwender durchzuführen sind. In nachfolgenden Arbeiten sollte demnach untersucht werden, wie *Model Smells* definiert und in konkreten Modellen erkannt werden können.

Design Patterns für Modelle Neben den zuvor erläuterten Ratschlägen für zukünftige Arbeiten, aufgrund der nicht umgesetzten Anforderungen, kann außerdem darauf aufgebaut werden, dass man mit *Role Models* auch die in [GHJV04] definierten *Design Patterns* beschreiben kann. Auf diese Weise könnte Kerievskys Arbeit in [Ker04] praktisch umgesetzt werden. Er beschreibt dort Refactorings, die in vorhandenen Code *Design Patterns* einführen. Diese gelten als gutes Design, Kerievsky warnt jedoch auch davor, Zusammenhänge damit komplizierter zu machen. Auf dieser Grundlage könnten *Design Patterns* zum einen in Form von *Role Models* umgesetzt werden, die dann als Ziel eines Refactorings zu verstehen sind. Zum anderen ergibt sich aus dieser Überlegung, dass *Design Patterns* nicht nur in Code, sondern vielmehr auch in Modellen relevant sein können. Aus diesem Grund könnte in zukünftigen Arbeiten untersucht werden, welche *Design Patterns* man in Modellen identifizieren kann, und diese dann mittels *Role Models* umsetzen. Diese Analyse wäre ein Schritt in die Richtung des automatischen Aufspürens von Mustern in Modellen.

Komposition von Refactorings Ein weiterer Punkt, der es wert ist untersucht zu werden, ist die Komposition von Modell-Refactorings. Auf diese Weise könnten aus schon definierten generischen Modell-Refactorings neue komposite Modell-Refactorings erstellt werden. Dadurch käme ein weiterer Aspekt der Wiederverwendung von generischen Modell-Refactorings hinzu, da derartige komplexe Modell-Refactorings nicht von Grund auf neu zu spezifizieren wären. Dem dabei auftretenden Problem gleicher Rollen-Bezeichner in den zu komponierenden *Role Models* kann, wie schon in Kapitel 6.1 vorgeschlagen, begegnet werden. Um Rollen eindeutig zu identifizieren, müssen diese nach dem Muster `Rollen-Modell.Rolle` angesprochen werden. Zu untersuchen bliebe trotzdem, wie die *Refactoring Specification* des einen Modell-Refactorings mit der *Refactoring Specification* eines anderen intensiver kombiniert werden kann und wie der Informationsfluss dazwischen zu spezifizieren ist. Außerdem stellt sich beim Thema der Komposition immer die Frage, bis zu welcher Tiefe dies gestattet werden soll und ob beispielsweise Rekursivität sinnvoll und gewünscht ist. Auf diese Weise könnten aus kompositen Modell-Refactorings noch komplexere komposite Modell-Refactorings zusammengesetzt werden.

***-Attribute** Darüber hinaus birgt das Metamodell *Role Model* weiteres Potenzial für zukünftige Untersuchungen. In einem *Role Model* können, wie in Abschnitt 6.2.1 beschrieben, Rollen-Attribute definiert werden. Diese können dafür benutzt werden, um bestimmte Attribute von Modell-Elementen während eines Modell-Refactorings zu verändern. Ein Rollen-Attribut kann dabei mit dem Modifizierer `optional` bestückt werden, womit angegeben wird, dass dieses nicht zwingend auf ein Metaklassen-Attribut abgebildet werden muss. Hier wäre denkbar, die Möglichkeit der optionalen Modellierung von Rollen-Attributen noch flexibler zu gestalten. Beispielsweise könnte untersucht werden, ob **-Attribute* vonnöten sind, durch die man ein Rollen-Attribut mit einem `*` markieren könnte, um auszudrücken, dass beliebig viele Metaklassen-Attribute diesem **-Attribut* entsprechen. Auf diese Art und Weise müsste die Anzahl der Rollen-Attribute nicht mehr statisch in einem *Role Model* angegeben, sondern kann dynamisch in dem jeweiligen *Role Mapping Model* umgesetzt werden.

Abbilden von Kollaborationen auf Operationen und virtuelle Elemente Des Weiteren könnte man sich in nachfolgenden Arbeiten damit befassen, ob Kollaborationen zwischen Rollen nicht auch auf Operationen von Metaklassen eines Metamodells abgebildet werden sollten, die zum einen gültige Modell-Elemente zurückgeben und demnach wie virtuelle Relationen zwischen Metaklassen wirken. Zum anderen wäre denkbar, Kollaborationen auch auf solche Operationen oder Relationen abzubilden, die im Ziel-Metamodell nicht explizit modelliert wurden. Auf diese Weise wäre es beispielsweise möglich, eine temporäre Operation für eine Metaklasse in Ecore zu spezifizieren, die alle ihre Unterklassen ermittelt. Eine derartige Operation kann dann wieder als Ziel einer Kollaborations-Abbildung dienen. Ein möglicher Schritt in diese Richtung ist das jüngst angekündigte Projekt *EMF Facet*[1]. Damit wird ein Framework bereitgestellt, mit

[1] `http://www.eclipse.org/proposals/emf-facet/`

dem man existierende Metamodelle dynamisch um neue Operationen oder Relationen erweitern kann.

Model-Driven Testing Zum Abschluss dieser Arbeit sei an dieser Stelle das modellgetriebene Testen (MDT) genannt. Beim konventionellen Testen entsteht oftmals das Problem, dass vor allem die Tests der GUI einer Anwendung nach und nach „veralten“ und sich in einem inkonsistenten Zustand befinden. Die Ursache ist die rasche Entwicklung der Benutzerschnittstelle, weshalb die GUI-Tests mit großem Aufwand manuell angepasst werden müssen beziehungsweise gar nicht mehr spezifiziert werden. Schafft man es doch, sowohl GUI als auch die damit verbundenen Tests mittels Modellen zu beschreiben, so kann die Konsistenz der Tests mit Modell-Refactorings bewahrt werden. Dazu müssten für die Änderungen an der GUI Modell-Refactorings definiert werden, bei deren Anwendung die korrespondierenden Umstrukturierungen dann auch in die GUI-Tests propagiert werden könnten.

Mit dem in dieser Arbeit entwickelten Konzept und der durchgeführten Umsetzung haben sich weitere interessante Herausforderungen herauskristallisiert. Diese bilden die Marschrichtung für die zukünftige Forschung. Die vorliegende Arbeit ist damit abgeschlossen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

API	Application Programming Interface
DSL	Domain Specific Language
EBNF	Extended Backus–Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
IDE	Integrated Development Environment
LTK	Eclipse Language Toolkit
M2C	Model to Code
M2M	Model to Model
MDA	Model Driven Architecture
MDSD	Model Driven Software Development
MDT	Model Driven Testing
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
SoC	Separation of Concerns
SPL	Software Product Line
UML	Unified Modeling Language
WFR	Well-Formedness Rule
XMl	Extensible Markup Language

Literaturverzeichnis

[Bé05] Bézivin, Jean: *On the Unification Power of Models.* Software and Systems Modeling, 4(2):171–188, 2005.

[BEK+06a] Biermann, Enrico, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer und Eduard Weiss: *EMF Model Refactoring based on Graph Transformation Concepts.* ECEASST, 3, 2006.

[BEK+06b] Biermann, Enrico, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer und Eduard Weiss: *Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework*, Band 4199/2006 der Reihe *Lecture Notes in Computer Science*, Seiten 425–439. Springer, 2006.

[BK10] Bürger, Christoff und Sven Karol: *Towards Attribute Grammars for Metamodel Semantics.* Technischer Bericht TUD-FI10-03 - März 2010, Technische Universität Dresden, März 2010.

[BLS+09] Brosch, Petra, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger und Wieland Schwinger: *An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example.* In: *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, Seiten 271–285, Berlin, Heidelberg, 2009. Springer.

[BSW+09] Brosch, Petra, Martina Seidl, Konrad Wieland, Manuel Wimmer und Philip Langer: *The Operation Recorder: Specifying Model Refactorings By-Example.* In: Arora, Shail und Gary T. Leavens (Herausgeber): *OOPSLA Companion*, Seiten 791–792. ACM, 2009.

[@EMF09] EMFText: *EMFText Overview.* `http://www.emftext.org/index.php/EMFText_Overview`, Juni 2009. Besucht am 22. Mai 2010.

[@EMFR] Refactor, EMF: *EMF Refactor.* `https://www.mathematik.uni-marburg.de/~swt/modref/`. Besucht am 13. Juni 2010.

[Eys08] Eysholdt, Moritz: *Towards EMF Ecore based Meta Model Evolution and Model Co-Evolution.* In: *Proceedings of the Second Workshop on MDSD Today*, Seiten 51–59. Shaker Verlag, Oktober 2008.

[FBB+99] Fowler, Martin, Kent Beck, John Brant, William Opdyke und Don Roberts: *Refactoring – Improving the Design of Existing Code.* Addison Wesley, Boston, MA, USA, 1999.

[Fla02] Flatscher, Rony G.: *Metamodeling in EIA/CDIF—Meta-Metamodel and Metamodels.* ACM Transactions on Modeling and Computer Simulation (TOMACS), 12(4):322–342, 2002.

[@Fow05] Fowler, Martin: *Language Workbenches: The Killer-App for Domain Specific Languages?* `http://www.martinfowler.com/articles/languageWorkbench.html`, Juni 2005. Besucht am 17. Mai 2010.

[@Fow09] Fowler, Martin: *Domain Specific Language.* `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html`, Oktober 2009. Besucht am 17. Mai 2010.

[GHJV04] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison Wesley, 2004.

[GS03] Greenfield, Jack und Keith Short: *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools.* In: *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 16–27, New York, NY, USA, 2003. ACM.

[HA08] Hosseini, Soodeh und Mohammad Abdollahi Azgomi: *UML Model Refactoring with Emphasis on Behavior Preservation.* In: *TASE 08: Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, Seiten 125–128, Washington, DC, USA, 2008. IEEE Computer Society.

[HJK+09] Heidenreich, Florian, Jendrik Johannes, Sven Karol, Mirko Seifert und Christian Wende: *Derivation and Refinement of Textual Syntax for Models.* In: *Fifth European Conference on Model-Driven Architecture Foundations and Applications, ECMDA-FA 2009, 23 - 26 June 2009, Enschede, The Netherlands*, 2009.

[HJSW10] Heidenreich, Florian, Jendrik Johannes, Mirko Seifert und Christian Wende: *Closing the Gap between Modelling and Java.* In: Brand, Mark van den und Jeff Gray (Herausgeber): *Proceedings of the 2nd International Conference on Software Language Engineering (SLE 2009), Revised Selected Papers*, Band 5969 der Reihe *Lecture Notes in Computer Science*, Seiten 374–383. Springer, März 2010.

[Hub08] Huber, Philip: *The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches.* Master Thesis, Technische Universität Wien, Institut für Softwaretechnik und Interaktive Systeme, 2008.

[Ker04] Kerievsky, Joshua: *Refactoring to Patterns*. Addison Wesley, 2004.

[@Ker10] Kermeta: *Kermeta - Breathe life into your metamodels*. `http://www.kermeta.org/`, März 2010. Besucht am 12. Juni 2010.

[Kle08] Kleppe, Anneke: *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison Wesley, 2008.

[Köh06] Köhler, Christian: *A Visual Model Transformation Environment for the Eclipse Modeling Framework*. Diplomarbeit, TU Berlin, Oktober 2006.

[Leh96] Lehman, Meir M.: *Laws of Software Evolution Revisited*. In: *Lecture Notes in Computer Science*, Seiten 108–124. Springer, 1996.

[Mar05] Marticorena, Raul: *Analysis and Definition of a Language Independent Refactoring Catalog*. In: *17th Conference on Advanced Information Systems Engineering (CAiSE 05). Doctoral Consortium, Porto, Portugal*, Juni 2005.

[MDDB+03] Mens, Tom, Serge Demeyer, Bart Du Bois, Hans Stenten und Pieter Van Gorp: *Refactoring: Current Research and Future Trends*. Electronic Notes in Theoretical Computer Science, 82(3):483 – 499, 2003. LDTA'2003 - Language descriptions, Tools and Applications.

[MFJ05] Muller, Pierre-Alain, Franck Fleurey und Jean-Marc Jézéquel: *Weaving Executability into Object-Oriented Meta-Languages*. In: L. Briand, S. Kent (Herausgeber): *Proceedings of MODELS/UML'2005*, Band 3713 der Reihe *LNCS*, Seiten 264–278, Montego Bay, Jamaica, Oktober 2005. Springer.

[MMBJ09] Moha, Naouel, Vincent Mahé, Olivier Barais und Jean-Marc Jézéquel: *Generic Model Refactorings*. In: *Model Driven Engineering Languages and Systems*, Band 5795/2009 der Reihe *Lecture Notes in Computer Science*, Seiten 628–643. Springer, 2009.

[MRG09] Mohamed, Maddeh, Mohamed Romdhani und Khaled Ghedira: *Classification of model refactoring approaches*. Journal of Object Technology (JOT), 8(6):143–158, September–Oktober 2009.

[MTM07] Mens, Tom, Gabriele Taentzer und Dirk Müller: *Challenges in Model Refactoring*. In: *Proc. 1st Workshop on Refactoring Tools*. University of Berlin, 2007.

[MTM08] Mens, Tom, Gabriele Taentzer und Dirk Müller: *Model-Driven Software Refactoring*. In: Rech, J. und C. Bunse (Herausgeber): *Model-Driven Software Development: Integrating Quality Assurance*, Seiten 170–203. IGI Global, Hershey, 2008.

[MTR07] Mens, Tom, Gabriele Taentzer und Olga Runge: *Analysing refactoring dependencies using graph transformation.* Software and Systems Modeling, 6(3):269–285, September 2007.

[MVG06] Mens, Tom und Pieter Van Gorp: *A Taxonomy of Model Transformation.* In: *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, Band 152, Seiten 125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[OJ90] Opdyke, William F. und Ralph E. Johnson: *Refactoring: An aid in designing application frameworks and evolving object-oriented systems.* In: *Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

[OMG02] Object Management Group: *Meta Object Facility (MOF) Specification*, April 2002.

[OMG03] Object Management Group: *MDA Guide Version 1.0.1*, Juni 2003.

[OMG06a] Object Management Group: *Meta Object Facility (MOF) Core Specification*, Januar 2006.

[OMG06b] Object Management Group: *Object Constraint Language, OMG Available Specification Version 2.0*, Mai 2006.

[OMG07] Object Management Group: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, Juli 2007.

[Opd92] Opdyke, William: *Refactoring Object-Oriented Frameworks.* Doktorarbeit, University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.

[Par72] Parnas, David Lorge: *On the Criteria To Be Used in Decomposing Systems into Modules.* Communications of the ACM, 15(12):1053–1058, 1972.

[Pfl98] Pfleeger, Shari Lawrence: *Software Engineering: Theory and Practice.* Prentice Hall, 1998.

[Rei09] Reimann, Jan: *Generisches Kompositionsmodell für UI-Mashups.* Großer Beleg, Technische Universität Dresden, Oktober 2009.

[RG98] Riehle, Dirk und Thomas Gross: *Role Model Based Framework Design and Integration.* In: *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 117–133, New York, NY, USA, 1998. ACM.

[RL04] Roock, Stefan und Martin Lippert: *Refactorings in großen Softwareprojekten.* dpunkt.verlag, Heidelberg, Deutschland, 2004.

[RLK+08] Rangel, Guilherme, Leen Lambers, Barbara König, Hartmut Ehrig und Paolo Baldan: *Behavior Preservation in Model Refactoring Using DPO Transformations with Borrowed Contexts*. In: *ICGT '08: Proceedings of the 4th international conference on Graph Transformations*, Seiten 242–256, Berlin, Heidelberg, 2008. Springer.

[Rob99] Roberts, Donald Bradley: *Practical Analysis for Refactoring*. Doktorarbeit, University of Illinois at Urbana-Champaign, Department of Computer Science, 1999.

[Rot89] Rothenberg, Jeff: *The Nature of Modeling*. Seiten 75–92, 1989.

[RR06] Robertson, Suzanne und James Robertson: *Mastering the Requirements Process*. Addison Wesley, 2. Auflage, 2006.

[RWL96] Reenskaug, Trygve, Per Wold und Odd Arild Lehne: *Working with objects – The OOram Software Engineering Method*. `http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects`, 1996.

[RZ95] Riehle, Dirk und Heinz Züllighoven: *A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor*. In: *Pattern Languages of Program Design*, Seiten 9–42. Addison Wesley, 1995.

[SBPM08] Steinberg, Dave, Frank Budinsky, Marcelo Paternostro und Ed Merks: *EMF Eclipse Modeling Framework*. Eclipse Series. Addison Wesley Professional, 2 Auflage, Dezember 2008.

[SK09] Streekmann, Niels und Steffen Kruse: *MDSD Umfrage 2009 – Ergebnisse und Trends – IF-ModE*, Dezember 2009.

[SPTJ01] Sunyé, Gerson, Damien Pollet, Yves Le Traon und Jean-Marc Jézéquel: *Refactoring UML Models*. In: *UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, Seiten 134–148, London, UK, 2001. Springer.

[SV05] Stahl, Thomas und Markus Völter: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, Heidelberg, 1. Auflage, 2005.

[Tah08] Taha, Walid: *Domain-Specific languages*. In: *International Conference on Computer Engineering Systems, 2008. ICCES 2008*, Seiten xxiii –xxviii, 2008.

[TDDN00] Tichelaar, Sander, Stéphane Ducasse, Serge Demeyer und Oscar Nierstrasz: *A Meta-model for Language-Independent Refactoring*. International Symposium on Principles of Software Evolution, Seiten 157 – 167, November 2000.

[TMM08] Taentzer, Gabriele, Dirk Müller und Tom Mens: *Specifying Domain-Specific Refactorings for AndroMDA Based on Graph Transformation.* In: *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, Seiten 104–119, Berlin, Heidelberg, 2008. Springer.

[@Tri10] Triskell: *Triskell Project.* `http://www.irisa.fr/triskell/home_html-en`, Mai 2010. Besucht am 2. Juni 2010.

[Var06] Varró, Dániel: *Model Transformation by Example.* In: Nierstrasz, Oscar, Jon Whittle, David Harel und Gianna Reggio (Herausgeber): *MoDELS*, Band 4199 der Reihe *Lecture Notes in Computer Science*, Seiten 410–424. Springer, 2006.

[vdSJM07] Straeten, Ragnhild van der, Viviane Jonckers und Tom Mens: *A formal approach to model refactoring and model refinement.* Software and System Modeling, 6(2):139–162, 2007.

[vGSMD03] Gorp, Pieter van, Hans Stenten, Tom Mens und Serge Demeyer: *Towards automating source-consistent UML Refactorings.* In: *In Proceedings of the 6th International Conference on UML -– The Unified Modeling Language*, Band 2863, Seiten 144–158, 2003.

[vKCKB05] Kempen, Marc van, Michel Chaudron, Derrick Kourie und Andrew Boake: *Towards proving preservation of behaviour of refactoring of UML models.* In: *SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, Seiten 252–259, Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[Wir86] Wirth, Niklaus: *Compilerbau - Eine Einführung.* Teubner, 1986.

[WSKK07] Wimmer, Manuel, Michael Strommer, Horst Kargl und Gerhard Kramler: *Towards Model Transformation Generation By-Example.* In: *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, Seite 285b, Washington, DC, USA, 2007. IEEE Computer Society.

[ZLG05] Zhang, Jing, Yuehua Lin und Jeff Gray: *Generic and Domain-Specific Model Refactoring using a Model Transformation Engine.* In: *Volume II of Research and Practice in Software Engineering*, Seiten 199–218. Springer, 2005.

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 12. Juli 2010