



TU Wien

Business Informatics Group

Institut für Softwaretechnik und Interaktive Systeme

Model Transformation By-Example: An Eclipse based Framework

**Diplomarbeit
zur Erlangung des akademischen Grades
eines Magister rerum socialium oeconomicarumque**

eingereicht
bei o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel
mitbetreuende Assistenten:
Univ.-Ass. Mag. Dr. Manuel Wimmer
Proj.-Ass. Mag. Michael Stommer

Abraham Müller, Bakk. rer. soc. oec.
Gerald Müller, Bakk. rer. soc. oec.

Wien, 19. Juni 2008

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum Eigenhändige Unterschrift

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum Eigenhändige Unterschrift

Danksagung

Wir möchten uns bei o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel und Mag. Michael Strommer sowie Mag. Manuel Wimmer für die Anregung zu dieser Diplomarbeit und für die Unterstützung bei der Erstellung der Arbeit bedanken. Weiters danken wir unseren beiden Familien für die mentale und finanzielle Unterstützung durch unser gesamtes Studium hindurch. Außerdem bedanken wir uns bei den KorrekturleserInnen: Monika Müller, Werner Müller und Katharina Puganigg.

Kurzfassung

Viele der momentan existierenden Ansätze sowie Sprachen zur Modelltransformation sind metamodel-basiert. Sie setzen detaillierte Kenntnisse der Metamodellebene und deren Syntax voraus. Meist wird die Transformation durch ein vollständiges Regelwerk auf der Metaebene bewerkstelligt. Dazu müssen dem Benutzer entweder die Konzepte dieser Ebene bekannt sein, oder er vertraut auf eine eventuell vorhandene Lösung.

Einen neuen benutzerfreundlichen Ansatz dafür beschreibt *Model Transformation By-Example* (MTBE). Der Benutzer arbeitet mit der ihm vertrauten Syntax auf Instanzebene und kann dort Mappings zwischen Elementen zweier verschiedener Modelle definieren, aus denen automatisch Regeln generiert werden. Das Ergebnis der Analyse kann während der Erstellung der Korrespondenzen laufend überprüft werden, sodass es möglich ist, diese iterativ zu bearbeiten, um schlussendlich alle nötigen Regeln abzuleiten. Des Weiteren erlaubt ein solches Framework das Erweitern der Analyse und ermöglicht es somit, auf diesen Schritt direkt Einfluss zu nehmen.

In dieser Arbeit wird ein Framework für MTBE auf Basis des *Graphical Modeling Framework* (GMF) umgesetzt. Die Implementierung von MTBE wird in ein Eclipse Plug-in eingebettet. Mappings auf der Modellebene können mit Hilfe eines automatisch generierten GMF-Editors gezeichnet werden. Eine so genannte Analyser Komponente soll dann diese Mappings auswerten und daraus ein *Weaving Modell* beziehungsweise den ATL Code generieren.

Für den in dieser Arbeit vorgestellten Prototyp wurden drei Analysealgorithmen entwickelt. Die Implementierung wurde so umgesetzt, dass sie über eine integrierte Schnittstelle noch beliebig erweiterbar ist. Zusätzlich zu diesem Prototyp wurden zwei GMF-Editoren für die beiden Modellierungssprachen UML und ER entwickelt, welche als Beispielanwendung herangezogen werden, um Funktion, sowie entwickelte Algorithmen anhand einer Transformation zwischen den Sprachen UML und ER zu testen und auch zu präsentieren.

Abstract

Many of the existing approaches for model transformation are metamodel-based and thus require detailed knowledge of the corresponding concepts and syntax. A solution often consists of a complete set of rules, which map meta elements between domains. These rules have to be provided by a third-party or the user herself. The latter case requires high levels of familiarity with both metamodels.

Model Transformation By-Example (MTBE) introduces a new concept. It enables the user to make use of her existing knowledge of the syntax and notation at the M1 level by deriving the rules automatically from mappings on the instance of a metamodel. The result of these generated rules and therefore the level of completeness can be applied continuously during the process and may lead to an iterative way of defining the necessary mappings. To be able to generate a complete rule set, the suggested framework is able to provide an extension mechanism to alter the behaviour of the whole analysing component.

This work proposes the implementation of an MTBE-based framework on top of the Graphical Modeling Framework (GMF). The framework is realized by utilising the plug-in capabilities of Eclipse. The graphical components used to create the diagrams and mappings are realized through a GMF editor. An extensible analyser is applied to translate those mappings to the next abstraction level and to provide a weaving model as well as a set of ATL (Atlas Transformation Language) rules for later use in transforming models in both directions.

The core of the analyser consists of three algorithms, which provide the basic logic and can be extended to handle additional domains. The Eclipse mechanism through extension points is used to provide this capability. For testing and demonstration purposes two GMF Editors were developed, one for UML class diagrams and one for the ER domain.

Inhaltsverzeichnis

1	Einleitung	15
1.1	Motivation für <i>Model Transformation By-Example</i>	15
1.2	Motivation für diese Arbeit	16
1.3	Zielsetzung	17
1.4	Aufbau	17
2	Grundlagen	19
2.1	Modellgetriebene Softwareentwicklung	19
2.1.1	Übersicht	19
2.1.2	Object Management Group (OMG)	21
2.1.3	Grundlagen von MDA	22
2.1.4	MOF - Meta Object Family [28]	22
2.1.5	UML	24
2.1.6	Abstraktionsebenen der Modelle in MDA	27
2.2	EMF - Eclipse Modeling Framework	29
2.2.1	Einleitung	29
2.2.2	Ecore (Metamodell)	29
2.2.3	Modellerstellung	32
2.2.4	Programmieren im EMF	38
2.3	GMF - Grafical Modeling Framework	45
2.3.1	Einführung	45
2.3.2	Beispiel	46
2.3.3	Domain Model	47
2.3.4	Graphical Definition Model	48
2.3.5	Tooling Definition Model	49
2.3.6	Mapping Definition Model	50
2.3.7	Code Generierung	51
2.4	Modelltransformationen	51
2.4.1	Einleitung	51
2.4.2	Atlas Transformation Language	55
2.5	Atlas Model Weaver	57

2.5.1	Core Weaving Metamodel	57
2.5.2	AMW Weaving Tool	58
2.6	By-Example Ansätze	60
2.6.1	Einleitung	60
2.6.2	Query-By-Example	61
2.6.3	Programming-By-Example	63
3	Model Transformation By-Example	65
3.1	Einleitung	65
3.2	Model Transformation Generation By-Example	66
3.2.1	Konzept	66
3.2.2	Framework	68
3.2.3	Integration von GMF	72
3.3	MTBE By-Example - UML2ER	75
3.3.1	Definition der Mappings	75
3.3.2	Generieren der Objekte	76
3.3.3	Setzen der Attributwerte	77
3.3.4	Verbinden der Objekte	78
3.3.5	Erzeugen der ATL Regeln	78
4	Implementierung des MTBE Prototyps	79
4.1	Übersicht	79
4.2	Systemdokumentation	81
4.2.1	Paketstruktur	81
4.2.2	Projekt Wizard	83
4.2.3	Merger	90
4.2.4	Analyse	98
4.2.5	Implementierte Algorithmen	107
4.2.6	Transformation	113
4.2.7	MTBE-Perspektiven	114
4.2.8	Erweiterbarkeit	116
4.3	Installationsanleitung	119
4.4	Benutzerdokumentation	120
4.4.1	Beschreibung der Funktionalität	120
4.4.2	Beschreibung der Benutzeroberfläche	120
4.4.3	Programmooptionen	121
4.4.4	Konfigurationsmöglichkeiten	122
4.5	Beispiel	122
4.5.1	Einleitung	122

4.5.2	Erstellen der Ausgangseditoren	123
4.5.3	Generieren des Diagramm Plug-ins	123
4.5.4	Zeichnen der Modelle	124
4.5.5	Einzeichnen der Mappings	126
4.5.6	Ausführen der Analyse	126
4.5.7	<i>Weaving Modell</i>	128
4.5.8	Generieren des ATL Codes	128
4.5.9	Test der Transformation	131
4.6	Kritische Betrachtung	131
5	Ausblick	135
6	Zusammenfassung	141
A	Anhang	143
A.1	EMF ER Metamodell	143
A.2	GMF und weitere Ausschnitte	146
	Literaturverzeichnis	149

Abbildungsverzeichnis

2.1	Auszug aus UML Klassendiagramm (von [29])	23
2.2	Stereotypen in UML	25
2.3	Erweitern des UML Metamodells	26
2.4	Taxonomie der MDA Modelle nach [12]	28
2.5	Klassenhierarchie des Ecore Metamodells [42]	30
2.6	Vereinfachtes ER-Metamodell	33
2.7	Beispiel - Student schreibt Arbeit	42
2.8	ER Modell im EMF Editor	43
2.9	Übersicht der GMF Komponenten [34]	45
2.10	Metamodell eines einfachen UML-Editors	47
2.11	EMF - Metamodell eines einfachen UML-Editors	48
2.12	Ausschnitt aus der Graphical Definition	49
2.13	Mapping Definition	50
2.14	Basis Konzept einer Modelltransformation [6]	52
2.15	Eingeteilt Modelltransformationen in Form eines Merkmaldiagramms	54
2.16	ATL Transformation	55
2.17	AMW Core Meta Model [7]	58
2.18	AMW Architektur [10]	59
2.19	Atlas Model Weaver	60
3.1	Verwendetes Beispieldiagramm für UML	65
3.2	Verwendetes Beispieldiagramm für ER	66
3.3	Beziehung zwischen abstrakter und konkreter Syntax	67
3.4	MTBE Framework	68
3.5	Komponentendiagramm des MTBE Frameworks	69
3.6	Standardmäßig zur Verfügung stehende Analyzer Pattern	71
3.7	MTBE Workbench: Screenshots von Ansicht 1	73
3.8	MTBE Workbench: Screenshots von Ansicht 2	73
3.9	MTBE Workbench: Screenshots von Ansicht 3	74
3.10	MTBE Workbench: Screenshots von Ansicht 4	74
3.11	MTBE für eine UML2ER bzw ER2UML Transformation	76

4.1	Überblick MTBE Implementierung	80
4.2	Paketstruktur	81
4.3	Arbeitsweise des Wizards	84
4.4	Übersicht des Zusammenfügens zweier Editoren	90
4.5	Pakete und Vererbung im gemeinsamen Metamodell	93
4.6	Arbeitsweise der Analyse	99
4.7	Das Weaving Meta Model für MTBE	101
4.8	Beispiel für Benutzermappings	109
4.9	Ein Mapping das nicht vom ContainmenReasoning gefunden wird	110
4.10	Arbeitsweise der Methode <i>TransformationAction</i>	113
4.11	MTBE Debug Perspective	116
4.12	MTBE Developer Perspective	117
4.13	Screenshot der MTBE Menuleiste	121
4.14	Überblick über den MTBE Prozess	122
4.15	MTBE Wizard - Erstellen eines neuen MTBE Projektes	124
4.16	MTBE Wizard - Exportieren des neuen Diagramm Plug-in	125
4.17	Der neue Diagrammeditor für beide Modelltypen (<i>example.er2uml_diagramm</i>)	126
4.18	Unser Diagramm in Baumdarstellung (<i>example.er2uml</i>)	127
4.19	Dialogbox für die Auswahl der Algorithmen	127
4.20	Das generierte <i>Weaving Modell</i>	128
4.21	Dialogbox für die Auswahl eines Modells zum Testen der Transformation	129
5.1	Beispiel Attribut- und Referenzanalyse	137
A.1	Graphical Definition für UML Editor	146

Tabellenverzeichnis

2.1	QBE-Query um die Betreuer aller Studenten mit dem Namen "Müller" zurückzugeben.	62
2.2	QBE-Query um alle Studenten mit dem Namen "Müller" zu löschen.	62
4.1	Verzeichnisstruktur eines MTBE-Projektes	85
4.2	Ressourcen die geladen oder neu erstellt werden müssen	92

Listings

2.1	Annotiertes Interface Entity	34
2.2	ER Metamodell als XML Schema	35
2.3	Ausschnitt aus EntityImpl.java	37
2.4	Setzen von Referenzen	37
2.5	Erzeugen einer Resource	39
2.6	Pakete erstellen	40
2.7	Klassen erzeugen	40
2.8	Referenzen erstellen	41
2.9	Ressourcen speichern	41
2.10	Student und Diagram erzeugen	42
2.11	Eine Relation erzeugen	43
2.12	Ein Modell laden	44
2.13	Eine Entität per Reflection erzeugen	44
2.14	ATL Beispieltransformation UML2ER	55
2.15	ATL Code	56
2.16	ATL Code	56
2.17	ATL Code	56
2.18	SQL-Query	62
4.1	Kopieren von Dateien aus einem laufenden Plug-in	86
4.2	Erstellen des GMF <i>Generator Modells</i>	88
4.3	Exportieren eines Projektes	89
4.4	Domain Elemente für das “Simple Mapping”	93
4.5	Programmatisch Erstellung eines Generatormodells	97
4.6	Durchführen der Analyse durch einzelne Algorithmen	100
4.7	Setzen der <i>left</i> und <i>right</i> Referenz des <i>Weaving Modell</i>	101
4.8	Erstellen einer “Full Equivalence”	102
4.9	Erstellen einer <i>Full Equivalence</i> als <i>owned Reference</i>	103
4.10	Erstellen einer “Conditional Equivalence”	105
4.11	Erstellen einer “Conditional Equivalence”	106
4.12	Finden von Mappings durch das “SimpleReasoning”	108

4.13	Definition der <i>Perspectives-Extensions</i> im plugin.xml	115
4.14	Extension Point des MTBE Plug-ins	116
4.15	Hinzufügen neuer Algorithmen im plugin.xml	117
4.16	ATL Beispieltransformation UML2ER	129
4.17	ATL Beispieltransformation ER2UML	130
A.1	Annotiertes Interface - Diagram	143
A.2	Annotiertes Interface - Entity	143
A.3	Annotiertes Interface - Relationship	144
A.4	Annotiertes Interface - Role	144
A.5	Annotiertes Interface - Attribute	144
A.6	ER Metamodell als XML Schema	145
A.7	amw.prop-File für den Weaving Editor	146
A.8	run.xml Script	147
A.9	transformation.xml Script	148

Kapitel 1

Einleitung

1.1 Motivation für Model Transformation By-Example

Die Anforderungen in der Softwareentwicklung haben sich in den letzten Jahrzehnten stark verändert. Es wird immer wichtiger mit möglichst kurzer Durchlaufzeit Software zu entwerfen, anzupassen oder weiterzuentwickeln, ohne dabei Qualitätseinbußen in Kauf nehmen zu müssen. Ein möglicher Ansatz diesen Anforderungen gerecht zu werden besteht darin, Software auf einer möglichst abstrakten Ebene zu entwickeln. Dieser wird in der modellgetriebenen Softwareentwicklung (MDA, MDSD) [12] verfolgt. Software wird dabei nicht in einer bestimmten Programmiersprache entwickelt, sondern anhand von Modellen beschrieben.

Ein wichtiger Bestandteil der modellgetriebenen Softwareentwicklung ist das Thema Modelltransformation. Modelltransformation dient im Allgemeinen dazu, Modelle in ein andere Modelle zu überführen, Modelle zu erweitern, mehrere domänenspezifische Modelle zusammenzuführen oder um aus Modellen Quellcode zu generieren. Kurz gesagt geht es um den Vorgang, aus einem bestimmten Input, der aus einem oder mehrerer Modellen besteht, über definierte Regeln den gewünschten Output zu erzeugen. Im Laufe der Zeit haben sich rund um die modellgetriebene Softwareentwicklung verschiedene Ansätze und Sprachen zur Modelltransformation entwickelt. Der Großteil dieser baut auf dem Metamodell auf. Das heißt die Transformationsregeln werden auf der Metaebene beschrieben. Diese Herangehensweise ist laut Wimmer et al. [49] auf Grund von zwei Hauptaspekten nicht benutzerfreundlich:

- *Problem 1* - Es besteht eine Diskrepanz in der automatisierten Darstellungsform und in der für Anwender aufbereiteten Ansicht von Modellen. Ein Modellierer arbeitet mit der konkreten Syntax, der Computer benötigt

hingegen Modelle die in der abstrakten Syntax beschrieben sind. Das Problem besteht darin, dass momentan verfügbare Transformationssprachen Regeln auf der abstrakten Syntax definieren.

- *Problem 2* - Es kommt vor, dass nicht alle Sprachkonzepte eines Metamodells klar erkennbar sind, sie sind “versteckt”. Wimmer et al. bezeichnen dieses Phänomen als *Concept Hiding*. Ein Beispiel dafür ist ein UML Attribut, welches im *Property* versteckt ist. Ein *Property* wird erst zum *Attribut* wenn eine Verbindung zu einer *Klasse* besteht.

Diese Punkte führen unter anderem dazu, dass Modelltransformationen meist sehr aufwändig und zeitintensiv sind.

Daraus entstand die Idee der *Business Informatics Group*, Modelltransformationen mit dem *By-Example* Ansatz zu kombinieren [49]. Hierbei beschreibt der Benutzer Mappings auf Instanzebene. Dies geschieht durch Verbindungen zwischen Elementen zweier konkreter Modelle. Diese werden analysiert und automatisiert in Transformationsregeln der Metaebene umgewandelt.

Diese Vorgehensweise hat mehrere Vorteile. Der Anwender benötigt kein Wissen über die darunter liegenden Metamodelle. Er muss sich nur auf der Instanzebene mit den konkreten Modellen auseinandersetzen. Zusätzlich wird durch das automatische Erstellen der Transformationsregeln die Notwendigkeit eine weitere Sprache zu erlernen, eliminiert. Der MTBE Ansatz ist dadurch wesentlich anwenderfreundlich als das manuelle Spezifizieren von Regeln auf der Metaebene.

1.2 Motivation für diese Arbeit

In der heutigen Zeit sind aus einem Softwareentwicklungsprozess Modelle nicht mehr wegzudenken. Aber auch in anderen Bereichen wie Geschäftsprozessmodellierung kommen Modelle zum Einsatz. Überall dort wo sie eingesetzt werden, und mehr als nur zu Dokumentationszwecken dienen, spielen Modelltransformationen eine wichtige Rolle. Trotzdem sind Modelltransformationen immer noch ein aufwändiger und zeitintensiver Prozess.

Das Konzept für *Model Transformation By-Example* (MTBE) befindet sich in einem fortgeschrittenen Status, und erfordert den MTBE Prozess praktisch umzusetzen um eine optimale Weiterentwicklung und Verfeinerung zu gewährleisten. Micheal Strommer et al. [40] beschreiben ein Framework für einen *Proof*

of *Concept* Prototyp. Der Vorteil eines Prototyps für MTBE liegt vor allem in den folgenden zwei Punkten:

- Die Validierung und Evaluierung der bestehenden MTBE Techniken ([49], [40], [39]) und des MTBE Prozesses als Ganzes.
- Das Bereitstellen einer Experimentierumgebung für die *Mapping Sprache* und der *Reasoning Algorithmen*.
- Die Evaluierung bestehender modellbasierter Frameworks und Technologien und zu welchem Grad diese als Grundlage für eine Umsetzung geeignet sind.

1.3 Zielsetzung

Diese Diplomarbeit beschreibt die praktische Umsetzung des *Model Transformation By-Example* Ansatzes und dokumentiert diese in Hinblick auf bestehende Arbeiten detailliert. Das Ergebnis dieser Arbeit ist ein lauffähiger Proof-of-Concept Prototyp auf Basis des *Graphical Modeling Framework* (GMF) [44] eingebettet in ein Eclipse Plug-in.

Dem Prototyp wurden mehrere Analysealgorithmen zur Verfügung gestellt, die zusätzlich über eine Schnittstelle beliebig erweitert werden können.

Zusätzlich zu diesem Prototypen wurden zwei GMF-Editoren, ein vereinfachtes UML Klassendiagramm und ein ER Diagramm entwickelt, die als Beispielanwendung herangezogen werden, um Funktionsweise, sowie entwickelte Algorithmen anhand einer UML2ER sowie ER2UML Transformation zu testen und auch zu präsentieren.

1.4 Aufbau

Diese Diplomarbeit wurde in drei Teile gegliedert. Sie beginnt mit einer Ausarbeitung essentieller Grundlagen rund um die Themen Modellgetriebene Softwareentwicklung, Modelltransformation und den *By-Example* Ansätzen jeweils mit Fokus auf die essenziellen Bausteine für die Implementierung des MTBE Frameworks, vor allem die Eclipse Frameworks GMF und EMF.

Der zweite Abschnitt behandelt die *Model Transformation By-Example* wie von Wimmer et al. beschrieben [49]. Weiteres wird neben dem Konzept auch das Framework für den *Proof of Concept* Prototypen vorgestellt.

Der letzte Abschnitt beschreibt die Implementierung des *MTBE Framework*. Neben einer detaillierten Beschreibung der einzelnen Komponenten wird die Umsetzung mit einem anschaulichen Beispiel aus Anwendersicht erläutert. Abschließend folgen noch eine kritische Betrachtung und ein Ausblick.

Im Anhang dieser Diplomarbeit finden sich neben dem Literatur-, Abbildungs-, Tabellenverzeichnis auch Listings von essentiellen Ausschnitten aus dem Quellcode, auf die in dieser Arbeit eingegangen wird.

Kapitel 2

Grundlagen

2.1 Modellgetriebene Softwareentwicklung

2.1.1 Übersicht

In den letzten Jahrzehnten dringt die IT-Technologie immer mehr in die Geschäftsprozess von Unternehmen ein. Durch die unternehmensübergreifende Vernetzung dieser Prozesse werden immer komplexere Lösungen nötig, die möglichst schnell, billig und in ausreichender Qualität zur Verfügung stehen sollten. Die Anwendung von Mustern und Standards hat sich in anderen Disziplinen bewährt und auch in der Softwareindustrie werden diese Konzepte vermehrt angewandt um die Erfolgsquote von Softwareprojekten zu erhöhen (vgl [12]).

Software stellt meist ein Modell von realen Objekten und Prozessen dar und deren Entwicklung weist mehrere Dimensionen, wie etwa Kosten, Qualität oder Laufzeit auf. Eine Änderung einer dieser Dimensionen beeinflusst dabei die Restlichen. Da Software eng mit der technologischen Entwicklung zusammenhängt ist es in der Vergangenheit immer wieder zu einer Veränderung des Einflusses der einzelnen Dimensionen auf das Gesamtergebnis gekommen.

Der erste größere Entwicklungssprung war der Wechsel von Maschinencode zu Assemblersprachen. Mit der dadurch erhöhten Abstraktion war es fortan nicht mehr zwingend nötig Programme völlig zu ändern wenn sich die Technologie (beispielsweise der Befehlssatz des Prozessors) änderte. Nur der Compiler musste angepasst werden, welcher dann alle Programme für die neue Plattform richtig übersetzen konnte. Darauf folgten weitere Stufen die die Abstraktion immer weiter erhöhten. Die zurzeit aktuellen Konzepte sind Objektorientierung und deren Modellierungsansätze, allen voran die Unified Modeling Language (UML), die komponentenbasierte Entwicklung oder der Einsatz von Middleware, um nur eini-

ge zu nennen. Diese Konzepte befinden sich wieder eine Abstraktionsebene höher und haben die Entwicklung und Wartung wiederum effizienter gemacht. Trotz des Einsatzes dieser Konzepte tauchen laut [12] und [15] in Entwicklungsprozessen oft einige der nachfolgenden Probleme auf.

Abstraktion Trotz der bereits angewandten Abstraktion in der Implementierung müssen sich die Entwickler sehr stark mit der verwendeten Technik auseinandersetzen. Erfahrungsgemäß ist diese ständigen Änderungen unterworfen und erfordert stetigen Lernaufwand, während sich der fachliche Bereich nicht in dem gleichen Ausmaß weiterentwickelt. Somit sind Entwicklungsressourcen zu einem gewissen Grad permanent ausgelastet um diese Entwicklung zu kompensieren. Falls ein Projekt die fachliche Entwicklung und die Implementierung personell trennt, treten zusätzlich noch Kommunikationsprobleme auf, da die unterschiedlichen Teams auf unterschiedlichen Ebenen agieren.

Methodische Brüche Zwischen den Modellen, die zurzeit hauptsächlich zu Dokumentationszwecken erstellt werden und jener die der Implementierung dienen, besteht ein methodischer Bruch. Werden die Modelle nicht auch während der Entwicklung ständig angepasst, sind sie nach kurzer Zeit unbrauchbar und somit auch in späteren Projekten schwer wieder verwendbar. Natürlich leidet auch die Wartbarkeit erheblich wenn die Dokumentation Fehler aufweist und die Implementierung nicht auf die Anforderungen zurückgeführt werden kann.

Wiederverwendbarkeit Wie bereits erwähnt unterscheiden sich die Veränderungen der fachlichen Domäne von der technischen Weiterentwicklung. Daher ist es wünschenswert Fachkonzepte in anderen Projekten wieder zu verwenden. Auch hat die Wiederverwendbarkeit durch die Einführung von Objektorientierung nicht den erhofften Quantensprung erfahren. Oft ist das Wissen der Entwickler über den Ablauf von Geschäftsprozessen in der Implementierung enthalten. Durch Abstraktion auf die fachliche Ebene kann dieses Wissen dokumentiert und für andere Entwicklungen verfügbar gemacht werden.

Wartungsaufwand Die ständige Evolution der verfügbaren Technik generiert Druck zur Modernisierung der eingesetzten Anwendungen. Nach der Einführung einer neuen Softwarelösung ändern sich bestehende Anforderungen oder es kommen neue hinzu. Diese werden umgesetzt und in die Anwendung integriert. Dadurch wird die Anwendung immer schwieriger zu warten und muss schließlich überarbeitet werden. Die andauernde Migration und Wartung verursacht Kosten und mit der Anzahl der zu wartenden Systeme sinkt das Budget für Neuentwicklungen. Oftmals werden von Entscheidungsträgern auch die technologischen Entwicklungen falsch eingeschätzt was zu einem hohen Änderungsaufwand in späteren Phasen führen kann.

Qualität Oft wird mit zunehmender Komplexität und den engen zeitlichen Grenzen eines Softwareprojektes besonders durch die Entwickler alles abseits der Implementierung als *Overhead* angesehen, der eingespart werden kann. In anderen Disziplinen wie zum Beispiel der Architektur wird eine solide Planung als unverzichtbar angesehen. Das Fehlen einer solchen führt meist zu Qualitätseinbußen und viel zu oft zu Projektabbrüchen.

Plattformunabhängigkeit Die Entwicklung von MDA wurde von dem Gedanken geleitet von der Zielpattform unabhängig zu werden. Idealerweise kann ein entworfenes Modell in jede Plattform (.NET, Java EE...) übersetzt werden. Leider funktioniert dies in der Praxis noch nicht ganz so einfach, da noch immer sehr viele Aspekte der Zielpattform in die Architektur einfließen und viele Funktionen von den Generatoren noch nicht, oder nur teilweise automatisch erzeugt werden können.

Die Entwicklungen seit den Anfangszeiten der Computertechnologie haben gezeigt, dass Abstraktion eine gute Möglichkeit darstellt die Effektivität in der Softwareerstellung zu verbessern. Die Zahl der eingesetzten Programmiersprachen zeigen darüber hinaus, dass nicht nur eine *beste* Lösung für ein Problem besteht. Zur Lösung der vorhandenen Probleme versucht MDA die nächste Ebene der Abstraktion durch Modelle zu erreichen. Dabei rücken diese ins Zentrum der Entwicklung und sind im Idealfall ausreichend um eine Anwendung zu generieren.

2.1.2 Object Management Group (OMG)

Die OMG [27] ist eine *non-profit* Vereinigung der IT-Industrie. Jede Organisation kann sich an der OMG beteiligen, wobei die führenden Unternehmen der Industrie Vertretungen in das *Board of Directors* entsenden. Seit 1989 hat die OMG zahlreiche Standards wie etwa UML, CORBA oder MDA entwickelt. Die Entwicklung erfolgt in so genannten *Task Forces*, wobei das Stimmrecht der beteiligten Organisationen unabhängig von ihrer Größe ist. Alle vorgeschlagenen Standards können öffentlich eingesehen werden, Kommentare können aber nur von Mitgliedern abgegeben werden. Viele der vorgeschlagenen Standards werden anschließend von der ISO zertifiziert. Für weitere Informationen über diese Organisation ist ihre Webseite [27] ein guter Startpunkt, wo unter anderem sowohl beschlossene als auch vorgeschlagene Spezifikationen veröffentlicht werden.

2.1.3 Grundlagen von MDA

MDA enthält viele bereits bewährte Konzepte und vereint diese mit einigen neuen Ansätzen zu einem Standard. So gibt es beispielsweise im Embedded Bereich bereits Sprachen, wie die *Real-Time Object-Oriented Modeling Language* welche aus einem Modell direkt Anwendungen erstellen können. MDA verfolgt den Ansatz aus mehreren Modellen die nötigen Informationen zu extrahieren um eine Anwendung zu generieren. Da sich die Musterorientierung in der Softwareentwicklung bewährt hat, haben diese auch in MDA Einzug gefunden. Es gibt unterschiedliche Klassen von Mustern, am bekanntesten dürften die Entwurfsmuster der *Gang of Four* [9] sein. Muster werden von den Entwicklern entweder explizit modelliert, können aber auch vom Generator erkannt werden und in eine entsprechende Implementierung überführt werden. Aspektorientierung wird in MDA bei der Trennung von fachlichen Modellen und plattformspezifischen Modellen berücksichtigt. Aspekte, wie zum Beispiel Persistenz, werden in den fachlichen Modellen Objekten zugeordnet. Wie dies in der Implementierung umgesetzt wird hängt von der Plattform ab. Beispielsweise bieten manche JEE Anwendungsserver Persistenzunterstützung durch integrierte Frameworks an.

Im Zentrum von MDA steht die MOF [28] und somit sind alle darauf basierenden Modelle in MDA einsetzbar. Im Folgenden wird nun auf die *Meta Object Family* eingegangen. Des Weiteren wird UML als weit verbreiteter Standard in diesem Zusammenhang im Detail behandelt. Abschließend werden noch die unterschiedlichen MDA Modelltypen vorgestellt.

2.1.4 MOF - Meta Object Family [28]

Der MOF-Standard [25] wurde von der OMG 1997 verabschiedet. Da Modelle unterschiedliche Bereiche der Realität abbilden müssen, muss es auch mehrere Modellierungssprachen geben. Diese werden über ihr Metamodell spezifiziert. Anstatt nun alle existierenden Modellkonstrukte in ein einziges großes Metamodell zusammenzufassen, entschied sich die OMG eine Möglichkeit zu schaffen alle diese Konstrukte einheitlich zu beschreiben. Über MOF ist es möglich verschiedenste Metamodelle zu definieren. Die Konstrukte dazu kommen vom UML Klassendiagramm mit Klassen im Zentrum, Attributen um Eigenschaften zu beschreiben und Assoziationen um Beziehungen abzubilden. Abbildung 2.1 zeigt als Beispiel einen Auszug des Metamodells vom UML-Klassendiagramm.

MOF spezifiziert vier Metaebenen M3 bis M0, wobei jedes Modell mit Ausnahme von M3 eine Instanz des jeweils darüber liegenden Metamodells ist. M3

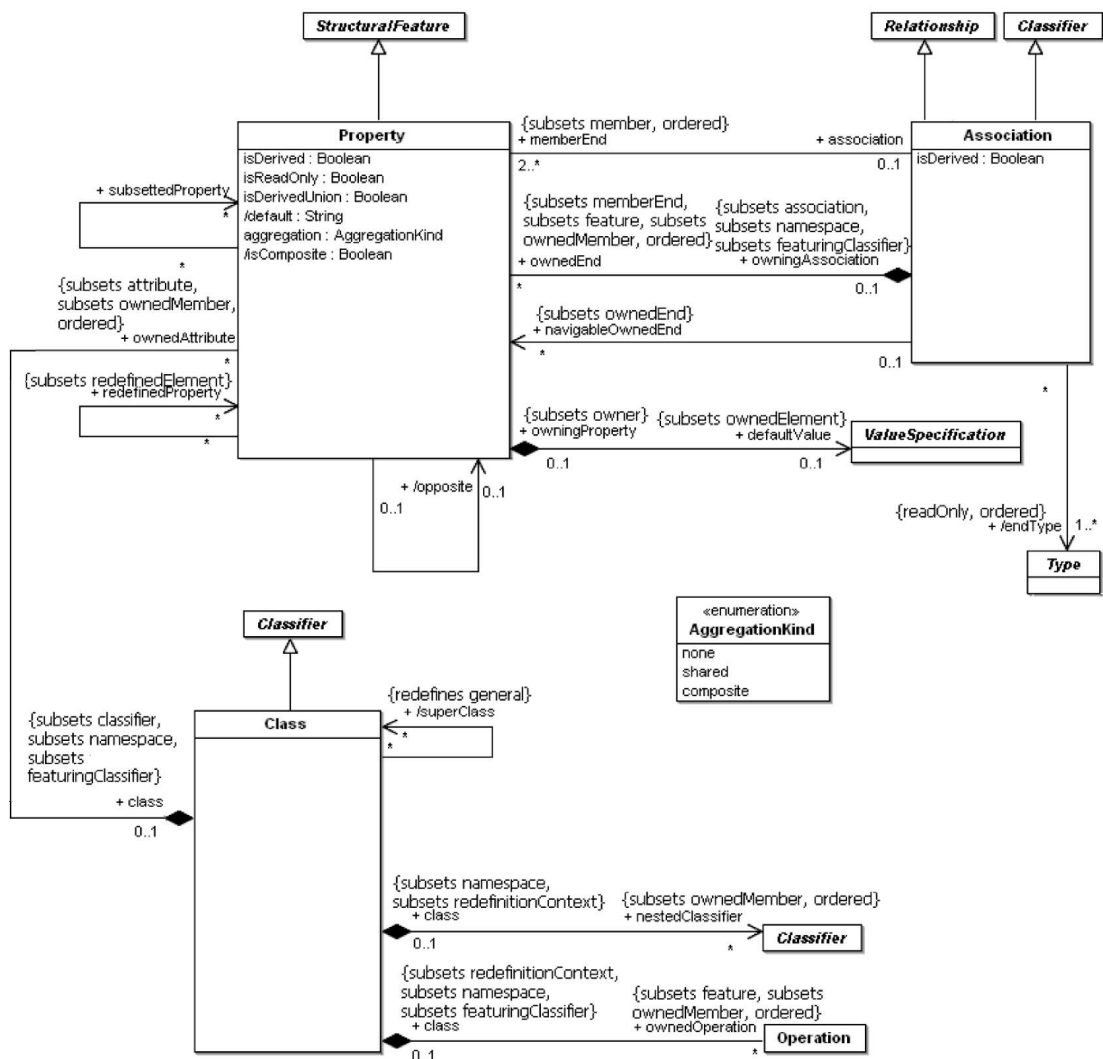


Abbildung 2.1: Auszug aus UML Klassendiagramm (von [29])

wiederum ist eine Instanz von M3, was heißt, dass MOF sein eigenes Metamodell ist, sich also selbst beschreibt.

- *M3* Diese Ebene ist MOF. Mit Hilfe der MOF Konstrukte lassen sich Metamodelle für viele verschiedene Anwendungsbereiche definieren.
- *M2* Metamodelle werden in dieser Ebene mithilfe der von MOF bereitgestellten Konstrukte beschrieben. Das Modell aus Abbildung 2.1 beispielsweise befindet sich auf dieser Ebene indem es in MOF beschreibt welche Elemente in einem Klassendiagramm vorkommen können.
- *M1* In dieser Ebene werden schließlich die Modelle selbst definiert. Dies wäre beispielsweise die Beschreibung einer Software als Klassendiagramm. Die meisten Entwickler werden am ehesten mit dieser Ebene vertraut sein.

Daher wird in MTBE die Definition der Mappings in dieser Ebene angesiedelt.

- *M0* Die unterste Ebene besteht aus Runtime Objekten wie zum Beispiel einer konkreten Instanz einer Klasse im Speicher.

2.1.5 UML

Wie schon erwähnt können in MDA alle auf MOF basierenden Modelle angewandt werden. UML [29] scheint sowohl von der Industrie als auch von den Anwendern favorisiert zu werden [15]. UML wird in späteren Abschnitten dieser Arbeit als Modellierungssprache verwendet. Häufig wird sie auch als Teil eines MOF Ansatzes verwendet und soll darum hier kurz vorgestellt werden. Anfang der 90er Jahre wurden unzählige Softwareentwicklungsmethoden entworfen, die alle ihre eigene Modellierungssprache mitbrachten. UML entstand aus der Zusammenführung von mehreren Ansätzen und wurde von der OMG standardisiert. Heute wird sie von dieser gepflegt und weiterentwickelt.

Meist wird bei UML an spezifische Diagrammarten gedacht. Diese sind jedoch nur eine Ausprägung der abstrakten Syntax. UML spezifiziert nicht die konkrete graphische Notation eines Diagrammes, sondern vielmehr welche Konstrukte erlaubt sind. Daher kann ein UML Modell beispielsweise auch in XML, als Java Klassen oder als CORBA Objekten vorliegen, was für die automatisierte Verarbeitung und Transformation von besonderer Bedeutung ist. Zum Beispiel kann ein Attribut in einem Klassendiagramm innerhalb einer Klasse dargestellt werden oder in Java als Variable innerhalb der Klammern dieser Klasse. Beides ist eine Ausprägung der konkreten Syntax des abstrakten Konzeptes *Ownership* dieses Attributes. UML definiert auch nicht welches Diagramm wann zur Anwendung kommt oder für welche Domäne welches Diagramm zu verwenden ist. Ein weiterer Vorteil von UML ist die Erweiterbarkeit über Profile. So spezifiziert UML nicht eine Lösung für alle möglichen Anwendungsgebiete, sondern stellt vielmehr einen Mechanismus bereit UML für einen bestimmten Zweck anzupassen.

Profile

Profile sind eine spezielle Form eines Pakets, sie werden äquivalent dargestellt und zusätzlich mit dem Schlüsselwort “profile” gekennzeichnet.

MDA baut auf UML Profilen auf, da Modelle für eine Unzahl von Domänen erstellt werden müssen, noch dazu auf unterschiedlichen Abstraktionsebenen.

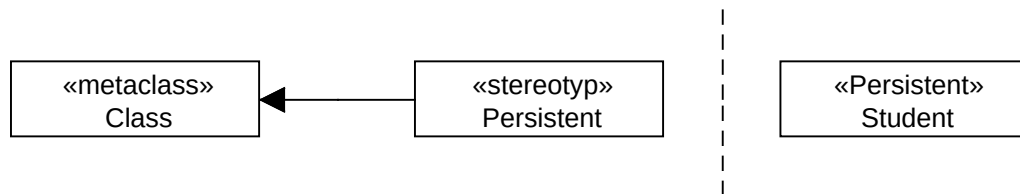


Abbildung 2.2: Stereotypen in UML

Reicht diese mächtige Eigenschaft nicht aus, können immer noch komplett eigene Metamodelle auf Basis von MOF erstellt werden, welche dann in MDA verwendet werden können. Dadurch büßt man aber die Möglichkeit ein, vorhandene Werkzeuge zu verwenden. Profile erlauben UML zu erweitern. So ist es möglich vorhandene Konstrukte mit zusätzlichen Informationen anzureichern oder Bedingungen zu verschärfen, nicht aber diese zu streichen oder aufzuweichen. Dies geschieht über Stereotypen, Bedingungen und *Tagged Values* (Tags). Einschränkungen können zwar über Bedingungen formuliert werden - beispielsweise kann spezifiziert werden dass Klassen keine Attribute vom Typ *String* haben dürfen - dies hat aber keine Auswirkungen auf das Metamodell.

- *Stereotypen* - Mithilfe von Stereotypen können Elemente näher spezifiziert werden. In der graphischen UML Notation werden Stereotypen durch “<<” und “>>” ausgezeichnet. Abbildung 2.2 zeigt die Definition eines Stereotypen <<Persistent>> welcher *Class* erweitert und dadurch einem Generator die Information liefert, dass dieses Element persistent gespeichert werden soll. In der M1 Ebene wird der Student dann mit diesem Stereotypen ausgezeichnet und sollte daraufhin einen Neustart des Systems überleben. Stereotypen können auch Attribute besitzen, welche im Modell als *Tagged Values* dargestellt werden. Seit UML 2.0 können diese auch andere Typen als *String* annehmen.
- *Bedingungen* - Wie auch in UML selbst können in Profilen Bedingungen modelliert werden um ein gewünschtes Verhalten zu erzwingen. Würde in Abbildung 2.2 eine Bedingung *required* an die Erweiterungsassoziation geknüpft werden, so könnte erreicht werden, dass alle Klassen eines Modells, welches dieses Profil importiert, persistent gehalten werden müssten was zugegebenerweise wenig Sinn macht.
- *Tagged Values* - *Tagged Values* können auch außerhalb von Stereotypen mo-

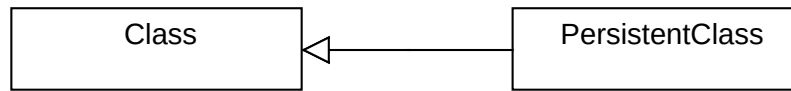


Abbildung 2.3: Erweitern des UML Metamodells

delliert werden. Diese werden, wie bei den Stereotypen mit `«` ausgezeichnet. Beispielsweise kann ein *Tagged Value* definiert werden, der das Metaelement Operation erweitert und besagt, dass diese Operation Transaktionen unterstützt und somit zurückgerollt werden muss.

Profile müssen nicht unbedingt das UML Metamodell referenzieren, sondern können auch ein bestehendes Profil weiter verfeinern. Viele der Konzepte die durch Profile verwirklicht werden, könnten natürlich auch über Konstrukte auf Instanzebene (M1) realisiert werden. Der Nachteil davon ist die Komplexität. Würde zum Beispiel das Transaktionsbeispiel in M1 modelliert werden, müsste dies durch Vererbung geschehen, was zur Folge haben würde, dass eine Klasse nur oder keine Operationen besitzen könnte, welche Transaktionen unterstützen. Dazu müsste man die Klassen entsprechend ihren Operationen aufteilen. Es ist leicht vorstellbar dass würde man mehrere solcher Eigenschaften gleichzeitig modellieren wollen, dies alles sehr unübersichtlich und komplex werden würde (siehe auch [12]).

UML mittels MOF erweitern

Eine weitere Möglichkeit besteht darin das UML-Metamodell selbst zu verändern. Wie schon erwähnt ist UML durch MOF definiert. Da das zugehörige Metamodell komplett verfügbar ist, kann dieses innerhalb der Möglichkeiten von MOF erweitert werden. Abbildung 2.3 zeigt wie mithilfe des MOF Konstruktes der Generalisierung die UML *Class* erweitert wird. Instanzen des veränderten Metamodells können nun anstatt vom Typ *Class* auch vom Typ *Persistent* sein, was wiederum einem Codegenerator als Eingabe dienen kann. Die direkte Veränderung des Metamodells hat den Vorteil, dass auch Reduzierungen möglich sind und der volle Umfang von MOF verwendet werden kann um komplexere Sachverhalte abzubilden oder Unzulänglichkeiten des UML Metamodells in einem gewissen Anwendungsgebiet auszubügeln. Der große Nachteil dieser Methode ist die fehlende Werkzeugunterstützung. Während Profile von vielen Werkzeugherstellern unterstützt werden trifft dies bei der Erweiterung des Metamodells nicht mehr zu.

Hersteller müssten die Werkzeuge speziell an das neue Metamodell anpassen oder erweitern, was aus Kostengründen eine eher unattraktive Alternative darstellt.

Aufbau von UML

UML ist nicht *eine* Spezifikation sondern eher eine Sammlung aus mehreren Spezifikationen, also einem UML-Stack, welcher als Kern die *InfrastructureLibrary* beinhaltet. Diese beschreibt die grundlegenden Konstrukte auf die alle anderen Spezifikationen aufbauen. Auch MOF ist eine dieser Spezifikationen und definiert, wie schon mehrmals erwähnt, das Metamodell für Metamodelle. Die *XML Metadata Interchange* Spezifikation dient als Format zur automatisierten Verarbeitung wie zum Beispiel der Codegenerierung, oder als Austauschformat zwischen UML Werkzeugen verschiedener Hersteller. Mit der *Object Constraint Language* lassen sich Modelle näher beschreiben oder Regeln zur Einschränkung erstellen. Grundlage für alle UML Elemente ist die *UML Superstructure* Spezifikation, die das eigentliche Metamodell aller Diagrammarten von UML darstellt. Sie unterteilen sich in Strukturdiagramme (statisch) und Verhaltensdiagramme (dynamisch).

2.1.6 Abstraktionsebenen der Modelle in MDA

Nachdem aus den Modellen schlussendlich ausführbarer Code entstehen und sie möglichst plattformunabhängig spezifiziert werden sollten, müssen diese irgendwann mit Informationen über die Eigenheiten der einzelnen Plattformen angereichert werden. Daher gibt es in MDA unterschiedliche Typen von Modellen auf unterschiedlichen Abstraktionsebenen, welche nach [12] wie in Abbildung 2.4 klassifiziert werden können. Folgende Erläuterungen folgen im Wesentlichen den Quellen [12, 15, 38].

Business Models Hierbei handelt es sich nicht um eine Modellart sondern es können mehrere Typen wie zum Beispiel das *Use-Case*, das Kollaborations- oder das *Activity* Diagramm zur Anwendung gebracht werden. Es werden neben automatisierten Aktivitäten auch andere, nicht von einem Softwaresystem vollzogene Abläufe dargestellt. Ziel ist ein Modell der zu bewältigenden Aufgabe, unabhängig davon, was computerunterstützt erledigt werden wird. Hier befindet sich auch eine Schnittstelle zwischen Entwicklern und Anwendern, wodurch es wichtig ist, dass beide Rollen die Modellierungssprache beherrschen und akzeptieren. Das *Domain Model* fällt somit in die Klasse der *Computation Independent Models* (CIM).

Requirement Models Mithilfe der *Requirements Models* wird festgelegt, was genau vom System erledigt werden muss. Es beschreibt welche Teile von der Software bereitgestellt werden müssen, lässt aber noch außer Acht, *wie* dies zu gesche-

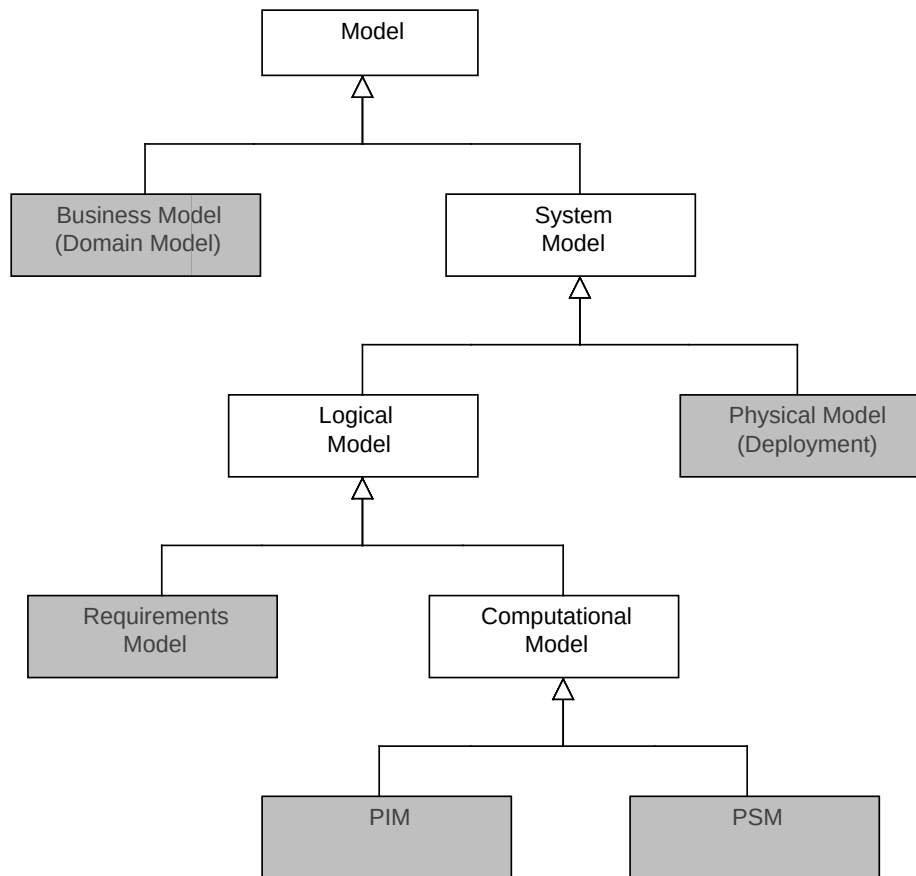


Abbildung 2.4: Taxonomie der MDA Modelle nach [12]

hen hat. Alle Anforderungen vom *Domain Model* die nicht automatisiert ablaufen sind hier nicht mehr zu finden, alle anderen Anforderungen werden jedoch genauer beschrieben. Auch hier kommen wieder Diagramme des UML Standards zur Anwendung. Beliebte sind vor allem das *Use-Case*, das *Activity* und das *Collaboration* Diagramm [12].

Platform Independent Models - PIM Hier wird die Ebene der CIMs verlassen. Es wird aber noch verlangt von der *Plattform* zu abstrahieren. Diese sollte bei der Definition des Modells genauer spezifiziert werden, bedeutet aber meist Unabhängigkeit von spezieller Middleware, Programmiersprachen, Form der Persistenz usw. Die OMG versteht darunter technologie-spezifische Standards (wie zum Beispiel Java EE, CORBA usw.) und Herstellerunabhängigkeit (.NET, WebSphere usw.). Es existieren zum Teil Werkzeuge die durch definierte Mappings das *Requirementsmodel* in ein PIM transformieren können, welche einen guten Ausgangspunkt für eigene Anpassungen liefern. In diesem Schritt wird das Modell durch erste technische Informationen angereichert. Dies kann zum Beispiel die Information sein, dass gewisse Entitäten auf verschiedenen Servern verteilt wer-

den, aber noch nicht durch welche Technologie dies geschieht (CORBA, RMI...). Dabei kommen speziell für gewisse Aspekte oder Domänen erstellte UML Profile zur Anwendung. Beispielsweise könnte ein UML Profil für Persistenz existieren und alle zu speichernden Entitäten werden mit den entsprechenden Stereotypen ausgezeichnet. Zu beachten ist, dass dies von der tatsächlich eingesetzten Technologie wie zum Beispiel Hibernate unabhängig ist.

Plattform Specific Models - PSM Im PSM wird schlussendlich auf die Zielplattform eingegangen. Die Modelle des PIM werden nun mit den entsprechenden UML Profilen für die jeweilige Plattform erweitert. So existiert beispielsweise das JSR26 des *Java Community* Prozesses welches ein UML Profil für *Enterprise Java Beans* zur Verfügung stellt.

2.2 EMF - Eclipse Modeling Framework

2.2.1 Einleitung

Das *Eclipse Modeling Framework* (EMF) [42] wurde ursprünglich als Implementierung der *Meta Object Facility* (MOF) [28] entwickelt und stellt nun eine Java Implementierung des MOF Kerns dar. (siehe auch EMOF [30]) EMF erweitert Eclipse um ein Framework welches unter anderem dazu verwendet werden kann Daten zwischen verschiedenen Eclipse Anwendungen auszutauschen. Es ermöglicht die Spezifikation des Modells in Java, XML Schema oder UML und kann die jeweils anderen automatisch generieren. Zusätzlich besteht die Möglichkeit Implementierungsklassen für Java zu erzeugen, welche bearbeitet und erweitert werden können. EMF ermöglicht die Rücktransformation des Codes in das Modell ohne dass diese Änderungen verloren gehen. Eigene Modelle können standardmäßig in XMI serialisiert und bei Bedarf geladen werden. Darüber hinaus verfügt das Framework über eine sogenannte *Reflection-API* mit deren Hilfe Modelle oder auch Instanzen manipuliert werden können.

2.2.2 Ecore (Metamodell)

Das Metamodell von EMF heißt Ecore. Alle Modelle in EMF müssen zu diesem konform sein. Ecore kann auch als vereinfachte Form des UML-Klassendiagramms angesehen werden [4]. Abbildung 2.5 stellt die Klassenhierarchie des Ecore Metamodells dar, wobei abstrakte Klassen oder Interfaces grau hinterlegt sind.

Die nachfolgende Erklärung ist stark an [4] angelehnt, mit dem Fokus auf die Elemente und Konzepte, die für die Implementierung von MTBE von besonderem Interesse sind.

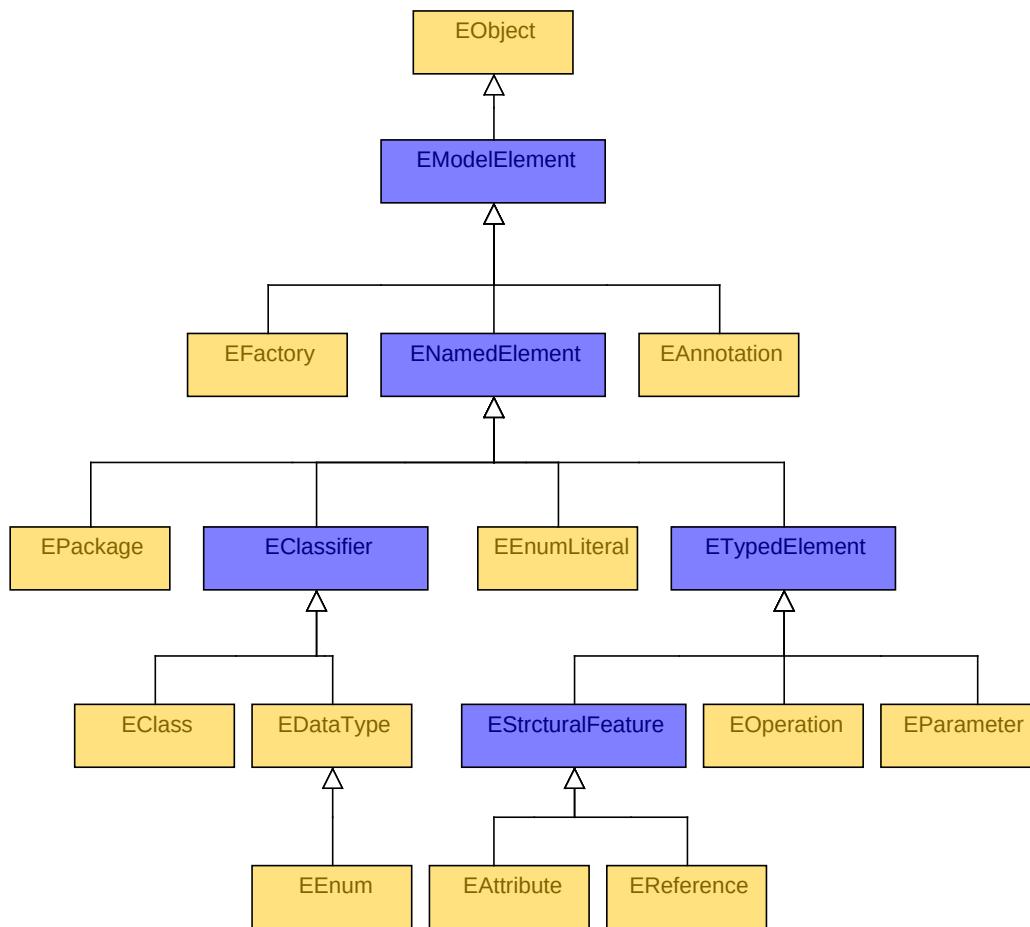


Abbildung 2.5: Klassenhierarchie des Ecore Metamodells [42]

EObject Ist konzeptuell vergleichbar mit dem *java.lang.Object* aus Java. Alle Objekte in EMF implementieren dieses Interface um die Grundfunktionalitäten *Reflection*, *Notifikation*, Zugriff auf das Metaobjekt (*EClass*) und auf den Container sowie die *Ressource* zu gewährleisten.

EStructuralFeature Diese abstrakte Klasse beschreibt den Charakter von Referenzen und Attributen. Die Attribute *changeable*, *transient*, *unique*, *unsettable* und *volatile* beschreiben wie Werte dieses Features behandelt werden. *lowerBound*, *upperBound*, *required* und *many* ermöglichen die Beschreibung der Multiplizität und der *defaultValue* ermöglicht den Zugriff auf den Standardwert des zugrunde liegenden Java Objekts. Falls ein solcher nicht existiert ist dieser *null*. Dieses Objekt lässt sich mit *setDefaultValue* explizit festlegen. Weiters kann die *FeatureID* abgefragt werden. Diese ist eine ganze Zahl welche für jedes Feature eindeutig innerhalb einer Klasse vom Framework automatisch vergeben wird und dient zum Zugriff über Reflection. Schließlich kann noch mit *getContainerClass* auf die Implementierungsklasse, die dieses Feature enthält, zugegriffen werden.

EAttribute Mit diesem Konzept können Attribute modelliert werden. Ein *EAttribute* hat einen Namen (über *ENamedElement*) und kann durch einen Verweis auf *EDatatype* typisiert werden. Attribute stehen in einer starken Abhängigkeit zu einer *EClass* (durch *eAttributes*), was bedeutet dass sie nur innerhalb einer solchen existieren dürfen.

EReference Eine *EReference* wird verwendet um Beziehungen zwischen Klassen modellieren zu können. Dabei ist sie immer nur in eine Richtung navigierbar. Soll eine bidirektionale Verbindung modelliert werden kann dies durch zwei Referenzen erreicht werden, wobei die beiden Referenzen als *eOpposite* die jeweils Andere enthalten. Am Ende einer Referenz darf nur eine *EClass* stehen, welche durch das Attribut *eReferenceType* angesprochen werden kann. Die Multiplizität kann durch *lower-* und *upperbounds* (von *EStructuralFeature*) festgelegt werden. Schlussendlich können starke Abhängigkeiten durch das Belegen des *Containments* modelliert werden.

EClassifier *EClass* und *EDatatype* erben von dieser Basisklasse. Sie ist das Ziel der *eType* Referenz des *ETypedElement*. Dies bedeutet, dass beispielsweise eine *EReferenz* oder ein *EAttribute* über diese Referenz sowohl mit einer *EClass* als auch mit einem *EDatatype* verbunden werden können. Über *instanceClassName* und *instanceClass* wird der Zugriff auf die Implementierungsklasse gewährt. Ähnlich wie beim *EStructuralFeature* ist auch hier ein *defaultValue* zu finden. Die Methode *isInstance()* gibt Auskunft ob das, als Parameter übergebene Objekt eine Instanz dieses *EClassifiers* ist. Dabei werden natürlich auch *EClassifier*, die von diesem abgeleitet sind mit einbezogen.

EClass Dieses Objekt wird verwendet um Klassen zu modellieren. Jede Klasse hat einen Namen (wieder von *ENamedElement*) und kann mehrere Referenzen und Attribute besitzen, welche über *eAllAttributes* und *eAllReferences* oder über *eAllStructuralFeatures* zugänglich sind. Eine *EClass* kann auf mehrere Superklassen (*eSuperTypes*) verweisen und unterstützt somit Mehrfachvererbung. Dies gilt auch für Klassen die in anderen Ressourcen gespeichert wurden. Die Attribute *interface* und *abstract* spezifizieren dasselbe Verhalten wie in Java. *isSuperTypeOf* überprüft, ob die als Parameter übergebene Klasse von dieser erbt. *getEStructuralFeature* ermöglicht den Zugriff auf Referenzen und Attribute über deren ID oder Namen.

EDatatype Im Gegensatz zu Klassen wird der *EDatatype* verwendet um einfache Datentypen darzustellen. Obwohl dieser auch eine beliebige Java-Klasse sein kann, ist es anzuraten bei komplexeren Strukturen eine *EClass* zu verwenden um von der automatischen Serialisierung und Notifikation zu profitieren. *instanceClassName* und *instanceClass* erlauben Zugriff auf diese zugrunde liegende

Java-Klasse. Der spezielle Datentyp *EEnum* kann jeden Wert (*EEnumLiteral*) aus einer vordefinierten Liste annehmen.

EPackage Wie Pakete in Java dient ein *EPackage* dazu, Elemente zu gruppieren. Um Pakete eindeutig zu identifizieren besitzen diese eine *nsURI* und einen *nsPrefix*. Alle *EClassifiers* eines Paketes sind in diesem enthalten und können durch die Referenz *eClassifiers* abgerufen werden. Pakete können hierarchisch aufgebaut werden, was mittels *eSubPackages* modelliert werden kann.

EFactory Elemente eines Paketes werden von dessen *EFactory* erzeugt. Die Referenz auf das entsprechende *EPackage* ist das einzige *EStructuralFeature*, das in einer *Factory* zu finden ist. Eine *EFactory* besitzt die Methoden *create*, *createFromString* und *convertToString*, um die entsprechenden Objekte zu erzeugen.

EAnnotation Jedes *EModelElement* und somit alle Elemente in Ecore können durch Annotationen mit zusätzlichen Informationen versehen werden.

EOperation Neben der Struktur kann auch das Verhalten von Klassen modelliert werden. Dies geschieht durch das Hinzufügen einer *EOperation*. Über die Klasse kann man die Operationen mittels *eAllOperations* ansprechen. Operationen können mehrere *EParameter* besitzen auf welche wiederum mittels *eParameters* zugegriffen wird. Das Konzept der *Exceptions* wie aus vielen Programmiersprachen bekannt, kann mit *eExceptions* modelliert werden. Das tatsächliche Verhalten kann mit den Operationen nicht abgebildet werden. Der Generator erzeugt nur die Methoden, der eigentliche Code muss in den Implementierungsklassen manuell erstellt werden.

2.2.3 Modellerstellung

Wie bereits eingangs erwähnt können Modelle in EMF auf drei unterschiedliche Arten erstellt werden. Zusätzlich zum Ecore-Modell wird dabei jeweils eine zweite Datei mit der Endung *genmodel* erzeugt. Diese fungiert als Dekoration um zusätzliche Informationen zur Codegenerierung bereitzustellen. Zur Illustration wird als Beispiel ein vereinfachtes Metamodell des ER Diagramms als Ecore-Modell erstellt. Abbildung 2.6 zeigt dieses Metamodell in UML Notation. Eine Entität kann mehrere Attribute besitzen, und Teil von Relationen sein. Jede Relation verweist auf bis zu zwei Entitäten (*sourceEntity* und *targetEntity*) und auf zwei Rollen (*ownedRoles*). Attribute können ohne Entität nicht existieren, daher ist diese Verbindung durch eine starke Abhängigkeit ausgezeichnet. Zusätzlich zu diesen Elementen wird eine Klasse *Diagram* erzeugt, die alle Klassen beinhalten soll die noch in keinem anderen Element enthalten sind. Dies ist nötig, da EMF von einer Baumstruktur ausgeht. Im Beispiel stehen die Klassen *Entity*, *Relationship* und *Role* in keiner starken Abhängigkeit zu einer anderen Klasse und werden

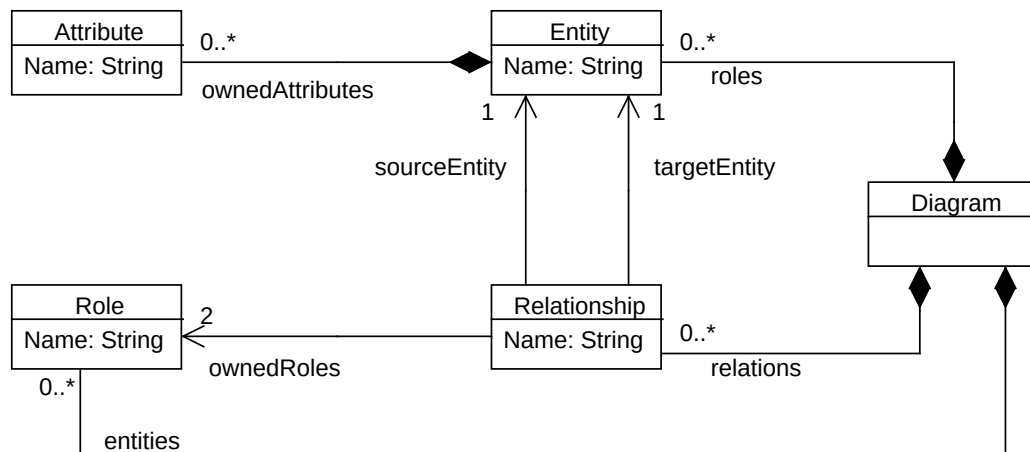


Abbildung 2.6: Vereinfachtes ER-Metamodell

somit der Diagrammklasse zugeordnet.

Der nächste Abschnitt veranschaulicht die Erstellung eines Ecore Modells durch die unterschiedlichen Methoden. Zusätzlich kann ein Modell auch dynamisch per Code erzeugt werden. Diese Methode wird vom *EcoreMerger* im Kapitel 4.1 angewandt.

1. Annotierte Java Interfaces

Anders als beim XMI- oder UML-Import kann der *Wizard* keine annotierten Java Klassen als Input für die Projekterstellung verwenden. Ein EMF-Projekt muss bereits existieren. Dies kann einfach mit dem *Empty EMF Project Wizard* angelegt werden.

Für erfahrene Java Programmierer stellt diese Art vermutlich den einfachsten Einstiegspunkt in EMF dar, da mit dieser Methode kein erheblicher Mehraufwand entsteht. Darüber hinaus wird lediglich ein Texteditor bzw. ein Javaeditor benötigt welcher ohnehin in Eclipse bereits integriert ist. Java dahingehend zu annotieren um daraus ein Ecore-Modell zu erstellen ist relativ einfach und intuitiv.

Listing 2.1: Annotiertes Interface Entity

```
1 package org.tuwien.mtbe.er;
2 import java.util.List;
3 import org.eclipse.emf.ecore.EObject;
4 /** @model */
5 public interface Entity extends EObject {
6     /** @model required="true" */
7     public String getName();
8     /** @model type="Attribute" containment="true" */
9     public List getOwnedAttributes();
10 }
```

Bei der Erstellung des Interfaces werden die einzelnen Methoden mit Annotationen versehen, um EMF anzuzeigen, dass diese Methode zu behandeln ist. Das Ergebnis kann durch Angabe zusätzlicher Informationen beeinflusst und verfeinert werden. *required* bei *getName* in Listing 2.1 zum Beispiel gibt an, dass eine gültige Instanz einer Entität einen Namen haben muss. Da eine Entität eine *one-to-many* Beziehung zu Attributen besitzt, muss der Typ dieser Referenz spezifiziert werden. Wird das Objekt *List*, wie in Java ab Version 1.5 üblich parametrisiert, kann diese Angabe entfallen. Die Auszeichnung *containment* gibt an, dass alle Attribute einer Entität in diesem enthalten sind. Dies ermöglicht die Modellierung von Kompositionen. Der Quellcode für *Diagram*, *Relationship*, *Attribute* und *Role* kann im Anhang (A.1) nachgelesen werden. Sind alle Interfaces erstellt und mit Annotationen versehen, kann über einen *Wizard* (*new >Other >EMF Model*) das Modell erzeugt werden. Dieser benötigt als Eingaben den Namen des zu generierenden Modells sowie das Paket, welches die Java Interfaces enthält. Er erzeugt sowohl das Generator- als auch das Ecore-Modell welche direkt mit dem, in EMF enthaltenen Editor geöffnet und bearbeitet werden können.

2. XML Schema

Ecore-Modelle können auch aus einem XML Schema erstellt werden. Hierbei wird *EPackage* zu einem Schema, *ComplexTypes* werden zu Klassen, *SimpleTypes* zu *DataTypes*. *DataTypes*, die in anderen Elementen enthalten sind werden zu Attributen und enthaltene *ComplexTypes* zu Referenzen. Da die Standardserialisierung von Ecore nicht zwangsweise zum ursprünglichen Schema konform ist, wird für alle erzeugten Ecore-Elemente eine entsprechende Annotation (siehe Metamodell 2.5) erzeugt, die die nötigen Informationen für die Serialisierung beinhalten.

Folgendes Listing 2.2 zeigt einen Ausschnitt aus dem XML Schema, welches zum Erstellen des ER-Metamodells verwendet werden kann. Für das vollständige XML Schema siehe Anhang A.6.

Listing 2.2: ER Metamodell als XML Schema

```

1      ...
2      <xsd:complexType name="Entity">
3          <xsd:sequence>
4              <xsd:element maxOccurs="1" minOccurs="1" name="name" type="
                    xsd:string"/>
5              <xsd:element maxOccurs="unbounded" minOccurs="0" name="
                    ownedAttributes" type="Attribute"/>
6          </xsd:sequence>
7      </xsd:complexType>
8      ...
9      <xsd:complexType name="Relationship">
10         <xsd:sequence>
11             <xsd:element name="name" type="xsd:string"/>
12             <xsd:element maxOccurs="1" minOccurs="1" name="
                    sourceEntity" type="Entity"/>
13             <xsd:element maxOccurs="1" minOccurs="1" name="
                    targetEntity" type="Entity"/>
14             <xsd:element maxOccurs="2" minOccurs="2" name="
                    ownedRoles" type="Role"/>
15         </xsd:sequence>
16     </xsd:complexType>
17     <xsd:complexType name="Role">
18         <xsd:sequence>
19             <xsd:element name="name" type="xsd:string"/>
20             <xsd:element maxOccurs="1" minOccurs="0" name="type"
                    type="Entity"/>
21         </xsd:sequence>
22     </xsd:complexType>
23     ...

```

Alle Vorkommen von *minOccurs* und *maxOccurs* werden im Ecore-Modell zu *lowerBound* und *upperBound*. Für das Attribut *name* vom Typ *attribute* wird sowohl *upperBound* als auch *lowerBound* mit 1 belegt, da im Schema das Attribut zwingend erforderlich ist (*required*). An dieser Stelle wird ersichtlich dass der Entwickler nicht über vergleichbare Steuerungsmöglichkeiten wie bei der Methode mit annotierten Java Interfaces verfügt. Durch die Transformationsregeln wird automatisch das *containment* gesetzt. Und zwar geschieht dies für alle komplexen Typen welche in anderen komplexen Typen enthalten sind. Im Beispiel sind *Entity* und *Relationship* und *Role* im *Diagram* enthalten. Zusätzlich aber enthält der Typ *Entity* auch *Relationship* Elemente und der Typ *Relationship* enthält Elemente vom Typ *Entity*. Um zu gewährleisten, dass *Entity*, *Relationship* und *Role* nur im *Diagram* enthalten sind, muss das generierte Ecore-Modell zusätzlich angepasst werden. Dazu wird bei der Rolle in den Eigenschaften der Referenz das Feature *containment* auf *false* gesetzt. Dasselbe muss für die Referenzen in der Relationsklasse vorgenommen werden. Etwas aufwändiger ist die direkte Anpassung des Schemas um diese *containments* gleich richtig transformiert zu bekommen. Dazu müssen die Referenzen als Attribute erstellt werden,

deren Wert ein *XPath* Ausdruck ist welcher auf das richtige Element verweist. Da das Generatormodell auf das Ecore-Modell verweist, muss dies praktischerweise nicht geändert werden und ist automatisch auf dem aktuellsten Stand.

3. Direct Ecore Editing

UML ist in den letzten Jahren zur Standardsprache in der Modellierung geworden. Fast alle Modellierungswerkzeuge unterstützen UML und die meisten Softwareentwickler haben in irgendeiner Form mit UML zu tun. Da zum Erstellen eines Ecore-Modells nur ein kleiner Teilbereich des UML-Klassendiagramms benötigt wird ist dies auch für Anfänger leicht zu meistern. Jene die bereits Klassendiagramme erstellt haben werden sich mit dieser Methode rasch anfreunden können, da diese Diagramme im Laufe des Entwicklungsprozesses meist ohnehin erstellt werden. Einige Ecore-Attribute lassen sich in UML nicht so einfach darstellen, wie diese modelliert werden hängt vom eingesetzten UML-Editor ab. Ist es mit diesem nicht möglich kann immer noch das erzeugte Ecore-Modell angepasst werden.

Jedes UML-Paket wird auch zu einem Paket in EMF transformiert. Dabei wird selbstverständlich auch die Pakethierarchie übernommen. Falls die *nsURI* in UML nicht modelliert wurde, wird sie durch die Transformation automatisch erstellt. Alle UML Klassen eines Paketes werden zu *Classifiers*, wobei der genaue Typ vom Stereotyp der Klasse abhängt. Fehlt dieser wird automatisch eine *EClass* erzeugt. UML-Klassen die mit `<<enumeration>>` versehen sind werden zum Typ *EEnum* und Klassen mit `<<datatype>>` werden zu einem *EDataType*. Dabei ist zu berücksichtigen, dass in UML meist nicht der primitive Typ der Implementierung angegeben wird, was aber für Ecore-Modelle sehr wohl anzugeben ist.

Generierung

Alle oben genannten Methoden kommen zum selben Ergebnis. Sie erzeugen ein Ecore-Metamodell sowie ein Generatormodell. Daraus kann nun ganz einfach Java-Code erzeugt werden. Dazu muss das Generatormodell geöffnet werden (*.gen-model*) und im Kontextmenü eines beliebigen Elements im Baum *Generate Model Code* ausgewählt werden. Zusätzlich kann ein Editor mittels *Generate Edit Code* und *Generate Editor Code* generiert werden. Falls das Ecore-Modell nicht aus annotierten Interfaces erzeugt wurde, erstellt der Generator diese während des Vorgangs. Zusätzlich werden die Implementierungsklassen der Interfaces und eine *Factory* erstellt, welche *create* Methoden für alle Objekte dieses Metamodells enthält. Instanzen von den Metaklassen sollten über diese *Factory* erzeugt werden, können aber auch durch den *new* Operator instanziiert werden. Listing 2.3

zeigt einen Ausschnitt aus der Implementierung der Klasse *EntityImpl*.

Listing 2.3: Ausschnitt aus EntityImpl.java

```
1 public class EntityImpl extends EObjectImpl implements Entity {
2     /** @generated */
3     public String getName() {
4         return name;
5     }
6     /** @generated */
7     public void setName(String newName) {
8         String oldName = name;
9         name = newName;
10        if (eNotificationRequired())
11            eNotify(new ENotificationImpl(this, Notification.SET,
12                ErPackage.ENTITY__NAME, oldName, name));
13    }
```

Der vom Generator erzeugte Code bietet Zugriff auf die Attribute und Referenzen einer Klasse über *Get* und *Set* Methoden. Abhängig von den im Modell gesetzten Werten für ein *EStructuredFeature*, wie zum Beispiel *upperBound* oder *lowerBound* variiert die Implementierung. Im ER-Beispiel wird für primitive Datentypen sowohl eine *get* als auch eine *set* Methode generiert, wohingegen für Referenzen nur eine *get* Methode erstellt wird die ein *EList* Objekt liefert. Diese ist eine Subklasse von *java.util.List*. Soll zum Beispiel eine neue Relation hinzugefügt werden, wird diese einfach an die vorhandene *EList* angehängt (2.4).

Listing 2.4: Setzen von Referenzen

```
1 Entity entity = ErFactory.eINSTANCE.createEntity();
2 Relationship relationship = ErFactory.eINSTANCE.createRelationship();
3
4 entity.getRelationships().add(relationship);
```

Bidirektionale Verbindungen sind etwas komplexer, da es zusätzlich erforderlich ist, das andere Ende der Referenz zu setzen. Es sei hier nur erwähnt, dass der EMF Generator dies selbstständig erledigen kann. Da im EMF nicht möglich ist Verhalten direkt zu modellieren werden alle *EOperations* als leere Methoden angelegt. Für weitere Informationen über die genaue Funktionsweise der Generierung ist das Studium von [4] und des EMF- Quellcodes zu empfehlen.

Wie bereits erwähnt bietet EMF die Möglichkeit andere Objekte über Änderungen am Modell zu informieren (siehe *Observer Design Pattern* [9]). Zu sehen ist dies bei der erstellten Implementierung der *Set* Methoden. Aus Effizienzgründen wird der relativ aufwendige Aufruf von *eNotify* nur dann ausgeführt wenn tatsächlich Beobachter vorhanden sind. Diese werden über die Änderung, sowie den bisherigen und den neuen Wert informiert. EMF bietet die Möglichkeit

ganze Bäume von Objekten zu beobachten. Dazu existiert die Klasse *EContentAdapter* die für diesen Zweck verwendet werden kann.

Die generierten Java-Klassen können nach Belieben erweitert und verändert werden. Wird danach das Modell geändert und der Quellcode neu erstellt, gehen die manuellen Änderungen nicht verloren. Alle Methoden die mit *@generated* ausgezeichnet sind, werden dabei überschrieben, während eigene Methoden (ohne *@generated*) unverändert in den neuen Code übernommen werden. Falls bereits eine Methode mit demselben Namen existiert wird die vom Generator produzierte verworfen. Werden Interfaces hinzugefügt, sollte das Modell, wie in Abschnitt 2.2.3 beschrieben aktualisiert werden. Dieser Prozess kann allerdings abgekürzt werden indem im Kontextmenü des Generatormodells *reload*, und im darauf erscheinenden *Wizard Annotated Java Interfaces* ausgewählt wird.

Modelle mit dem Editor instanziiieren

Wie bereits erwähnt liefert EMF zusätzlich die Möglichkeit einen Editor für eigene Modelle zu generieren. Dieser wird erstellt indem zusätzlich mittels *Generate Edit Code* und *Generate Editor Code* zwei Eclipse-Plug-in Projekte angelegt werden. EMF trennt dabei Code der für die Darstellung verwendet wird und somit von Eclipse abhängig ist und unabhängigen Modellcode. Der Editor lässt sich entweder in einer neuen Eclipse Instanz, oder nach dem Export und Neustart in Eclipse selber starten. Dazu muss im Kontextmenü des Plug-ins *Run as* und *Runtime Workbench* ausgewählt werden. In der *Runtime Workbench* steht unter *Example EMF Model Creation Wizards* ein neuer *Wizard* zur Verfügung. Als Wurzelement muss in diesem das Diagrammelement angegeben werden. Im Editor können nun an diesen Knoten beliebig viele Entitäten, Relationen und Rollen angehängt werden. Attribute können aber nur als Kindelemente der Entitäten vorkommen. Der Grund hierfür sind *Containment*-Attribute der Referenzen. Der Editor liefert *Drag- and Drop* Funktionalität, das Erstellen und Löschen aller Elemente, sowie einen *Properties View* in welchem die Eigenschaften der Elemente festgelegt werden können.

2.2.4 Programmieren im EMF

EMF bietet die Möglichkeit Modelle dynamisch zur Laufzeit zu erzeugen. Da das Ecore-Metamodell sein eigenes Metamodell ist, kann bei der Erstellung eines Modells der Modellteil des EMF Ecore-Projektes auf die gleiche Weise verwendet werden, als würde eine Instanz eines selbst erstellten Metamodells erzeugt. Dieser

Abschnitt soll zeigen wie das ER Metamodell dynamisch erzeugt werden kann. Anschließend wird erläutert auf welche Weise Instanzen dieses Metamodells erstellt werden können. Schlussendlich wird noch vorgeführt, wie dasselbe Ergebnis ohne Codegenerierung nur mit Hilfe von Reflection erreicht werden kann. Der vollständige Quellcode findet sich im Anhang A.1.

Metamodelle dynamisch erzeugen

Da das ER-Metamodell als Instanz des Ecore-Metamodells erzeugt werden soll, kann dazu die *EcoreFactory* benutzt werden. Diese hat, wie der aus eigenen Modellen erzeugte Quellcode, Methoden zum erzeugen von Objekten des Metamodells. Im Ecore Metamodell sind dies zum Beispiel *EClass* oder *EAttribute*. Demnach unterscheidet sich dies nicht sonderlich von der dynamischen Instanziierung von eigenen Metamodellen und stellt, sofern man mit dem Ecore-Metamodell einigermaßen vertraut ist auch keine besondere Herausforderung dar.

Listing 2.5: Erzeugen einer Resource

```
1 Resource.Factory.Registry.INSTANCE. getExtensionToFactoryMap().put("XMI",
    new XMIResourceFactoryImpl());
2
3 ResourceSet rs = new ResourceSetImpl();
4 URI erModelURI = URI.createFileURI(args[0]);
5
6 Resource outputResource = rs.createResource(erModelURI);
7
8 EcoreFactory emf = EcoreFactory.eINSTANCE;
9
10 EcorePackage emfPackage = EcorePackage.eINSTANCE;
```

Zeile 1 aus Listing 2.5 registriert die XMI Serialisierung als Standard ("XMI"). Falls Eclipse als Laufzeitumgebung verwendet wird kann dies entfallen, da dieser Schritt schon von den EMF Plug-ins erledigt wurde. Es besteht die Möglichkeit eine eigene Serialisierung zu registrieren, beispielsweise steht auch eine für XML bereit. Ein *ResourceSet* ist eine Sammlung von gemeinsam geladenen oder erzeugten Ressourcen und ist für die Verwaltung und Persistenz dieser zuständig. In Zeile 2 wird ein neues *ResourceSet* erzeugt und in Zeile 4 wird von diesem die neue Ressource *outputResource* angelegt. Dabei wird die *Registry* verwendet um für die gewünschte Dateierweiterung die entsprechende Serialisierung zu laden. Da vorher als Standard XMI registriert worden ist, wird eine neue XMI Resource angelegt. Das *ResourceSet* registriert anhand einer URI welche Ressourcen bereits geladen wurden und verhindert somit dass eine Ressource doppelt im Speicher vorhanden ist. Sind Proxies vorhanden lädt das *ResourceSet* diese beim Zugriff automatisch, sofern die zugehörige Ressource aufgelöst werden kann. Die Zeilen 5

und 6 setzen schlussendlich noch die Variablen *emf* und *emfPackage* auf die entsprechenden statischen Implementierungsklassen. Die *EcoreFactory* ermöglicht im Folgenden die Erstellung von Elementen des Ecore-Metamodells und das *EcorePackage* bietet einfachen Zugriff auf die Klassen der primitiven EMF Typen wie zum Beispiel dem *EString*.

Listing 2.6: Pakete erstellen

```
1 EPackage root = emf.createEPackage();
2
3 root.setName("er");
4 root.setNsURI("http://org.tuwien.tbte.er");
5 root.setNsPrefix("er");
6
7 outputResource.getContents().add(root);
```

Zeile 1 aus Listing 2.6 erstellt mittels der *EcoreFactory* das Paket *root* welches alle Klassen des ER- Metamodells enthalten wird. In Zeile 3 wird der *namespace* gesetzt. Dieser wird von der *Registry* verwendet um Pakete eindeutig zu identifizieren. In Zeile 4 wird das Paket zur Ressource hinzugefügt. Da nachfolgend alle Elemente an das Paket *root* angehängt werden ist somit der ganze Baum in der Ressource enthalten.

Listing 2.7: Klassen erzeugen

```
1 EClass entity = emf.createEClass();
2 entity.setName("Entity");
3
4 EAttribute nameAttribute = emf.createEAttribute();
5 nameAttribute.setName("Name");
6 nameAttribute.setEType(emfPackage.getEString());
7 nameAttribute.setLowerBound(1);
8 nameAttribute.setUpperBound(1);
9
10 entity.getEStructuralFeatures().add(nameAttribute);
11
12 root.getEClassifiers().add(entity);
```

Listing 2.7 zeigt wie eine *EClass* dynamisch erstellt wird. Hierbei handelt es sich um die *Entity* Klasse für die in den Zeilen 3 bis 7 das Attribut *name* realisiert wird. Der Name ist vom Typ *String* und es muss genau einmal vorkommen. Das fertige Attribut wird, da es vom Typ *EStructuralFeature* ist (neben der *EReference*) an die Featureliste der Entität angehängt. Die Methode *getEAttributes* könnte hier auch verwendet werden, würde aber beim Kompilieren zu dem Hinweis führen, dass *getEStructuralFeatures* benutzt werden sollte. In Zeile 9 wird die Liste der *EClassifier* des Pakets um die *Entity* Klasse erweitert. Noch sind in dieser Klasse keine Referenzen auf Attribute und Relationen enthalten. Das kann erst geschehen wenn die entsprechenden Klassen selbst erzeugt wurden. Der Quellcode für diese

Operationen kann aus Anhang A.1 entnommen werden, unterscheidet sich aber nicht wesentlich von der Erzeugung einer Entität.

Listing 2.8: Referenzen erstellen

```
1 EReference entity2Attribute = emf.createEReference();
2 entity2Attribute.setName("ownedAttributes");
3 entity2Attribute.setLowerBound(0);
4 entity2Attribute.setUpperBound(-1);
5 entity2Attribute.setEType(attribute);
6 entity2Attribute.setContainment(true);
7 entity.getEStructuralFeatures().add(entity2Attribute);
```

In Listing 2.8 wird eine Referenz für die Entität kreiert. Es handelt sich um eine Referenz auf eine Klasse. Attribute können demnach in unbegrenzter Anzahl vorkommen. Gültig ist aber auch eine leere Liste. Der Einfachheit halber ist diese Referenz nicht bidirektional, dazu müsste zusätzlich *eOpposite* belegt werden. Schlussendlich wird das *containment* auf *true* gesetzt. Der Grund ist die Komposition zwischen Attributen und Entitäten. Attribute können ohne die zugehörige Entität nicht gespeichert werden. Sie werden vom Framework auch automatisch gelöscht wenn ihre Entität entfernt wird.

Listing 2.9: Ressourcen speichern

```
1 try {
2
3     outputResource.save(Collections.EMPTY_MAP);
4
5 } catch (IOException e) {
6     e.printStackTrace();
7 }
```

Wenn alle Elemente erstellt und entsprechend im Baum eingehängt wurden kann die Ressource serialisiert werden. Die Ressource stellt im EMF einen Container für Persistenz zur Verfügung. Zu beachten ist, dass *EObjects*, die an den Inhalt einer Ressource angehängt wurden, von ihrer ursprünglichen Resource entfernt werden. Im ER-Beispiel kommt das nicht zum tragen, da die Klassen erzeugt wurden und somit in keiner anderen Ressource enthalten waren. Diese Tatsache wird erst später beim Kopieren von Elementen eine Rolle spielen. Da in Listing 2.5 die XMI Serialisierung als Standard definiert worden ist, wird das ER Metamodell in XMI serialisiert. Eigene Implementierungen können aber auch in beliebige Ausgabeformate, beispielsweise in eine relationale Datenbank schreiben. Der *save* Operation können über eine *Map* Einstellungen zur Serialisierung übergeben werden. Eine Beschreibung wie die Serialisierung an eigene Bedürfnisse angepasst werden kann findet sich in [22].

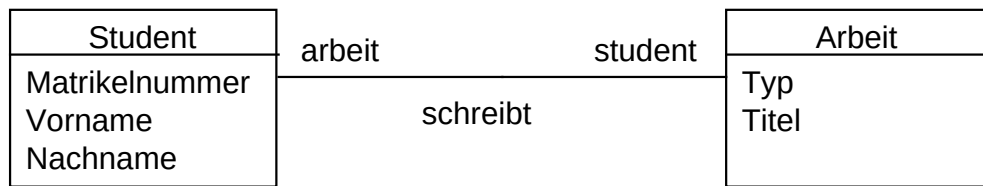


Abbildung 2.7: Beispiel - Student schreibt Arbeit

Metamodelle instanziiieren

Natürlich können die erstellten Metamodelle auch instanziiert und serialisiert werden. Das funktioniert auf dieselbe Art wie das Metamodell erstellt wurde. Am einfachsten ist es aber den generierten Editor zu verwenden wie schon in Abschnitt 2.2.3 beschrieben wurde. Programmatisch ist die übersichtlichste Weise die vom Generator erzeugten Modellklassen zu verwenden. Diese sind üblicherweise im ursprünglichen Projekt enthalten während für die Edit- und die Editorklassen jeweils ein neues Projekt angelegt wird.

Zur Veranschaulichung soll das Beispiel 2.7 *Student schreibt Arbeit* anhand des zuvor erstellten ER Metamodells erzeugt werden. Da das einfache Metamodell keine Multiplizität unterstützt, sind diese im Beispiel weggelassen.

Listing 2.10: Student und Diagram erzeugen

```

1 ErFactory factory = ErFactory.eINSTANCE;
2 Diagram diagram = factory.createDiagram();
3
4 /* Erzeugen von Student */
5 Entity student = factory.createEntity();
6 student.setName("Student");
7
8 Attribute attribute = factory.createAttribute();
9 attribute.setName("Matrikelnummer");
10
11 student.getOwnedAttributes().add(attribute);
12
13 diagram.getEntities().add(student);

```

Da der EMF Generator den Code zur Manipulation des Modells erzeugt hat, gibt es entsprechend auch eine *Factory* für das ER Paket. Diese enthält *create* Methoden für alle Elemente im Paket. Listing 2.10 beginnt mit der Erstellung des *Diagram* Elements. Dieses dient als Wurzelknoten und kann Entitäten, Relationen und Rollen beinhalten. Danach wird die Entität “Student” erzeugt. Die Studentklasse erhält ein Attribut und wird der Entitätsliste der Diagrammklasse hinzugefügt.

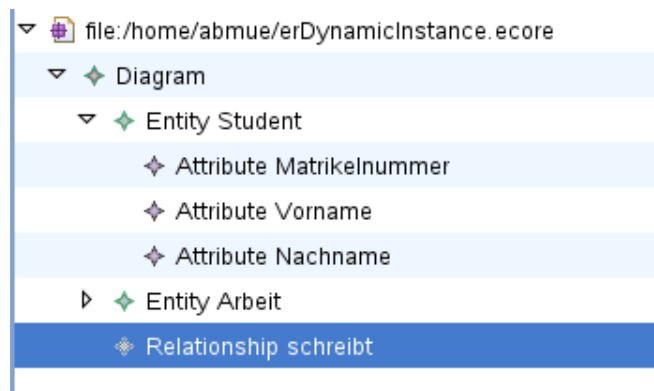


Abbildung 2.8: ER Modell im EMF Editor

Die Erstellung der Klasse *Arbeit* erfolgt auf die gleiche Weise.

Listing 2.11: Eine Relation erzeugen

```

1 Relationship schreibt = factory.createRelationship();
2 schreibt.setName("schreibt");
3
4 diagram.getRelationships().add(schreibt);
5
6 schreibt.setSourceEntity(student);
7 schreibt.setTargetEntity(arbeit);

```

Im Unterschied zu den Attributen bei den Entitäten sind das *sourceEntity* und das *targetEntity* nicht in der Relation enthalten, sodass das Entfernen einer Relation aus dem Modell die beteiligten Entitäten nicht beeinflusst. Ähnlich werden auch die Rollen für die Relation erzeugt. Wird das Modell serialisiert, kann es im EMF-Editor geöffnet werden.

Reflection

Praktischerweise können Modelle auch erstellt werden, ohne dass zuvor Code generiert wurde. Dies macht vor allem dann Sinn, wenn generisch programmiert werden soll um möglichst viele Modelle zu unterstützen. EMF bietet dazu die *Reflection API*. Die wichtigsten Methoden sind *eGet*, *eSet*, *eIsSet* und *eUnset* welche durch das *EObject* spezifiziert sind. Wurde der Modellcode bereits generiert, können Attribute und Referenzen anhand ihrer *FeatureId* angesprochen werden, die für jedes *EStructuralFeature* innerhalb einer Klasse eindeutig ist. Diese wird erst bei der Codegenerierung vergeben, das jeweilige Feature kann aber auch anhand seines Namens identifiziert werden. Ist auch dieser nicht bekannt muss über alle Attribute und Referenzen iteriert werden um das gewünschte Feature anhand anderer Eigenschaften zu finden. Der vollständige Quellcode des Beispiels ist wieder im Anhang unter A.1 nachzulesen.

Listing 2.12: Ein Modell laden

```

1 URI modelUri = URI.createFileURI(args[0]);
2
3 Resource inputResource = rs.getResource(modelUri, true);
4
5 EPackage metaPackage = (EPackage)inputResource.getContents().get(0);
6
7 EClass diagramMeta = (EClass)metaPackage.getEClassifier("Diagram");
8 EClass entityMeta = (EClass)metaPackage.getEClassifier("Entity");
9 EClass attributeMeta = (EClass)metaPackage.getEClassifier("Attribute");
10 EClass relationshipMeta = (EClass)metaPackage.getEClassifier("
    Relationship");

```

In Listing 2.12 wird eine Ressource durch die Angabe ihrer URI geladen. Da das ER-Metamodell nur aus einem Paket besteht das alle Elemente enthält, kann in Zeile 3 davon ausgegangen werden, dass das Wurzelement ein *EPackage* ist. Das Paket bietet die Methode *getEClassifier(String)*, mittels derer die Metaklassen des ER Diagramms geladen werden können. Falls generischer Code erstellt wird, sind die Namen der Elemente natürlich nicht bekannt. Es ist dann aber ohne Benutzerinteraktion nicht möglich herauszufinden welche Klasse beispielsweise die Entität darstellt, was aber nicht unbedingt notwendig ist um mit Modellen zu arbeiten, wie später in Kapitel 4.1 gezeigt werden wird.

Listing 2.13: Eine Entität per Reflection erzeugen

```

1 EObject entityInstStudent =
2 metaPackage.getEFactoryInstance().create(entityMeta);
3 entityInstStudent.eSet
4     (entityMeta.getEStructuralFeature("name"), "Student");
5
6 EObject attributeInst =
7 metaPackage.getEFactoryInstance().create(attributeMeta);
8 attributeInst.eSet
9     (attributeMeta.getEStructuralFeature("name"), "Matrikelnummer");
10
11 ((EList)entityInstStudent.eGet
12     (entityMeta.getEStructuralFeature("ownedAttributes"))).add(
13     attributeInst);
14 ((EList)diagramInst.eGet(
15     diagramMeta.getEStructuralFeature("entities"))).add(
16     entityInstStudent);

```

Mit der *Factory* des Pakets können alle Elemente, die in diesem Paket enthalten sind, erzeugt werden. Da kein Modellcode generiert wurde, kann nicht wie im letzten Abschnitt einfach *create<Elementname>* aufgerufen werden. Stattdessen wird die generische Implementierung verwendet die als Parameter die entsprechende Klasse erwartet. Wegen der fehlenden Javaimplementierung kann das erzeugte Objekt auch nicht per *Cast* in ein *Entity* Objekt transformiert werden. Darum muss auch beim Setzen von Attributen und Referenzen die Reflection verwendet werden. In Zeile 2 des Listings 2.13 wird der Name der Entität mit *Student* belegt.

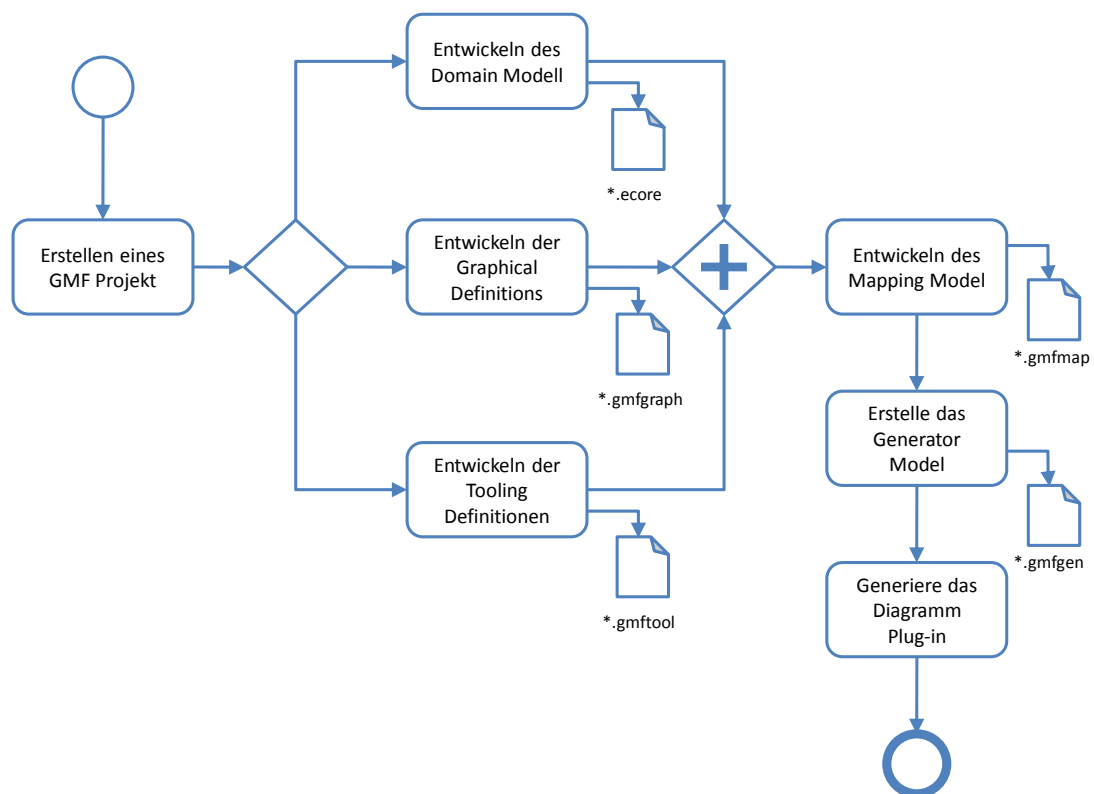


Abbildung 2.9: Übersicht der GMF Komponenten [34]

Um das Feature zu identifizieren wird dieses von der Metaklasse extrahiert und der *eSet* Methode als Parameter übergeben. In den Zeilen 3 bis 5 wird das Attribut *Matrikelnummer* erzeugt und an die Attributliste der Entität angehängt. Da Entitäten im *Diagram* Element enthalten sind wird in Zeile 6 dieses an die Liste der Entitäten angefügt. Sind alle gewünschten Elemente erstellt, kann das Modell über eine Ressource serialisiert werden. Das resultierende Ecore Modell entspricht genau dem aus dem vorigen Abschnitt.

2.3 GMF - Grafical Modeling Framework

2.3.1 Einführung

Das *Graphical Modeling Framework* [44] stellt eine generische Verbindung zwischen dem *Eclipse Modeling Framework* (EMF) und dem *Gaphical Editing Framework* (GEF) [46] dar. GMF ermöglicht das Erstellen graphischer Editoren auf Basis von Metamodellen in EMF. GEF stellt dabei die graphischen Fähigkeiten zur Verfügung. Dazu greift es auf *Draw2D* zurück. *Draw2D* ist wiederum Bestandteil vom *Standard Widget Toolkit* (SWT), auf welchem Eclipse aufbaut.

Einen graphischen Editor in GEF zu entwickeln setzt Kenntnisse in Draw2D und SWT voraus, während in GMF diese Dinge vom Generator erledigt werden können. Die Definition eines GMF Projektes geschieht deklarativ mittels unterschiedlichen Modellen, welche alle im Ecore Format abgelegt werden. In welcher Reihenfolge diese erstellt werden und wie sie zusammenhängen ist aus Abbildung 2.9 ersichtlich. GMF ist als Eclipse Plug-in implementiert mit welchem relativ einfach ein neues GMF Projekt erstellt werden kann. Die Informationen in diesem Kapitel stammen aus dem GMF-Tutorial [34] sowie aus unseren Erfahrungen im Umgang mit GMF während dieses Projektes. Zusätzlich wird versucht GMF anhand eines praktischen Beispiels zu erläutern, für einen detaillierten Einblick ist darüber hinaus zu empfehlen den Quellcode des MTBE Projektes zu studieren.

Nach der Installation des GMF Plug-ins können neue GMF Projekte angelegt werden. Der Wizard erzeugt dabei ein Plug-in Projekt mit den nötigen Bibliotheken und erstellt einen Ordner *model* in welchem die benötigten Modelle erzeugt werden um den graphischen Editor zu generieren. Ob zuerst das *Domain Model* oder die *Graphical Definition* erzeugt werden spielt dabei keine Rolle.

Zum Verständnis wird empfohlen das Tutorial auf der offiziellen GMF Projekthomepage durchzuspielen, sowie Abschnitt 4.2.3 der Diplomarbeit zu lesen. Mittels GMF können sehr schnell einfache Editoren erstellt werden. Das aber auch umfangreiche und komplexe Editoren möglich sind zeigt das UML2 Projekt [47]. Dieses hat zum Ziel einen möglichst vollständigen UML2 Editor als Eclipse Plug-in bereitzustellen. Unter dem Titel UML2 Tools existiert auch eine Implementierung auf Basis von GMF.

2.3.2 Beispiel

Um die nachfolgenden Erläuterung zu veranschaulichen wird als Beispiel ein sehr vereinfachter Editor für Klassendiagramme erstellt.

Abbildung 2.10 zeigt das Metamodell dieses Editors. Da alle Elemente das Attribut *name* besitzen, wird dieses von einer gemeinsamen Basisklasse *Named-Element* vererbt. Jede Klasse kann mehrere Attribute besitzen welche vom Typ *Property* sein müssen. Eine Assoziation verweist auf zwei *Properties*, welche als Rollen dienen können. Dieses Metamodell weicht natürlich vom UML Metamodell ab, es ist aber vor allem durch seine Einfachheit gut geeignet später für die Beschreibung der Implementierung zu dienen. Zu Beachten ist in diesem Metamodell vor allem, dass die *Properties* unabhängig von ihrer Funktion zu der jeweiligen

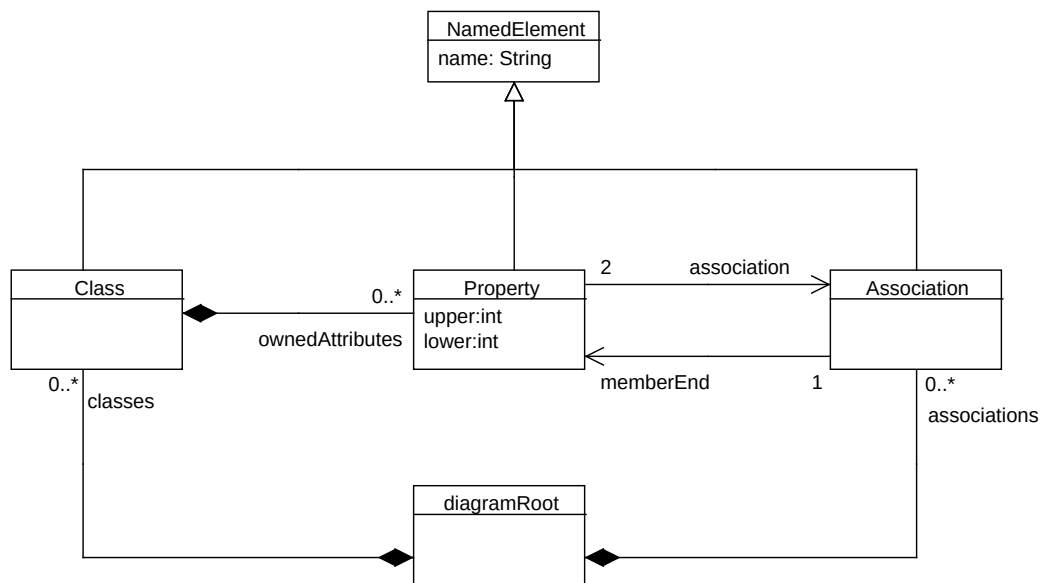


Abbildung 2.10: Metamodell eines einfachen UML-Editors

Klasse gehören, was einen interessanten Aspekt bei einer späteren Analyse darstellt.

2.3.3 Domain Model

GMF ermöglicht die Erstellung eines graphischen Editors für Modelle. Das Metamodell dieses Editors muss im Ecore-Format vorliegen und kann auf mehrere verschiedene Arten - wie in Kapitel 2.2.1 beschrieben - erstellt werden. Das Ergebnis ist eine Datei mit der Endung *.ecore* und eine mit der Endung *.genmodel*, wobei Letztere aber auch aus dem Domänenmodell erzeugt werden kann. Dies geschieht mit Hilfe des *EMF Model Wizard*. Für das Beispiel können somit die beiden Dateien aus dem EMF Projekt verwendet werden, es ist aber ratsam lediglich die *.ecore* Datei zu kopieren und das *.genmodel* neu zu generieren um die Pfade für die einzelnen Plug-ins nicht händisch ändern zu müssen.

Abbildung 2.11 stellt das zuvor beschriebene Metamodell (Abbildung 2.10) in EMF dar. Zusätzlich zu den im Metamodell dargestellten Klassen existiert noch die Klasse *DiagramRoot*. Diese enthält alle Klassen- und Assoziationsobjekte und dient als Wurzelement. Die *Properties* werden als Kindelemente (*ownedAttributes*) in der Klasse abgelegt.

Um später einen lauffähigen graphischen Editor zu erhalten muss der *Model Code* und *Edit Code* generiert werden, da das automatisch erstellte GMF Plug-in die Implementierung verwendet um die Domain Elemente zu manipulieren.

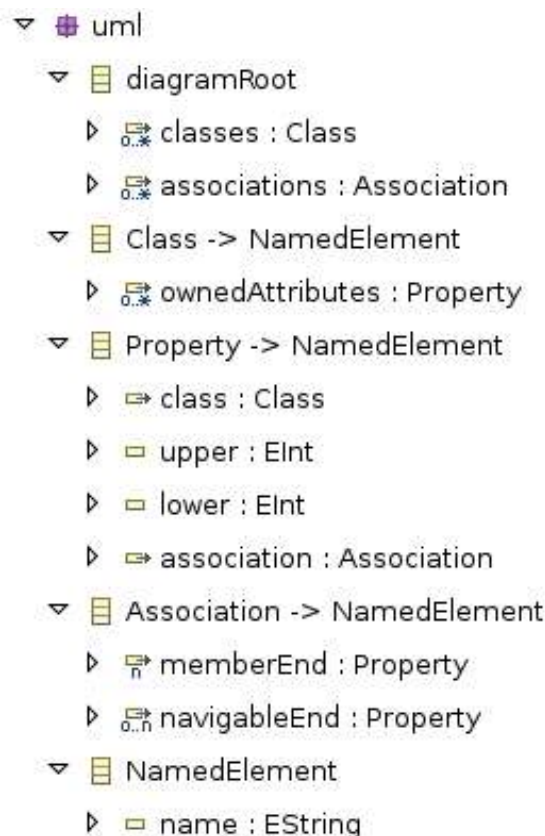


Abbildung 2.11: EMF - Metamodell eines einfachen UML-Editors

2.3.4 Graphical Definition Model

Dieses Modell beinhaltet, wie der Name schon sagt alle graphischen Elemente die für den graphischen Editor verwendet werden. Für das Beispiel werden lediglich Rechtecke, Linien und Beschriftungen benötigt, es ist aber möglich beliebige andere Elemente zu erstellen. GMF stellt bereits einen Wizard *Simple Graphical Definition Model* zur Verfügung, mit dem man aus einem Domänenmodell eine entsprechende Datei erzeugen kann, die bereits für alle Bestandteile eines Modells graphische Elemente generiert. Diese müssen nach Bedarf dann entsprechend angepasst werden. Das graphische Modell liegt selbst im Ecore-Format vor und muss somit ein Wurzelement besitzen welches *Canvas* genannt wird.

Das GMF Tutorial des Projektes [34] zeigt das vollständige Metamodell der *Graphical Definition* welches bei komplexeren Fällen konsultiert werden sollte. Da GMF klar zwischen Modell- und graphischen Elementen trennt ist es möglich graphische Definitionen für unterschiedliche Domänen wiederzuverwenden.

Das GMF Projekt stellt einige oft benötigte Definitionen zur Wiederverwendung bereit. Diese können mittels *Load Ressource* unter dem Pfad *plat-*

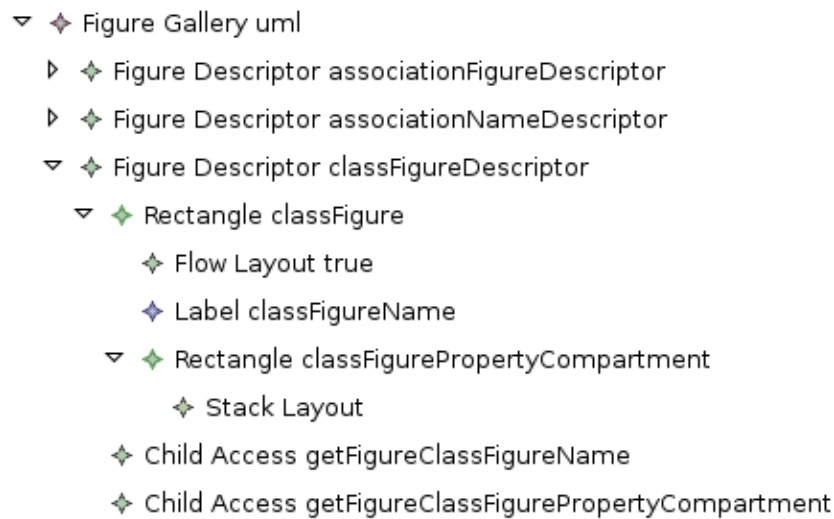


Abbildung 2.12: Ausschnitt aus der Graphical Definition

`form:/plugin/org.eclipse.gmf.graphdef/models/` eingebunden werden. Wie in Abbildung 2.12 dargestellt wird für die Klasse des UML Beispiels ein *Figure Descriptor* erstellt welcher ein Rechteck enthält. Dieses stellt den Außenrahmen der Klasse bereit. Das Rechteck enthält wiederum ein *Label* das den Klassennamen darstellen wird, sowie ein weiteres Rechteck. Dieses dient als Container für die eventuell enthaltenen Attribute der Klasse. Um darauf zugreifen zu können benötigt der *Figure Descriptor* ein *Child Access* Element. Ein solches muss folgerichtig auch für die Beschriftung erstellt werden. *Figure Descriptor* Elemente sind in einer *Figure Gallery* gruppiert. Um die graphischen Elemente aber mit Domänelementen verbinden zu können müssen dem *Canvas* Element noch jeweilige *Nodes* für Elemente wie zum Beispiel für die Klasse, *Connections* für Verbindungen, *Compartments* und *Labels* angehängt werden. Die vollständige *Graphical Definition* für diesen Editor ist im Anhang unter A.1 zu finden.

2.3.5 Tooling Definition Model

In der *Tooling Definition* wird bestimmt welche Elemente über die Palette erstellt werden können. Auch hierfür stellt GMF einen Wizard bereit der aus dem Modell einen Vorschlag generiert welcher dann nur mehr an die eigenen Bedürfnisse angepasst werden muss. Da die vom Wizard erstellte Datei in den meisten Fällen ausreichend ist, wird hier auf eine detaillierte Erklärung verzichtet.

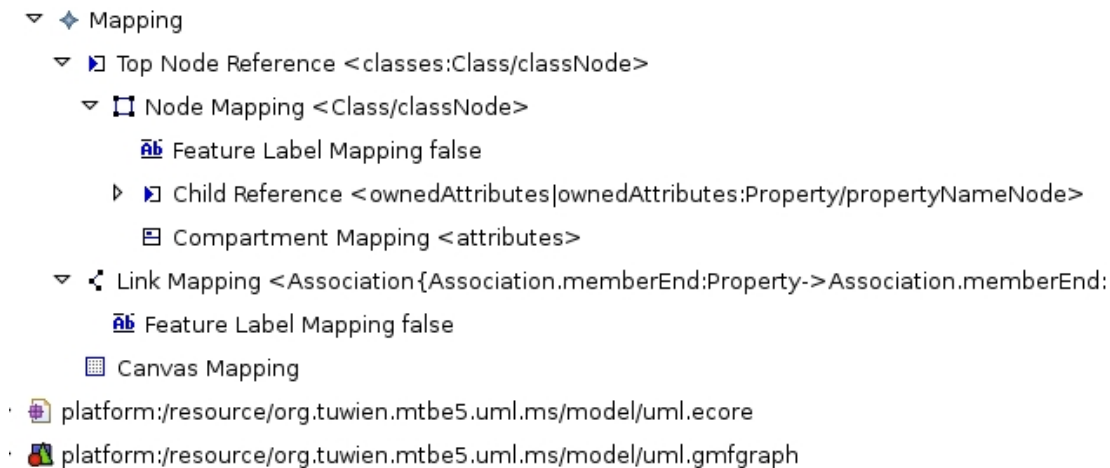


Abbildung 2.13: Mapping Definition

2.3.6 Mapping Definition Model

Wie bereits bekannt werden oftmals die gleichen graphischen Elemente in unterschiedlichen Domänen verwendet. GMF verfügt über eine klare Trennung zwischen graphischer Representation und Modell. Erst durch die *Mapping Definition* wird die Beziehung zwischen diesen beiden Teilen erstellt. Somit lassen sich sowohl unterschiedliche graphische Editoren für dasselbe Modell einfach erstellen, als auch bestehende graphische Elemente einfach auf andere Anwendungsgebiete übertragen. Beispielsweise wäre es vorstellbar die graphischen Komponenten aus dem UML Klassendiagramm zu übernehmen um einen MOF Editor zu generieren. Die *Mapping Definition* ermöglicht es die drei bereits behandelten Modelle zu verknüpfen. Wiederum existiert ein Wizard *Guide Mapping Model Creation*, welcher eine Ausgangsbasis für das eigene Projekt generieren kann. Am Wurzelement *Mapping* hängt der Knoten *Canvas Mapping* welcher Referenzen auf die Wurzelemente der drei vorigen Modelle enthält.

Wie schon im Kapitel EMF erläutert, ist EMF als Baum aufgebaut und alle Elemente müssen einen Container haben. Für die Referenzen des Wurzelements einer Domäne werden sogenannte *Top Node References* erstellt.

Diese können somit direkt auf die Diagramfläche gezeichnet werden da sie ohne ein anderes Element existieren können. Eine *Top Node Reference* beinhaltet ein *Node Mapping* welches ein Domain Element mit einem Diagrammknoten und einem Palettenelement verknüpft.

Node Mappings können sogenannte *Child References* beinhalten welche dazu dienen die Kindelemente des modellierten Domänenelements ebenfalls an graphische Notationen zu binden. In dem dargestellten Beispiel 2.13 wird eine *Top Node Reference* für die Klasse und eine *Link Reference* für die Assoziation erstellt. Die

Feature Label Mapping Elemente ermöglichen es Attribute der Domain mit *Diagram Label* aus der *Graphical Definition* zu verknüpfen. Dies geschieht für die Klasse zum einen für den Klassennamen, zum anderen für alle Attribute, welche über eine *Child Reference* der Klasse mit dem im Klassenrechteck enthaltenen Fach verbunden werden (siehe 2.3.4). Die Assoziation enthält lediglich ihren Namen. Die Rollen einer Assoziation befinden sich in diesem Editor in der Klasse und somit verbindet eine Assoziation zwei Attribute.

Da die in dieser Arbeit beschriebene Implementierung als *Proof of Concept* gedacht ist, und kein fertiges Produkt darstellt, wurden solche Konzepte noch nicht berücksichtigt und würden beim Zusammenfügen der Editoren verloren gehen.

2.3.7 Code Generierung

Das Generatormodell wird in der Regel automatisch von GMF aus der *Mapping Definition* erstellt und beinhaltet die nötigen Informationen zur Codegenerierung. Zusätzlich zu den in den anderen Modellen bereitgestellten Informationen enthält es weitere Hinweise für den Generator über das zu erstellende Plug-in. Dies ähnelt dem *.genmodel* aus EMF. Diese Informationen steuern beispielsweise ob ein Plug-in oder eine eigenständige Eclipse Applikation erstellt werden sollen, welches Layout verwendet wird und dergleichen. Der Quellcode wird durch den Aufruf *generate Diagram code* im Auswahlménü der entsprechenden *.gmfgen* Datei erstellt.

2.4 Modelltransformationen

2.4.1 Einleitung

Modelltransformationen spielen in der modellgetriebenen Softwareentwicklung eine wesentliche Rolle. Neben dem naheliegendsten Anwendungsbereich ein Modell in ein anderes überzuführen, dienen Modelltransformationen auch zum Erweitern von Modellen, um Code aus ihnen zu generieren oder mehrere domänenspezifische Modelle zusammenzuführen. Im Grunde geht es immer darum, aus einem bestimmten Input von einem oder mehreren Modellen über definierte Regeln den gewünschten Output zu erzeugen.

Modelltransformationen können auf verschiedene Arten definiert werden. Kleppe et al. [1] definiert eine Transformation als das automatische Generieren eines Zielmodells aus einem Ausgangsmodell, entsprechend einer bestimmten Definition. Diese Definition einer Transformation ist ein Satz von Regeln, welche die

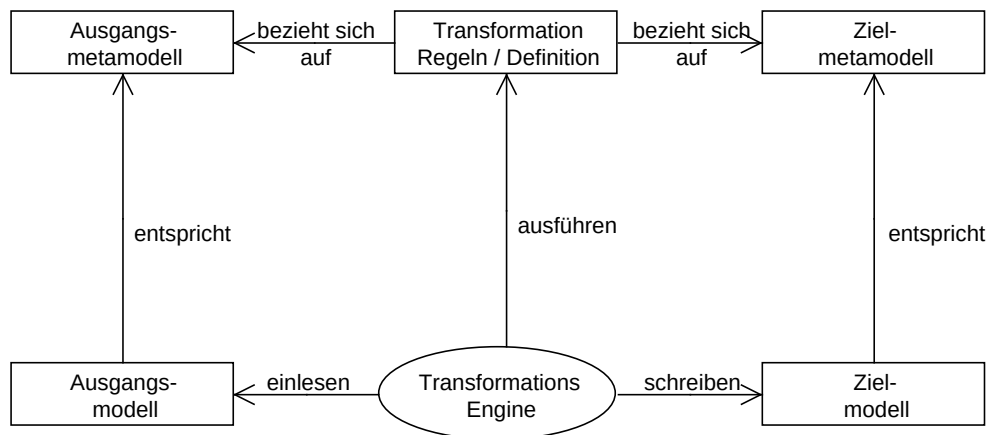


Abbildung 2.14: Basis Konzept einer Modelltransformation [6]

Transformation vom Ausgangsmodell in das Zielmodell beschreiben. Eine einzelne Regel beschreibt wie ein oder mehrere Konstrukte aus der Ausgangssprache in ein oder mehrere Konstrukte der Zielsprache transformiert werden können. Im MDA Leitfaden [24] der OMG wird Modelltransformation einfach als der Umwandlungsprozess von einem Modell in ein anderes Modell desselben Systems definiert.

Abbildung 2.14 zeigt einen Überblick über das Basis Konzept einer Modelltransformation. Sie zeigt ein einfaches Szenario mit einem Ausgangsmodell und einem Zielmodell. Beide Modelle sind jeweils konform zu ihrem Metamodell, in welchem die abstrakte Syntax beschrieben ist. Die Transformation wird in Bezugnahme auf diese beiden Metamodelle beschrieben. Diese Definition wird dann auf den zwei konkreten Modellen von einer *Transformation Engine* ausgeführt.

Eine Transformation von einem Modell in ein anderes Modell kann auf drei Arten erfolgen, man unterscheidet zwischen vertikaler, horizontaler und schiefer Transformation [15].

- *Horizontaler Transformation* - Unter horizontaler Transformation versteht man die Veränderung der Struktur eines Modells, bekanntestes Beispiel hierfür wäre eine Refaktorisierung, die Transformationen zwischen Sprachen derselben abstrakten Ebene.
- *Vertikale Transformation* - Eine vertikale Transformation verändert die Abstraktionsebene eines Modells, das allgemeinste Beispiel dafür ist eine PIM

zu PSM Transformation, bei welcher das PIM mit plattformspezifischen Informationen erweitert wird.

- *Schiefe Transformation* - Eine Transformation die sowohl die Struktur als auch die Abstraktionsebene ändert, bezeichnet man als schiefe Transformation.

Eingeteilt können Modelltransformationen auch nach ihren speziellen Charakteristika werden [6], in der nächsten Auflistung werden die Charaktere der obersten Ebene aufgelistet. Czarnecki et. al. beschreiben diese Unterteilung nicht nur speziell für Modelltransformationen. In Bezugnahme auf das Thema dieser Arbeit wird der Fokus der folgenden Erläuterungen auf diesem Bereich gelegt.

Spezifikation Manche Transformationsansätze bieten die Möglichkeit verschiedene Bedingungen vor und nach der Transformation zu prüfen, ausgedrückt in Form von OCL. Im Normalfall wird das Ausgangsmodell vor und das Zielmodell nach der Transformation auf die Einhaltung der gegebenen Bedingungen geprüft.

Transformationsregeln Die Transformationsregeln sind in der Komponente des jeweiligen Ansatzes zusammengefasst, die den eigentlichen Umwandlungsprozess durchführt. Czarnecki et. al. untergliedern diese Eigenschaft noch in weitere Charaktere, wie der *Domaine*, der *Multidirectionality* oder auch der *Reflection*.

Rule application control Darunter wird die Reihenfolge und Art in der das Quellmodell durchlaufen wird verstanden, diese Eigenschaft wird noch weiter unterteilt in *Location determination* und *Rule scheduling*. Beispiele für *Location determination* sind deterministische, nicht deterministische oder iterative Vorgehen.

Rule organization Manche Ansätze ermöglichen das Gruppieren und Strukturieren verschiedener Transformationsregeln. Diese Eigenschaft kann wieder weiter unterteilt werden.

- *Modularity Mechanisms* - Zusammenfassen von Regeln in Module
 - *Reuse Mechanisms* - Wiederverwendbarkeit von Regeln
 - *Organizational Structure* - Einteilung der Regeln passend zur Struktur von Quell- oder Zielmodell.
-

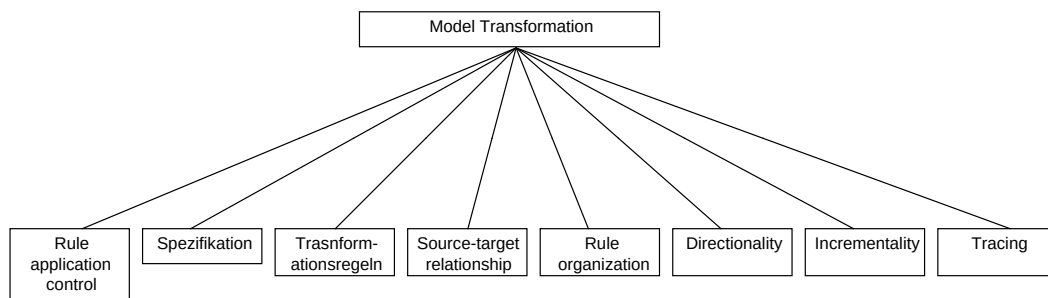


Abbildung 2.15: Eingeteilt Modelltransformationen in Form eines Merkmaldiagramms

Source-target relationship Beschreibt den Zusammenhang zwischen Quell- und Zielmodell dahingehend, ob das Ergebnis der Transformation ein neues Zielmodell erzeugt oder das Quellmodell verändert wird.

Incrementality Hierbei wird zwischen drei Eigenschaften unterschieden.

- *Target Incrementality* - Erweiterungen im Ausgangsmodell werden auch in das Zielmodell übergeführt.
- *Source Incrementality* - Nur tatsächliche Änderungen der Ausgangsmodelle werden in die inkrementelle Transformation übernommen.
- *Erhalt von benutzerdefinierten Anpassungen* - Die vom Benutzer erweiterten Teile des Zielmodells werden erhalten.

Directionality Charakterisiert ob Transformationen unidirektional oder multidirektional sind. *Model Transformation By-Example* ist ein Ansatz, der Transformationen in beide Richtungen ermöglicht.

Tracing Manche Transformationsansätze ermöglichen das Mitprotokollieren der ausgeführten Befehle, damit nachvollzogen werden kann welche Regeln auf welche Elemente tatsächlich angewandt wurden.

Im Folgenden wird auf eine dezidierte Transformationssprache näher eingegangen, dazu wurde die Sprache ATL [17] gewählt. Des Weiteren wird noch ein Konzept aus der *ATLAS Model Management Architecture* näher erläutert, der *ATLAS Model Weaver*, welcher im MTBE-Prototyp ebenfalls zum Einsatz kommt.

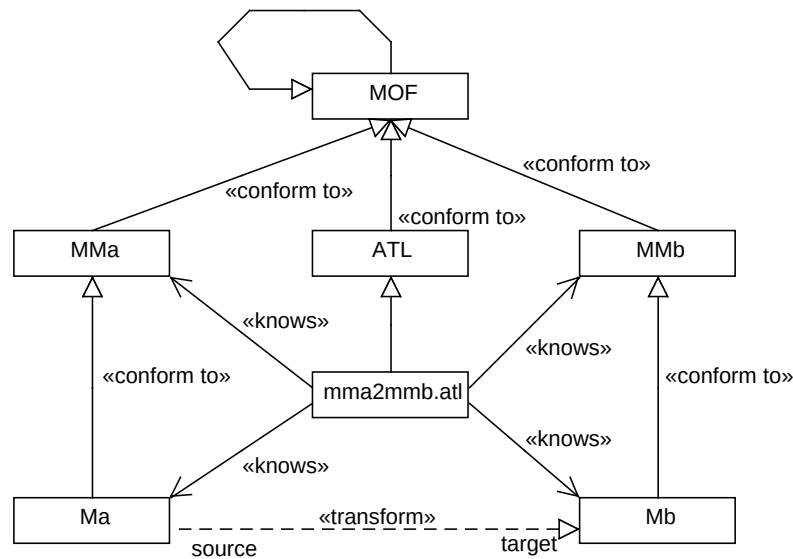


Abbildung 2.16: ATL Transformation

2.4.2 Atlas Transformation Language

Die *Atlas Transformation Language* (ATL) wurde von der *ATLAS INRIA & LINA* Forschungsgruppe als eine Programmiersprache zur Transformation von Modellen entwickelt und wurde zu einem wesentlichen Teil des *Eclipse M2M Projekts*. Das ATL Design zielt darauf ab mit den MDA Standards konsistent zu bleiben, im speziellen mit MOF QVT [26]. Sie ist eine der vier Grundpfeiler der *ATLAS Model Management Architecture*. ATL wird als eine hybride Sprache bezeichnet, das heißt sie ist sowohl eine deklarative als auch eine imperative Sprache. Die abstrakte Syntax dieser Modelltransformationssprache basiert auf einem Metamodell, daher ist eine ATL Transformation selbst auch ein Modell.

Das folgende Listing 2.14 zeigt beispielhaft wie eine Transformation mit ATL aufgebaut ist.

Listing 2.14: ATL Beispieltransformation UML2ER

```

1 module UML2ER;
2 create OUT : ER from IN : UML;
3 rule P2A {
4 >from p : UML! Property (
5     p.owningClass.ocIsUndefined() = false and
6     p.association.ocIsUndefined()
7 )
8 to a : ER!Attribute(
9     name <- p.name,
10    entity <- p.owningClass
11 )

```

Es besteht aus einem Satz von Regeln, die die einzelnen Elemente des Ausgangsmodells in Elemente des Zielmodells überführen. In diesem konkreten Beispiel wird das UML Property auf das ER Attribute *gemappt*. Allgemein wird der Aufbau eines ATL Moduls untergliedert in: [2]

- *Header* - In diesem Bereich werden Name des Moduls sowie Quell- und Zielmodell angegeben. Die Syntax ist wie folgt definiert.

Listing 2.15: ATL Code

```

1      module module_name;
2      create output_models [from|refines]
        input_models;

```

- *Import* - Zusätzlich existierende ATL Bibliotheken für die Transformation können hier falls gewünscht importiert werden.

Listing 2.16: ATL Code

```

1      uses extensionless_library_file_name;

```

- *Hilfsfunktionen* - *ATL helper* sind vergleichbar mit einer Java Methode, sie ermöglichen es den ATL Code zu unterteilen, um Codestücke immer wieder an beliebigen Stellen aufrufen zu können. *ATL Helper* sind laut folgendem Schema definiert.

Listing 2.17: ATL Code

```

1      helper [context context_type]? def :
        helper_name(parameters) : return_type =
        exp;

```

- *Regeln* - In ATL wird zwischen zwei verschiedenen Arten von Regeln unterschieden, den *matched Rules* und den *called Rules*.
 - Mit den sogenannten *matched Rules* ist es möglich zu spezifizieren für welche Art von Quellelement welches Zielelement generiert werden soll, und wie die generierten Elemente initialisiert werden müssen.
 - *Called Rules* sind das entsprechende Konstrukt für die imperative Programmierung. Sie können im Grunde als eine Art *Helpers* angesehen werden.
-

2.5 Atlas Model Weaver

Unter dem Begriff *Model Weaving* wird das Verknüpfen von Modellen mit Hilfe einer grafischen Schnittstelle verstanden. Die Information über die Verknüpfung der Modelle wird dabei in einem weiteren Modell gespeichert, dem sogenannten *Weaving Modell*.

In dem in dieser Arbeit beschriebenen MTBE Prototypen wurde auf den *Atlas Model Weaver* (AMW) [10] zurückgegriffen. Dieser *Model Weaver* ist ein Teilprojekt des *Eclipse Generative Modeling Tools (GMT) Projekts* und lässt sich relativ gut integrieren und erweitern.

Der *Atlas Model Weaver* ist ein Werkzeug um verschiedene Arten von Verbindungen zwischen Modellen sowie Metamodellen herzustellen, und diese serialisiert abzulegen. Die definierten Verbindungen werden in einem *Weaving Modell* gespeichert, wahlweise in Form eines Ecore-Modells oder in einem eigenen Format (*.amw). Wie auch schon die zuvor vorgestellte ATL Transformationssprache ist auch der *Atlas Model Weaver* ein Teil der AMMA Plattform.

2.5.1 Core Weaving Metamodel

Der Atlas Model Weaver basiert auf einem (Core-)Weaving Metamodell, welches die grundlegenden Verknüpfungen zwischen Modellen beschreibt.

Das *WElement* stellt das Hauptelement dar, von ihm erben alle anderen Elemente. Es enthält als Attribute einen Namen und eine Beschreibung. Das Wurzelement wird vom *WModel* repräsentiert, es beinhaltet alle Modellelemente. *WLink* stellt den Verbindungstyp zwischen den Modellelementen dar. Es hat eine Referenz *end* welche beliebige Verbindungen zwischen Elementen ermöglicht. Jedes *WLinkEnd* stellt einen Endpunkt da, es referenziert die Elemente über ein *WElementRef*. *WModelRef* ist ähnlich dem *WElementRef*, aber es beinhaltet zusätzlich Modellreferenzen. Beide erben von der abstrakten Klasse *WRef*.

Das Metamodell des *Atlas Model Weaver* wurde möglichst einfach gehalten, erst durch die Erweiterung von diesem Basis Metamodell durch weitere *Weaving Metamodelle* wird die genau semantische Bedeutung definiert. Als praktisches Beispiel hierzu ist das *Weaving Meta Modell* des MTBE Plug-ins zu erwähnen, siehe dazu *weavingmodel.ecore*.

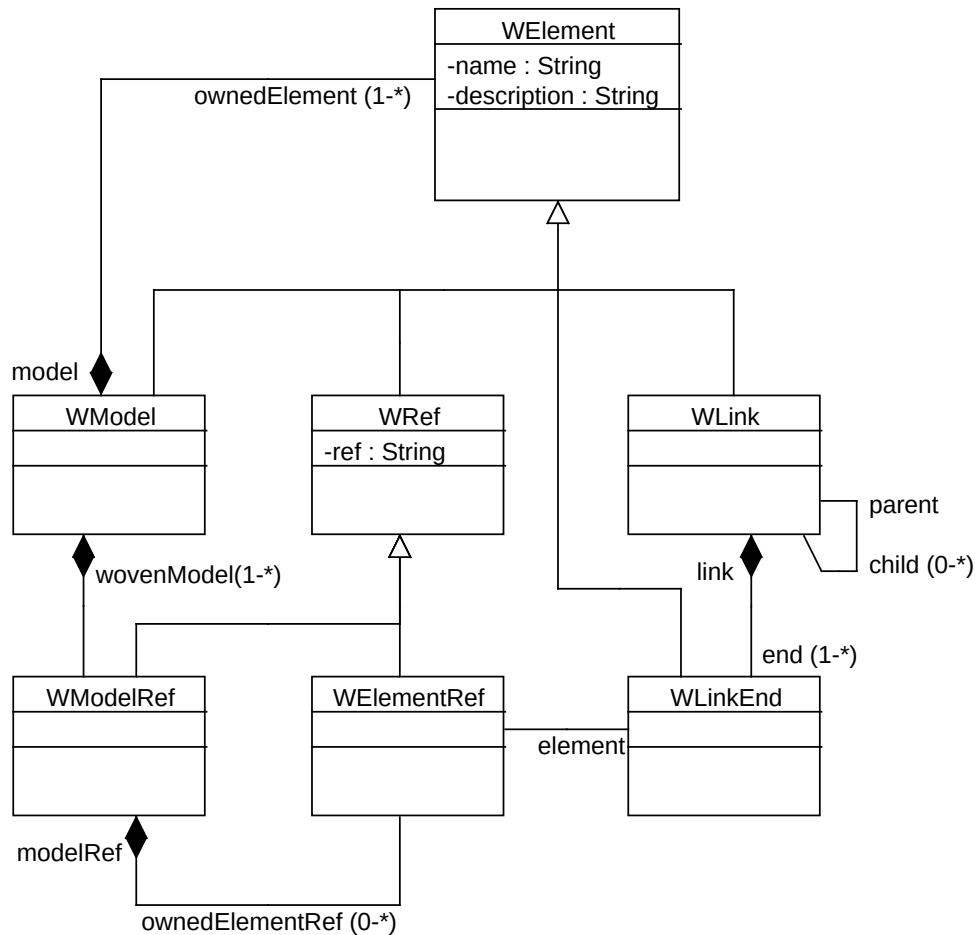


Abbildung 2.17: AMW Core Meta Model [7]

2.5.2 AMW Weaving Tool

Der *Atlas Model Weaver* wurde wie auch MTBE als Plug-in für die *Eclipse Plattform* entwickelt. Zur Verwaltung der Basis Modelle verwendet der *Model Weaver* das *Eclipse Modeling Framework Project* [45]. Ziel der Entwickler war es ein stabile Basis zu schaffen, die einem breiten Spektrum an Anwendungsfällen passende Erweiterungsmöglichkeiten bietet. Basierend auf der Idee der *Eclipse Contributions* wurden neben der Umsetzung als Plug-in, *Extension Points* definiert.

Abbildung 2.18 zeigt die erweiterbare Architektur des *Atlas Model Weavers*. Basis ist die Eclipse EMF Plattform, über dessen API der Zugriff auf Modelle und Metamodelle erfolgt, die wie schon erwähnt auf dem Ecore Metametamodell aufbauen. Das zentrale Element dieses Weaving Tools ist die *AMW workbench*, welche mehrere Extension Points anbietet.

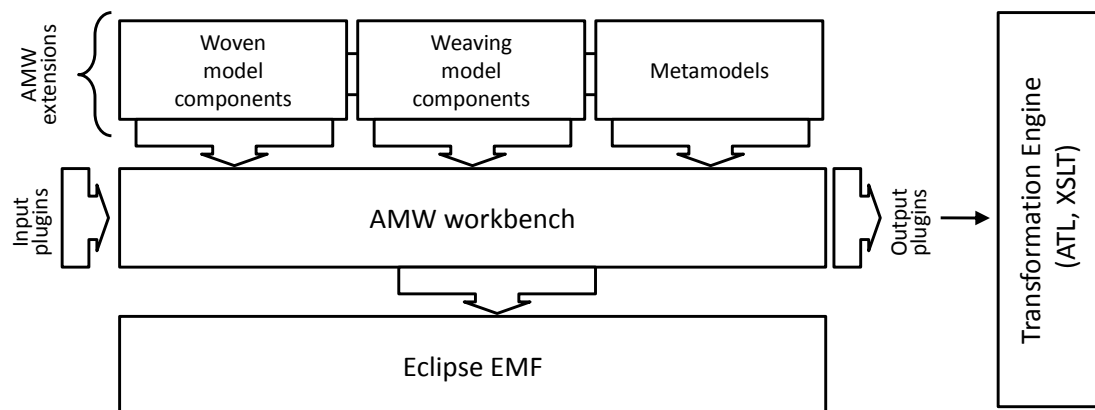


Abbildung 2.18: AMW Architektur [10]

Woven Model Extensions Der *Woven Model Extension Point* ermöglicht es neue Benutzeroberflächen zur Darstellung der Modelle zu entwickeln.

Weaving Model Extensions Über den *Weaving Model Extension Point* können die unterschiedlichen Aktionen, zum Beispiel das Erstellen neuer *Weaving Elemente*, gesteuert werden.

Metamodel Extensions Die *Metamodel Extension* ermöglicht das Erweitern des Metamodells. Da AMW auf die *EMF Reflection API* zurückgreift können *Metamodel Extensions* einfach durch das Angeben der *Metamodel Extension Datei* eingebunden werden.

Folgende Abbildung 2.19 zeigt das *Atlas Model Weaver Tool*, im Rahmen der Anwendung des MTBE Plug-ins. Die Oberfläche ist in drei Teile gegliedert. Im rechten und linken Container befinden sich die zwei Metamodelle, in diesem Beispiel ist das *leftModel* ein ER-Diagramm und das *rightModel* ein UML-Klassendiagramm. In der Mitte dieser beiden Diagramme befindet sich das eigentliche *Weaving Modell*, welches die definierten Verbindungen zwischen dem linken und rechten Modell beinhaltet.

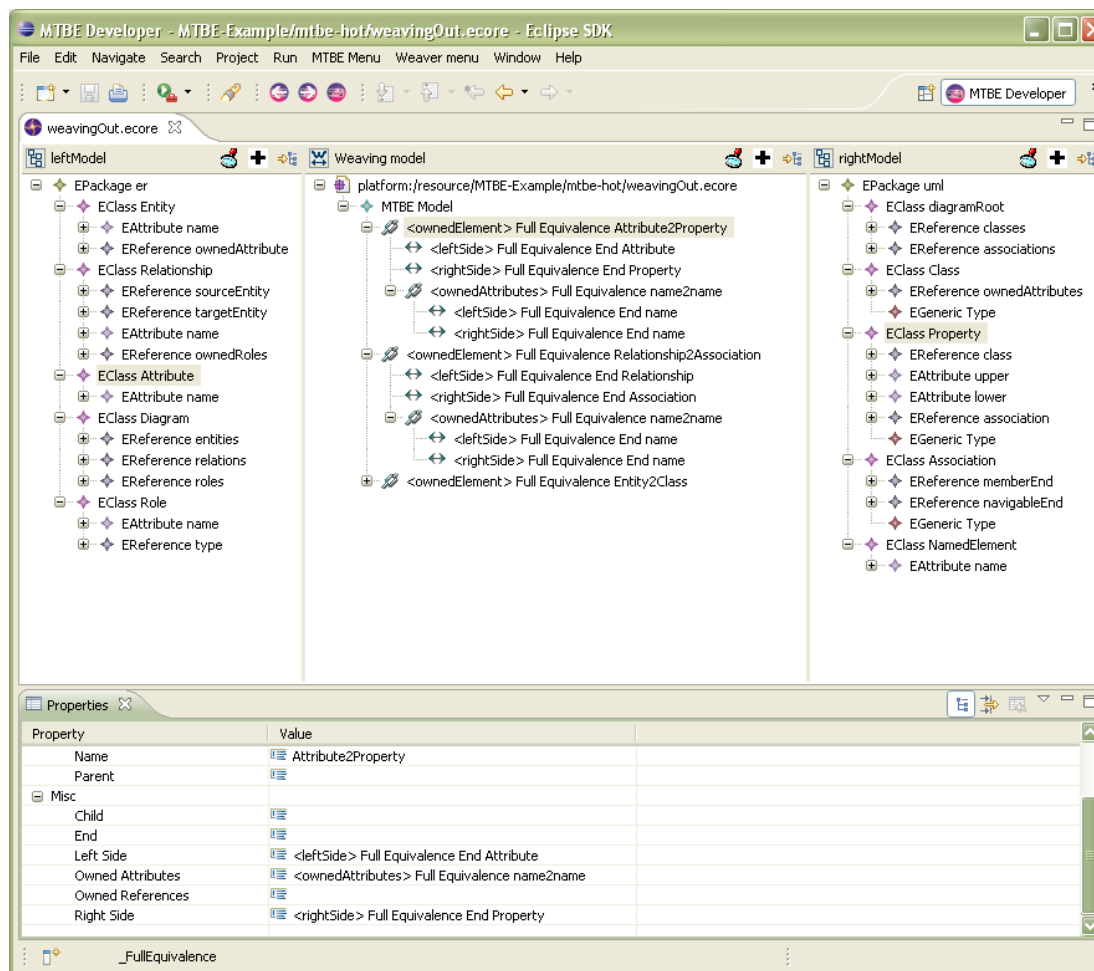


Abbildung 2.19: Atlas Model Weaver

2.6 By-Example Ansätze

2.6.1 Einleitung

Meist ist es leichter etwas anhand einfacher Beispiele zu erfassen als aus einer abstrakten Idee. Darauf bauen *By-Example* Ansätze auf. Sie versuchen Anhand von einfachen Beispielen allgemeine gültige Regeln zu erstellen. Der erste *By-Example* Ansatz wurde in den 1970er Jahren von Moshé M. Zloof, Mitarbeiter des IBM Research Centers, entwickelt. Sein sogenanntes *Query-By-Example* (QBE) [50] ermöglicht SQL-Abfragen anhand von Beispiel Tabellen zu erstellen. Der Grundgedanke hinter diesem Ansatz kann wie folgt zusammengefasst werden. Der Vorteil für den Benutzer liegt darin,

- dass er mit wenig Information und Wissen über die dahinterliegenden Konzepte zu brauchbaren Ergebnissen kommt,
- dass er sich nur mit den relativ einfachen Sprachkonzept von QBE ausein-

anderssetzen muss.

QBE Varianten findet man heutzutage in verschiedenen Datenbanksystemen wieder. Ein Beispiel dafür wäre das Datenbanksystem *Microsoft Access*. Auf dieser Idee bauen bis heute aber nicht nur Ansätze im Datenbank Bereich auf, sondern noch in vielen anderen, wie zum Beispiel in Bereichen der Softwareentwicklung. Im täglichen Leben begegnet uns ein *By-Example* Ansatz bei der Suche im Internet mit der Funktion “Ähnliche Seiten anzeigen”.

Im Folgenden werden einige Konzepte beschrieben, allen voran der Pionier der *By-Example* Ansätze *Query-By-Example*.

2.6.2 Query-By-Example

Query-By-Example [51] [33] ist als eine High-Level Datenbankmanagementsprache entwickelt worden, die es Anwendern ohne Programmierkenntnisse ermöglichen soll relationale Datenbanken abzufragen, zu aktualisieren und auch zu definieren. QBE-Befehle werden mittels einer leeren Tabelle, einem sogenannten Tabellengerüst (*skeleton table*) eingegeben. Eine Abfrage würde zum Beispiel so funktionieren, dass der Anwender das Tabellengerüst mit möglichen Ergebnissen ausfüllt, als Ergebnis erhält er, basierend auf der sich daraus ergebenden Abfrage dann die vollständige Tabelle.

Folgendes Beispiel soll die einfache Anwendung von QBE veranschaulichen. Ausgegangen wird von einer Datenbank mit zwei Relationen Student(Matrikelnr, Vorname, Nachname, Betreuer) und Professor(PID, Titel, Vorname, Nachname) auf.

Nun sollen die Betreuer von allen Studenten mit dem Namen “Müller” zurückgeliefert werden.

In QBE würde die Abfrage wie in Tabelle 2.6.2 aussehen. Sie besteht aus zwei leeren Tabellen, der Relation Student und der Relation Professor. Die Einträge _BETREUER, _VORNAME und _NACHNAME sind Variablen, gekennzeichnet durch den Unterstrich, und “Müller” eine Konstante. Das “P” (PRINT) vor einer Variablen bedeute, dass alle Werte dieser Variablen als Ergebnis zurückgeliefert werden sollen.

Im Vergleich dazu würde die entsprechende SQL-Abfrage folgend lauten:

Student	Matrikelnr	Vorname	Nachname	Betreuer
			Müller	_BETREUER

Professor	PID	Titel	Vorname	Nachname
	_BETREUER		P._VORNAME	P._NACHNAME

Tabelle 2.1: QBE-Query um die Betreuer aller Studenten mit dem Namen “Müller” zurückzugeben.

Listing 2.18: SQL-Query

```

1  select pr.Vorname, pr.Nachname from Student as
      st, Professor as pr where st.Betreuer =
      pr.PID and st.Nachname='Mueller'

```

Die QBE Pendant zu den SQL Befehlen INSERT, UPDATE, DELETE sind die Buchstaben I, U, D. Sie werden wie der Befehl P mit einem nachstehenden “.” verwendet. Würde man zum Beispiel alle Studenten mit dem Namen “Müller” löschen, würde es wie folgt eingegeben werden:

Studenten	Matrikelnr	Vorname	Nachname	Betreuer
.D			Müller	

Tabelle 2.2: QBE-Query um alle Studenten mit dem Namen “Müller” zu löschen.

2.6.3 Programming-By-Example

Programming-By-Example (PBE) beschreibt im Gegensatz zu QBE keine konkrete Umsetzung, PBE ist ein Sammelbegriff für verschiedene Ansätze, die das Erstellen einer Anwendung anhand von Beispielen über Verhalten und Ziel des Programmes ermöglichen. Die Motivation dahinter ist schon wie bei QBE, dem Benutzer, vor allem dem ohne Programmierkenntnisse, das Programmieren oder Anpassen von Programmen zu ermöglichen, ohne Programmiersprachen wie C oder Java lernen zu müssen.

Im Allgemeinen spricht man von PBE wenn Anwendungen mit Hilfe von Beispielen generiert werden, wobei die Beispiele als Platzhalter für die Abstraktion dienen [5]. Eine alltägliche Anwendung die PBE implementiert ist das Erstellen von Excel Macros. Durch drücken des Buttons “Makro Aufnehmen” erhält der Computer den Befehl “Whatch what I do” [5]. Der Anwender “zeigt” dem Computer was er gerne hätte. Er markiert Beispielsweise eine Zelle rot, und Excel generiert den entsprechenden Code.

Der erste PBE Ansatz stammt aus dem Jahre 1975 von David C. Smith mit dem Namen *Pygmalion* [37]. *Pygmalion* enthält zwei für die damalige Zeit völlig neue Konzepte. Auf der einen Seite das Programmieren anhand von Beispielen aber auch die Idee der Icons, welche heute weit verbreitet ist. *Pygmalion* ist eine zweidimensionale, visuelle Programmierumgebung. Einige der Konzepte aus *Pygmalion* haben durchaus eine Berechtigung in der heutigen Zeit zu bestehen [5].

- Pygmalion ermöglichte es, Ideen als Entwurf am Bildschirm auszuarbeiten, und wieder auf neuen Daten auszuführen.
- Programme werden als Filme dargestellt. Der Programmierer sieht sein Programm als eine Serie von Bildern. Er “programmiert” indem er ausgehend von einem Startbild, jeweils aus dem hervorgehenden ein neues Bild erstellt. Wird nun das Programm ausgeführt, ist es ähnlich als ob man sich einen Film ansieht. Der Unterschied ist, dass abhängig vom Input jeder *Pygmalion Film* ein anderer ist.
- Es führt Icons als das Basis Element zum Darstellen und Steuern eines Programmes ein.
- Programmiert wurde durch das Bearbeiten von Entwürfen und das Aufnehmen dieser Schritte. Befehle müssen nicht mehr abstrakt niedergeschrieben werden.

- Beispieldaten sind immer konkret niemals abstrakt. Jedes Bild enthält konkrete Beispiele der Programmdaten.
- Es verwendet analoge Darstellungen der Daten, um die mentalen Modelle den Computer Modellen möglichst ähnlich zu halten.

Kapitel 3

Model Transformation By-Example

3.1 Einleitung

Der von Wimmer et. al. [49] vorgestellten Ansatz zur Modelltransformation basiert auf den im Abschnitt 2.6.2 behandelten *By-Example* Ansätzen wie dem *Query-By-Example* oder dem *Programming-By-Example*. Bei *Model Transformation By-Example* (MTBE) werden die Mappings zwischen konkreten Domainmodellen auf der Modellebene M1 definiert, und aus diesen Mappings dann “By-Example” die entsprechenden Transformationsregeln generiert. Um aus den Beispiel Mappings brauchbare Transformationsregeln erstellen zu können, müssen zusätzlich Korrespondenzen zwischen konkreten und abstrakten Syntax-Elementen in einer Notation beschrieben werden. Diese Notation enthält die Bedingungen (*Constraints*), wie die Elemente der abstrakten Syntax mit jenen der konkreten Syntax in Beziehung stehen.

In den folgenden Abschnitten der Arbeit wird sowohl die Beschreibung des MTBE Konzepts sowie die Vorstellung der Implementierung mit einem konkreten Beispiel erläutert. Als einfaches Beispiel werden zwei UML Klassen “Student” und “Professor” mit einer *one-to-many* Beziehung herangezogen. Zusätzlich wird diese Problemdomäne noch als ER Diagramm dargestellt.

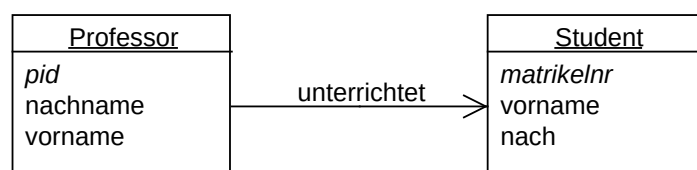


Abbildung 3.1: Verwendetes Beispieldiagramm für UML

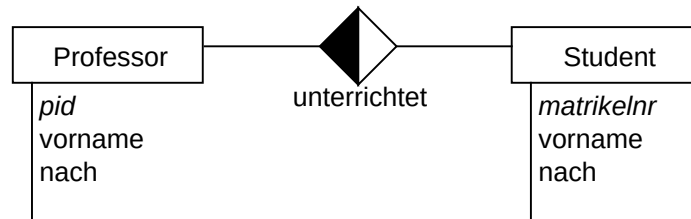


Abbildung 3.2: Verwendetes Beispieldiagramm für ER

Im folgenden Abschnitt wird nun überblicksmäßig der konzeptuelle Aufbau von *Model Transformation By-Example* erläutert, sowie das von Michel Strommer und Manuel Wimmer vorgeschlagene Framework für eine mögliche Implementierung vorgestellt.

3.2 Model Transformation Generation By-Example

3.2.1 Konzept

Die Grundidee hinter *Model Transformation By-Example* ist die Transformation auf Basis der konkreten Syntax der Modellierungssprachen. Dazu ist es wichtig, dass die Beziehung zwischen der abstrakten und der konkreten Syntax sowie die Mappings zwischen diesen ausreichend definiert sind. Die Beziehung zwischen diesen beschreibt Abbildung 3.3. Das Paket *Abstract Syntax* enthält alle Elemente der abstrakten Syntax, wie zum Beispiel dem Metamodell, und das Paket *Concrete Syntax* alle graphischen Elemente.

Das Paket *as_2cs* beschreibt wie die Elemente der abstrakten Syntax auf denen der konkreten Syntax abgebildet werden, dies erfolgt in Form von Tripel, die wie folgt aufgebaut sind.

$$Triple := \langle as_E, cs_E, const(as_E)? \rangle \quad (3.1)$$

Die ersten zwei Teile der Gleichung stehen für das Element der abstrakten sowie der konkreten Syntax, der dritte Teil $const(as_E)$ ist optional. Er steht für eine Bedingung, wie zum Beispiel einer in OCL, die die Beziehung zwischen as_E zu cs_E beschreibt. Ist dieser Wert nicht gesetzt, handelt es sich um ein direktes Mapping.

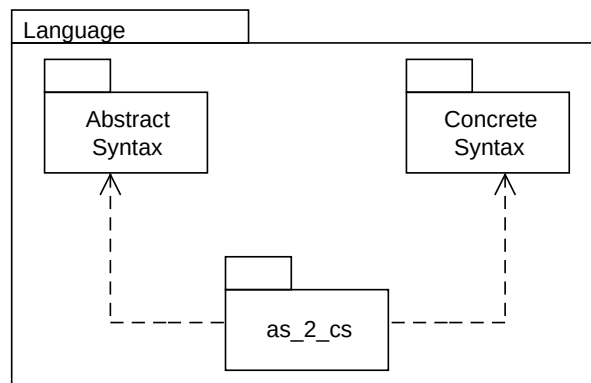


Abbildung 3.3: Beziehung zwischen abstrakter und konkreter Syntax

Die Modell Transformation mit MTBE erfolgt, basierend auf dem Paper [40], im Grunde in fünf Schritten, siehe Abbildung 3.4.

- *Schritt 1* - Im Initialisierungsschritt erfolgt die Definition der zwei Ausgangsmodelle aus einer gemeinsamen Problemdomäne. Dem Benutzer steht es offen ein einzelnes Modell, welches alle Sprachkonzepte abbildet oder mehrere Modelle zu wählen, wobei jedes dieser auf gewisse Aspekte eingeht. In den meisten Fällen erweist sich die erste Variante als vorteilhafter. Beide Modelle müssen zwei Anforderungen erfüllen.
 - Erstens müssen beide mit ihren jeweiligen Metamodellen übereinstimmen,
 - und zweitens müssen die zur Transformation verwendeten Beispiele alle verfügbaren Modellierungskonstrukte abdecken.
- *Schritt 2* - Im zweiten Schritt geht es nun um die Definition der semantischen Übereinstimmungen zwischen den Modellen, wobei der Benutzer Mappings zwischen Elementen des linken und denen des rechten Modells bestimmt. Wichtiger Unterschied hierbei zu anderen Ansätzen ist, dass dieser Schritt wie schon erwähnt auf der Modellebene M1 durchgeführt wird.
- *Schritt 3* - Nachdem die Mappings definiert worden sind, wertet die *Analyzer*-Komponente diese aus und erkennt die entsprechenden Korrespondenzen zwischen den zwei gegebenen Metamodellen. Der Output dieses Schrittes ist ein *Weaving Modell* auf Basis dieser beiden Metamodelle.
- *Schritt 4* - Aufbauend auf dem *Weaving Modell* aus dem letzten Schritt, kann nun der *Modeltransformation Generator* (MTGen) ausgeführt werden. Dieser generiert aus dem in Schritt 3 definiertem *Weaving Modell* ein

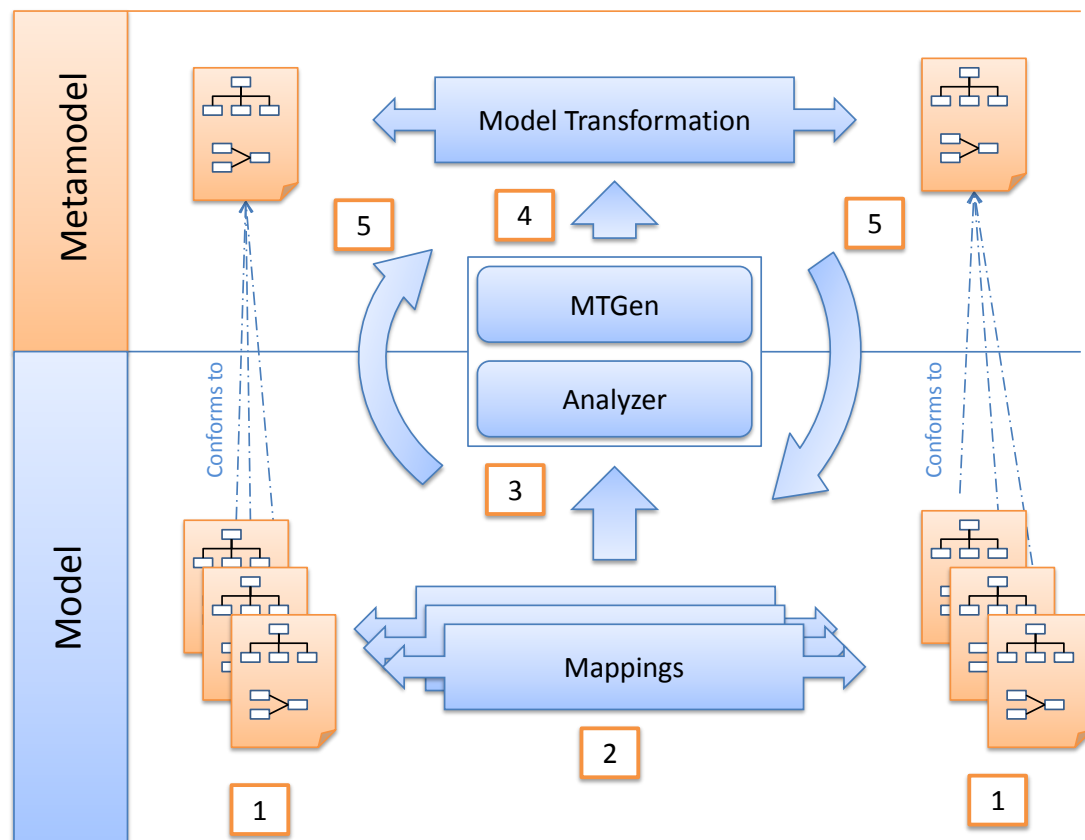


Abbildung 3.4: MTBE Framework

Transformationsprogramm basierend auf dem ursprünglichen Metamodellen. Aus diesem Programm können dann beliebige, auf dem Quellmetamodell basierende Ausgangsmodelle entsprechend in, auf dem Zielmetamodell basierende, Modelle transformiert werden.

- *Schritt 5* - Grundsätzlich wurden mit dem vierten Schritt die Transformationsregeln erstellt. In diesem abschließenden Schritt wird nun aber noch die Vollständigkeit der zuvor generierten Regeln überprüft. Der Benutzer sieht in wie weit seine Beispiel Mappings ausreichend waren, er kann nun bei Bedarf weitere Mappings hinzufügen oder manuell Regeln hinzufügen und diese erneut durch MTBE analysieren lassen. MTBE stellt demnach einen Iterativen Prozess dar.

3.2.2 Framework

Strommer et. al. präsentieren ein Framework zur Modelltransformation *By-Example* [40] aufbauend auf dem Eclipse Framework. Es setzt sich aus mehre-

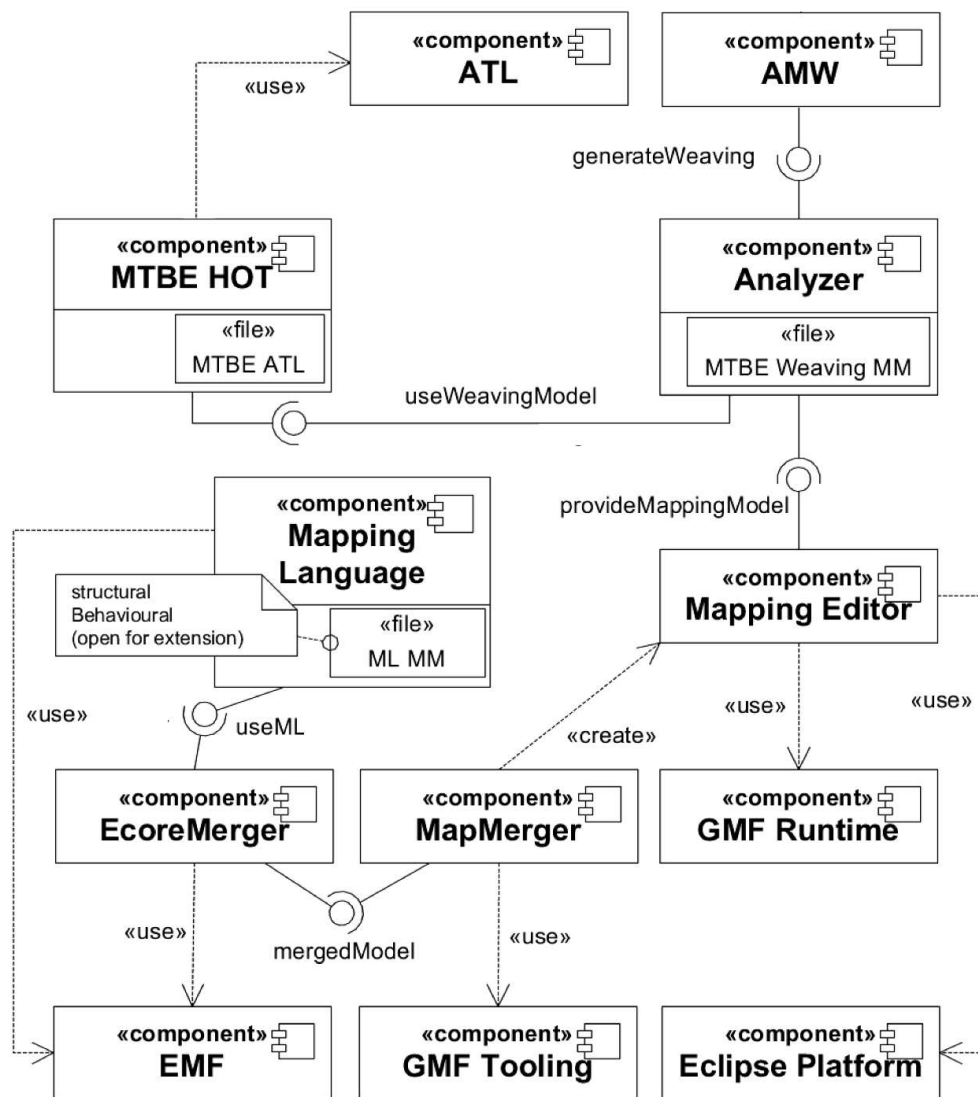


Abbildung 3.5: Komponentendiagramm des MTBE Frameworks

ren Eclipse Projekten wie dem schon im ersten Teil dieser beschriebenen EMF und GMF zusammen. Wobei GMF die Grundlage für die Editoren darstellt, es dient dazu einfach Editoren zu erstellen, oder bereits existierende (z.B.: UML) verwenden zu können. EMF dient sozusagen als Datenspeicher, sowohl für die Metamodelle als auch für die vom Benutzer erstellten Modelle und Mappings.

Die folgende Abbildung 3.5 veranschaulicht die einzelnen Komponenten aus denen sich dieser MTBE Ansatz zusammensetzt.

Merge Komponente

Das MTBE Framework sieht hier zwei Merger vor, zum Einen den *Ecore Merger* und zum Anderen, für die grafischen Editoren, den Mapping Merger. Der

Ecore Merger erstellt aus zwei unterschiedlichen Metamodellen ein gemeinsames, welches als Grundlage zum Erstellen der Mappings verwendet wird. Der *Mapping Merger* generiert durch das Verbinden der GMF-Komponenten aus den zwei schon vorhandenen GMF-Editoren (zb. UML und ER) einen gemeinsamen grafischen Editor.

Model Mapping Language Komponente

Diese Komponente beinhaltet die Spezifikation der einzelnen Benutzer Korrespondenzen basierend auf dem Metamodell *ML MM*. Im derzeitigen Entwicklungsstadium von MTBE wurden sowohl *Structural Mappings*, wie das *Simple Mapping* als auch *Behavioral Mappings*, wie zum Beispiel das *Compound Mapping* umgesetzt. Die Definition dieser Mappings stellt das Herz des MTBE Frameworks dar, alle anderen Komponenten greifen direkt oder indirekt darauf zu.

MTBE Mapping Editor

Bei dieser Komponente handelt es sich um den grafischen Editor der es ermöglicht Beispieldiagramme aus den Elementen beider Ursprungseditoren zu zeichnen und zwischen ihnen Mappings zu definieren. Er basiert wie schon erwähnt auf der *GMF Runtime*.

Analyzer Komponente

Diese Komponente analysiert den Output aus dem generierten *MTBE Mapping Editor* sowie die entsprechenden Metamodelle, mit dem Ziel bestimmte Muster zu erkennen. Der *Analyzer* erhält als Input das mit dem Editor erstellte Diagramm mit den Mappings und den zugehörigen Metamodellen. Der Benutzer kann entscheiden welche Patterns beziehungsweise Algorithmen für die Analyse verwendet werden sollen und diese beliebig kombinieren. Folgende in Abbildung 3.6 dargestellte sechs Patterns stehen dem Benutzer zur Verfügung.

- *Pattern 1* - Dieses Pattern erfasst einfache Mappings. Es erkennt ob Klassen und Verbindungen im Metamodell auf beiden Seiten identisch sind und eindeutig übereinstimmen - *full equivalence*.
 - *Pattern 2* - Das zweite Pattern soll alternative Pfade finden. Nur durch die Analyse beider Modelle in der abstrakten Syntax kann überprüft werden ob solch ein Weg existiert.
-

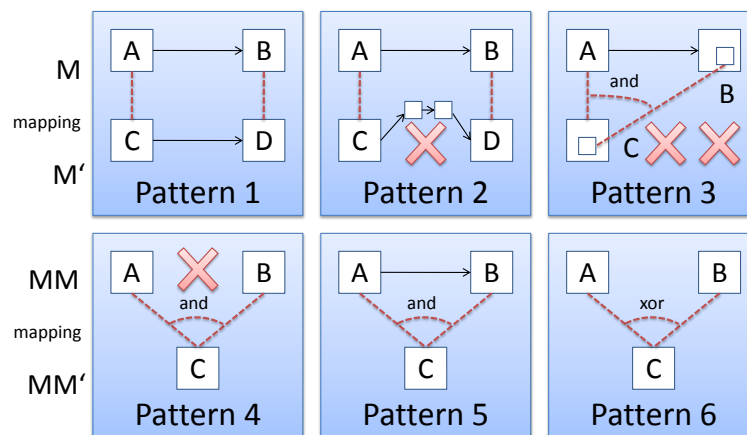


Abbildung 3.6: Standardmäßig zur Verfügung stehende Analyzer Pattern

- *Pattern 3* - Dieses Pattern beschreibt den Fall wenn zwei Konzepte in einer Modellierungssprache eine Klasse darstellen und in der anderen in einer Klasse enthalten sind.
- *Pattern 4* - Ist ähnlich dem Pattern 3, mit dem Unterschied, dass die zwei Elemente A und B voneinander unabhängig sind.
- *Pattern 5* - Dieses ist eine Erweiterung von Pattern vier. Ein Element einer Seite darf höchstens eine eingehende oder ausgehende Kante haben.
- *Pattern 6* - Dieses Pattern erkennt wenn zwei unterschiedliche Konzepte auf eine Klasse gemappt werden.

Der Output dieser Komponente ist ein *Weaving Modell* auf der Metaebene basierend auf den eingezeichneten Mappings.

MTBE-HOT

Einen der wichtigsten Teile des Frameworks stellt die *MTBE HOT* Komponente dar, wobei HOT für *High Order Transformation* steht. Diese Komponente generiert aus dem *Weaving Modell* ATL Code für beide Transformationsrichtungen. Dies erfolgt in zwei Schritten.

- Zuerst wird aus dem *Weaving Modell* ein zum ATL 2006 Metamodell konformes ATL Modell erzeugt.
- Im zweiten Schritt wird mit dem *AM3 extractor* für den Benutzer lesbarer ATL Code erzeugt

Die *MTBE-HOT* Komponente wird so integriert, dass sie für andere Transformationssprachen einfach entsprechend ausgetauscht werden kann. MBTE soll sich nicht nur auf ATL beschränken.

MTBE Workbench

Um MTBE für den Benutzer leicht verwendbar zu machen, wurde ein *MTBE Workbench* entwickelt. Dieser besteht aus vier unterschiedlichen Ansichten, die je nach Anwendungsfall verwendet werden können.

- *Ansicht 1* - In dieser Ansicht soll der Benutzer mit Hilfe des MTBE Editors die Beispiel Modelle zeichnen und die entsprechenden Mappings definieren 3.7.
- *Ansicht 2* - Hier werden dem Benutzer die erkannten Mappings mit Hilfe des *Atlas Model Weaver* angezeigt. Diese Ansicht dient, falls notwendig, auch dem Erweitern und Anpassen des *Weaving Modells* 3.8.
- *Ansicht 3* - In dieser Ansicht kann der erzeugte ATL Code angezeigt und bearbeitet werden 3.9.
- *Ansicht 4* - Diese Ansicht dient dem Testen des generierten Transformationscodes 3.10.

3.2.3 Integration von GMF

Um eines der Hauptziele von MTBE, die Benutzerfreundlichkeit von Modelltransformationen, umsetzen zu können wurde das *Graphical Modeling Framework* in MTBE integriert. Es erweitert MTBE um einen graphischen Editor. Der wichtigste Aspekt der für GMF spricht, war die große Community die hinter diesem Eclipse Projekt steht.

Damit mit Hilfe von GMF ein Editor generiert werden kann, wurden in MTBE eigene *Model Merging* Komponenten integriert.

Ecore File Sharing

Die Funktionsweise von GMF wurde schon ausreichend in Abschnitt 2.3 erläutert. Aus dem *GMF Mapping Definition Model*, welches wiederum auf einem Ecore Metamodell basiert, wird der Editor generiert.

Um aus den zwei Ecore Metamodellen eines zu bilden, wird MTBE um die so genannte *Ecore Merger Komponente* erweitert. (Siehe Abbildung 3.5) Diese

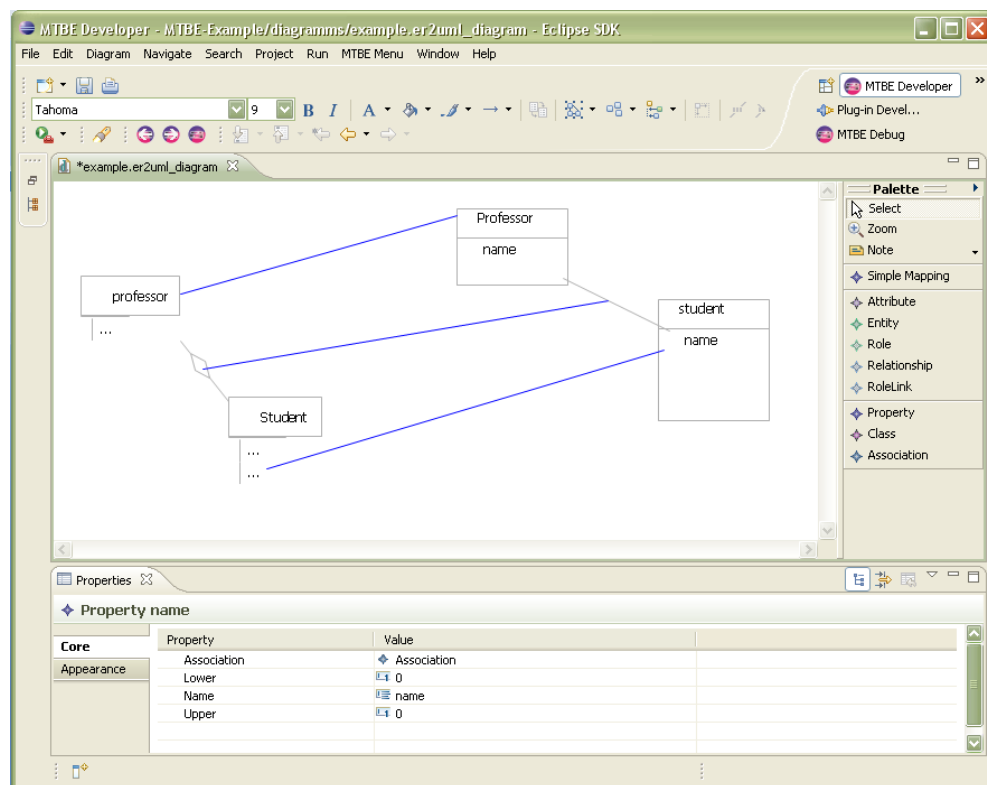


Abbildung 3.7: MTBE Workbench: Screenshots von Ansicht 1

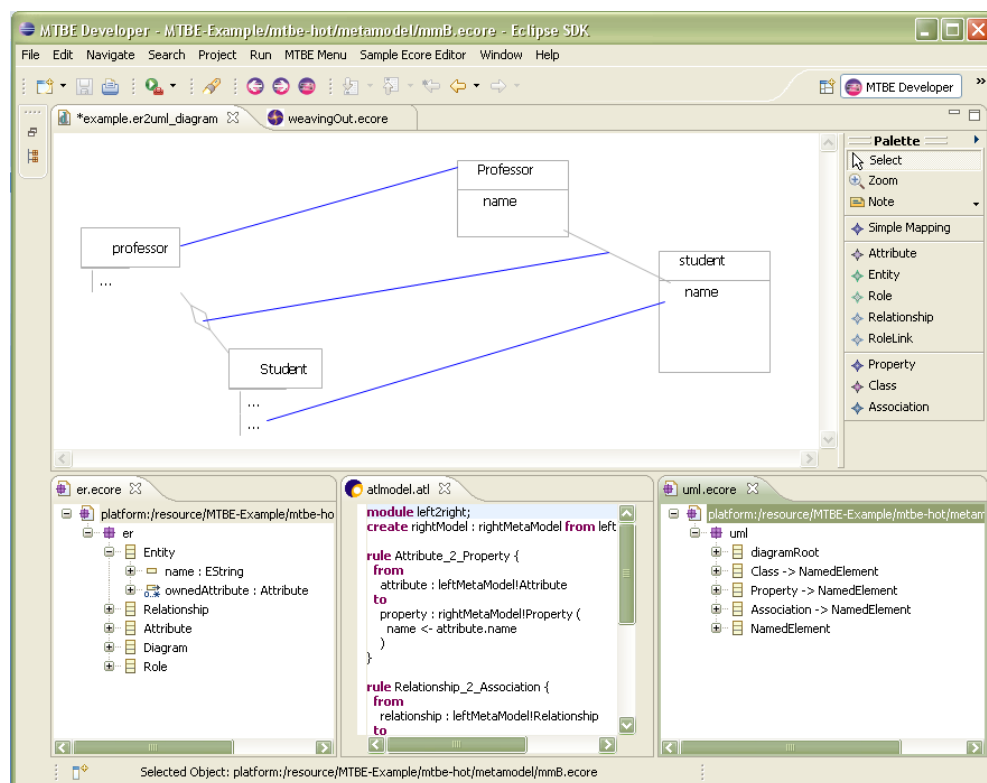


Abbildung 3.8: MTBE Workbench: Screenshots von Ansicht 2

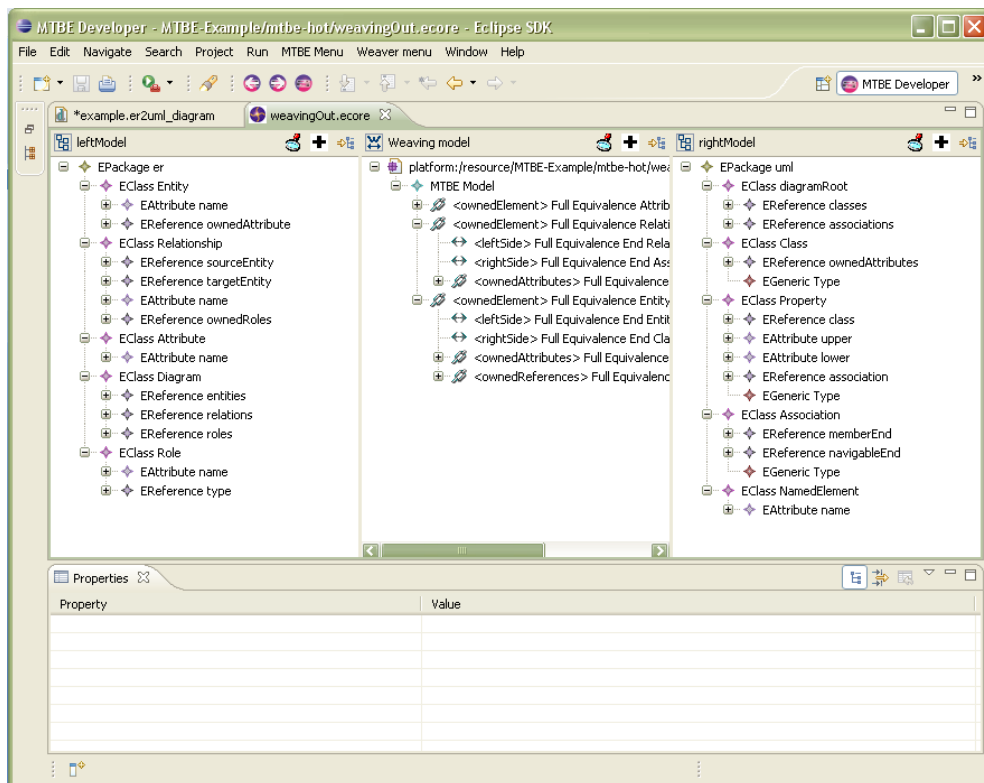


Abbildung 3.9: MTBE Workbench: Screenshots von Ansicht 3

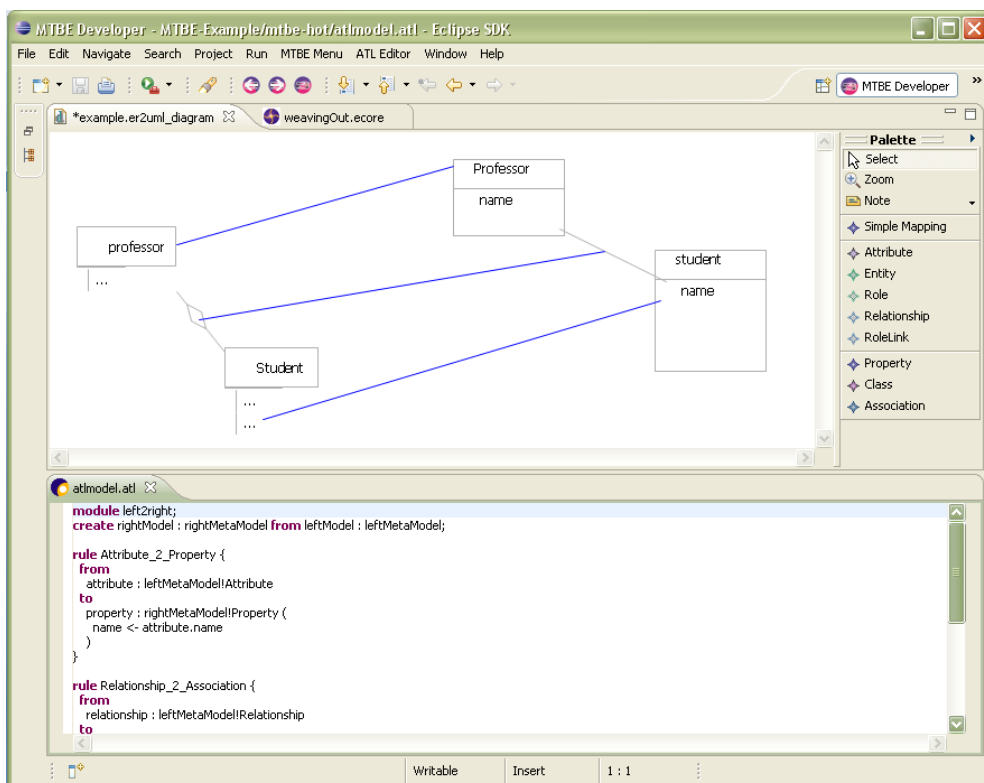


Abbildung 3.10: MTBE Workbench: Screenshots von Ansicht 4

Komponente nimmt die zwei Modelle aus den zwei verschiedenen Modellierungssprachen und liefert eine einzelne Ecore Datei. Wobei jede dieser Sprachekonzepte als eigenes *Package* dargestellt wird.

Aus den zusammengefassten Ecore Dateien kann nun das *EMF Generation Model*, sowie das *EMF Model* und der *Edit Code* generiert werden.

Mapping Definition Merging

Ähnlich wie die beiden Ecore Dateien müssen auch die *GMF Mapping Definition Models* zu einem einzigen Modell zusammengefasst werden. Dabei werden die einzelnen Elemente aus beiden Mapping Modellen in eines zusammenkopiert. Zusätzlich müssen noch die XPath Ausdrücke für die *Domain Model* Elemente der neuen Ecore Datei angepasst werden. Das Ergebnis dieses so genannten *Map Mergers* (siehe Abbildung 3.5) ist das zusammengefasste *GMF Mapping Definition Model* aus welchem im nächsten Schritt der Diagramm Code generiert werden kann.

3.3 MTBE By-Example - UML2ER

Im nächsten Abschnitt wird, zum besseren Verständnis, noch im Detail auf einzelne Punkte des MTBE Prozesses eingegangen. Eine Transformation zwischen den Sprachen UML und ER, dargestellt in Abbildung 3.11, wird als Beispiel herangezogen.

3.3.1 Definition der Mappings

Wie schon beschrieben erstellt der Benutzer im zweiten Schritt des MTBE Prozesses die Mappings zwischen den zwei konkreten Domainmodellen, in Abbildung 3.11 dargestellt durch die *strichlierte* roten Linien, welche Elemente des UML und ER Diagramms verbinden. In diesem Beispiel werden nur ein paar exemplarische Mappings eingezeichnet, um die Übersicht zu wahren. Diese definierten Mappings sind ausschließlich sogenannte *full equivalence* Mappings, also einfache *one-to-one* Mappings.

Zusätzlich zu den Mappings zwischen den konkreten Modellen, die vom Benutzer hinzugefügt wurden, werden als weitere Information die Korrespondenzen zwischen den einzelnen Elementen der konkreten und der abstrakten Syntax benötigt. Diese sind in Abbildung 3.11 durch dünne blau Linien dargestellt. Aus

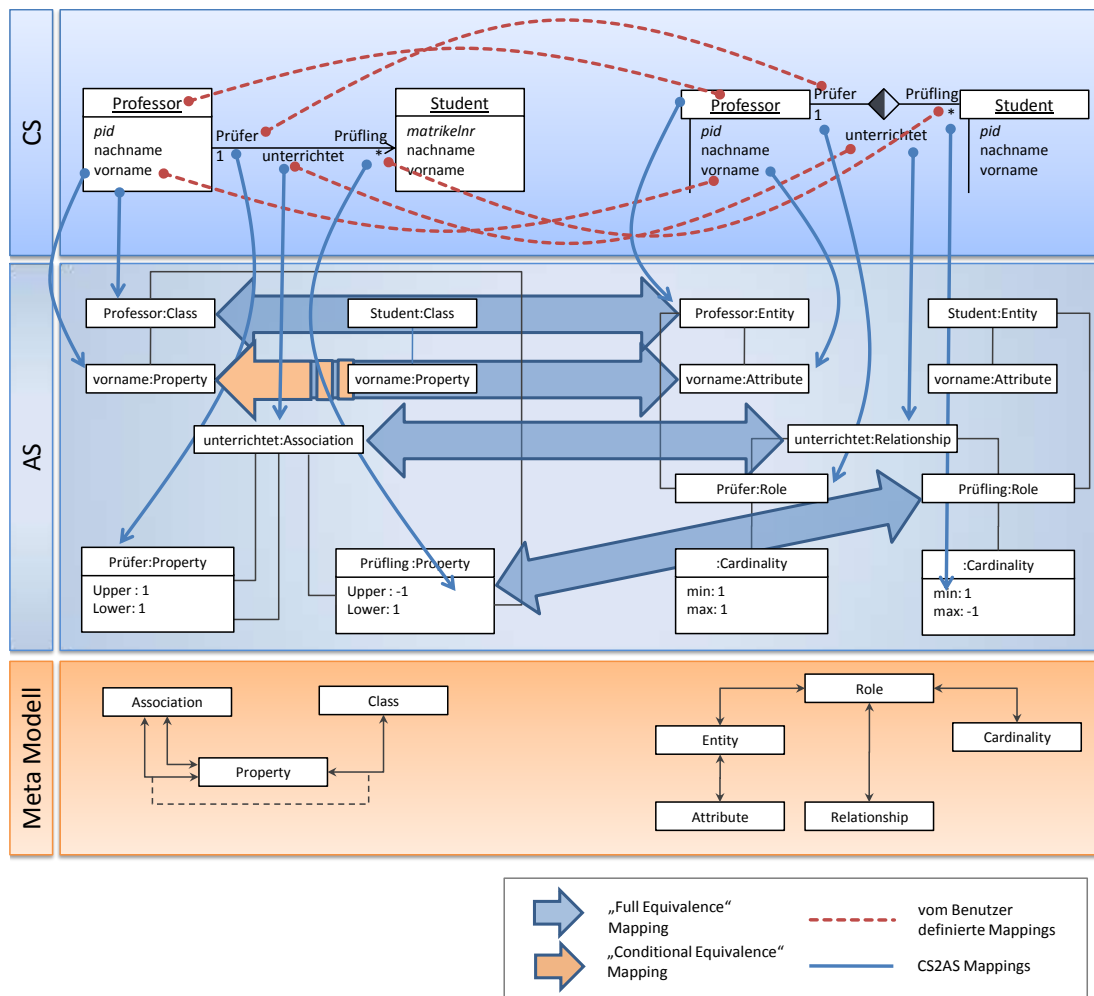


Abbildung 3.11: MTBE für eine UML2ER bzw ER2UML Transformation

Gründen der Übersichtlichkeit wurden wieder einige Verbindungen weggelassen. Die Definition dieser Mappings werden vom Paket *as_2cs* zur Verfügung gestellt.

3.3.2 Generieren der Objekte

Die benutzerdefinierten Mappings werden auf entsprechende Korrespondenzen hin analysiert, dabei wird geschaut welches Element aus dem UML Modell auf das welches im ER Modell mappt. Zum einen werden *Full Equivalence Mapping* Mappings gefunden, Elemente die direkt miteinander Verbunden sind. In Abbildung 3.11 ist das ein Mapping zwischen der Klasse "Professor" und der Entität "Professor". In Kombination mit den äquivalenten Mappings kann der Schluss gezogen werden das auch auf der abstrakten Ebene eine *Full Equivalence Mapping* besteht und es kann daher die entsprechende Transformation Regel generiert werden.

Werden Elemente von einem bestimmten Typ gefunden die zu Elementen ei-

nes anderen Typs “gemappt” werden, spricht man von *Conditional Equivalenz Mappings*. Im Bezug auf das gewählte Beispiel 3.11 wären das die Kombination der Mappings zwischen dem *Property* der *Klasse* “Professor” und dem *Attribute* der *Entität* “Professor” sowie zwischen dem *Property* der zugehörigen *Association* und der *Rolle* der zugehörigen *Relationship*. Da sowohl die *Association* als auch die *Klasse* laut Metamodell ein *Property* zugeordnet haben können. Entsprechend der Bedingungen des *as_2cs* Paketes muss nun entschieden werden welches *Property* zu einem *Attribute* wird und welches nicht. In Kombination mit den äquivalenten Mappings ergibt sich ein Korrespondenz auf der abstrakten Ebene mit den erkannten Bedingungen und es kann die entsprechende Transformation Regel generiert werden.

Wurde diese Analyse auf allen benutzerdefinierten Mappings angewandt, können die Objekte des ER Diagrammes auf Basis des UML Diagrammes generiert werden, beziehungsweise im Fall einer UML zu ER Transformation in die gegengesetzte Richtung.

3.3.3 Setzen der Attributwerte

Aus den im letzten Schritt erstellten Objekten ergibt sich nun der nächste Schritt, das Setzen der Attributwerte für diese Objekte. Hierbei wird zwischen zwei Arten von Attributen unterschieden, den ontologischen und den linguistischen.

- *Ontologische Attribute* - repräsentieren Semantik aus der “Real-World-Domäne”. Beispiel dafür sind *Class.name* oder *Attribute.name*, also “Professor” oder “Vorname”. Laut dem Mapping in diesem Beispiel 3.11 kann der Schluss gezogen werden, dass der *name* der *Klasse* dem *namen* der *Entität* entspricht, da beide denselben Attribut-Wert *Namen* haben.
- *Linguistische Attribute* - hingegen werden zur Vergegenständlichung von Modell Konstrukten, die nicht vom Benutzer selbst gesetzt werden, verwendet. Beispiele dafür wären *Class.isAbstract* oder *Property.aggregation*. Um diese Attribute zu setzen muss auf die Informationen der *as_2cs* zurückgegriffen werden. In Bezugnahme auf die Abbildung 3.11, tritt dieser Fall zum Beispiel bei der Transformation zwischen dem *Attribute* “vorname” und dem *Property* “vorname” auf, denn hier müssen auch die linguistischen Werte wie *Property.aggregation* gesetzt werden.

3.3.4 Verbinden der Objekte

Nun müssen nur mehr die Verbindungen zwischen erstellten Objekten hergestellt werden, dazu wird auf

- das Metamodell,
- die Triples der *as2cs* Mappings mit den OCL Bedingungen,
- den vom Benutzer definierten Mappings
- und bei Bedarf auf zusätzliche Benutzereingaben

zurückgegriffen.

3.3.5 Erzeugen der ATL Regeln

Der letzte Schritt ist das Generieren des ATL Codes, dazu werden die Informationen aus den vorhergehenden Schritten gesammelt und aus ihnen die Transformationsregeln generiert. Das MTBE Konzept sieht vor die Transformationsprache frei zu wählen und beschränkt sich nicht auf ATL. Im beschriebenen Framework wird aus dem generierten *Weaving Modell* ein zum ATL 2006 Metamodell konformes ATL Modell erzeugt, aus welchem dann mittels *AM3 Extractor* der für den Benutzer lesbare ATL Code generiert wird.

Kapitel 4

Implementierung des MTBE Prototyps

4.1 Übersicht

In diesem Kapitel wird beschrieben, wie das von Strommer und Wimmer vorgestellte Framework für ihren *Modelltransformation By-Example* Ansatz umgesetzt wurde. Für die Implementierung wurde der MTBE in drei Hauptblöcke eingeteilt (siehe Abbildung 4.1), dem Erstellen des Projektes, dem Zeichnen der Diagramme mit den benutzerdefinierten Mappings und den abschließenden Prozessen zur Analyse und zur Transformation.

New MTBE Project ... Der erste Block umfasst das Generieren des Diagramm Editors sowie alle nötigen Schritten, um dem Benutzer ein Rahmen zur einfachen Verwendung von MTBE zu bieten. Der Benutzer erstellt ein neues Projekt, in dem er Projektname, Modellnamen und gmf Modelle des Zielmodells sowie des Quellmodells angibt. Das daraus erstellte Eclipse Projekt beinhaltet die geeignete Ordnerstruktur mit allen notwendigen Dateien sowie ein Diagramm-Plug-in kombiniert aus den beiden einzelnen Editoren.

In diesem Block wurden die Komponenten *Mapping Language*, *Ecore Merger*, *Map Merger* des MTBE Frameworks umgesetzt.

New MTBE Diagramm... Der Benutzer kann nun mit Hilfe des Diagramm-Plug-ins Diagramme aus beiden Problemdomänen in einem Editor zeichnen, dazu stehen ihm die gewohnten Paletten aus beiden Diagrammen zur Verfügung, sowie die jeweiligen implementierten Mappings.

Dieser Block entspricht der Komponente *Mapping Editor* des MTBE Frameworks.

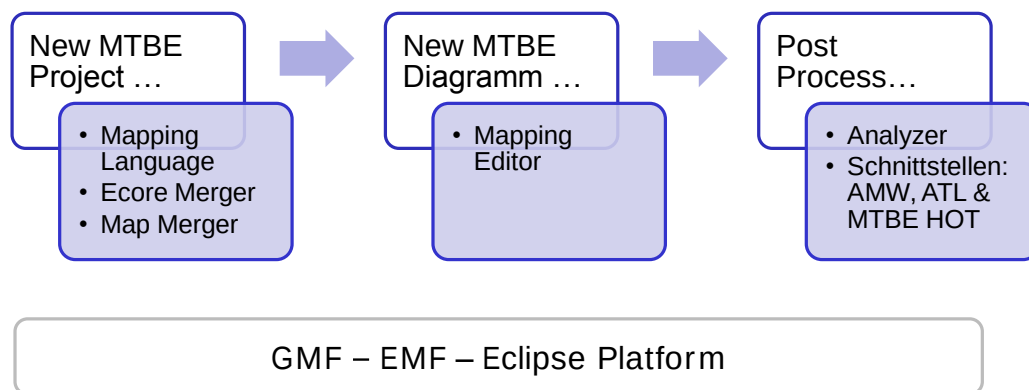


Abbildung 4.1: Überblick MTBE Implementierung

Post process ... Hat man nun die Diagramme gezeichnet, kann man eine Analyse auf diesen ausführen, welche mit ausgewählten Algorithmen das *Weaving Modell* erzeugen. Im Moment stehen als Algorithmen *Simple Reasoning*, *Containment Reasoning* und *Feature Reasoning* zur Verfügung, diese sind aber über eine integrierte Schnittstelle noch beliebig erweiterbar. Als Beispiel für einen über dies Schnittstelle erweiterten Algorithmus wurde der *Feature Mapping* Algorithmus als eigenes Eclipse Plug-in umgesetzt.

In diesem Block wurden die Komponente *Analyzer* des MTBE Frameworks sowie die Schnittstellen zum *Atlas Model Weaver*, zur ATL Transformation und zur MTBE HOT Komponente. Des Weiteren wurde in diesem Block eine Komponente zum Testen der generierten Transformationsregeln erstellt.

Wie von Strommer und Wimmer vorgeschlagen baut der in dieser Arbeit vorgestellte Prototyp auf dem Eclipse Framework auf. Er setzt sich aus den Eclipse Projekten EMF, GEF und GMF, sowie der *ATLAS Model Management Architecture* zusammen.

- Eclipse 3.4M3
- Eclipse Modeling Framework EMF 2.4
- Graphical Editing Framework (GEF) 3.3.1
- Eclipse Graphical Modeling Framework (GMF) 2.0.1
- ATL Bundle 2.0 Standard Version 2.0RC2

Im Folgenden werden nun die einzelnen Komponenten der drei Blöcke detailliert beschrieben, beginnend mit einer Übersicht über die Paketstruktur des Programmes.

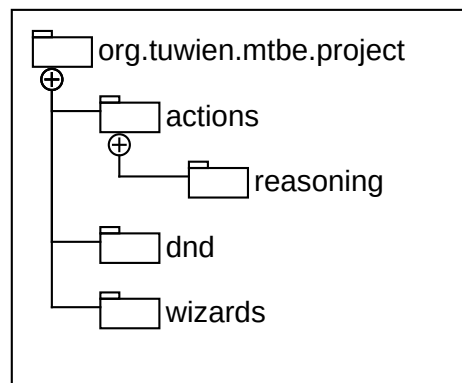


Abbildung 4.2: Paketstruktur

4.2 Systemdokumentation

Die Systemdokumentation umfasst eine Beschreibung der wichtigsten Programmmodule des MTBE Prototyps sowie eine Übersicht über die Paketstruktur des Plug-ins. Neben dem *Projekt Wizard* wurden besonders die drei Komponenten *Merger*, *Analyse*, *Transformation* sowie Aufbau und Funktion der *MTBE-Perspektiven* dokumentiert. Abschließend werden noch die *Extension Points* zur Erweiterung um neue Analysealgorithmen beschrieben.

4.2.1 Paketstruktur

Das MTBE Projekt gliedert sich in die Pakete *actions* und *reasoning* für die MTBE Analyse, das Paket *wizard* mit dem Wizard zum Erstellen eines neuen MTBE Projektes und den Dialogfeldern, sowie dem *dnd* Paket mit den Drag'n'Drop Funktionen welche die Darstellung der Perspektiven unterstützt.

org.tuwien.mtbe.project Dieses Paket beinhaltet neben dem Plug-in-*Activator*, die Klasse für bestehende und neue Analysealgorithmen und -klassen zur Darstellung der Eclipse Perspektiven.

- *Activator* - Standard Klasse eines Eclipse Plug-ins, sie kontrolliert den “life cycle” des Plug-ins.
- *ExtensionReasoning* - Zuständig für die Verwaltung der Analysealgorithmen. Ein Analysealgorithmus hat einen Namen, eine Id und einen Typ, welcher der entsprechenden “Reasoning Methode” entspricht. Weiters enthält diese Klasse eine statische Methode *loadExtensions* welche aus der *ExtensionRegistry* alle registrierten Algorithmen zurückliefert.

- *MTBEPerspective* - Initialisiert die MTBE Debug Perspektive.
- *MTBEPerspectiveDev* - Initialisiert die MTBE Developer Perspektive.

org.tuwien.mtbe.project.actions Enthält neben dem Analyse Prozess und dem Transformationsprozess, weitere wichtige Klassen für diese beiden Prozesse. Außerdem enthält dieses Paket die Klasse “DiagramPlug-in”, welche den neuen Editor generiert und das daraus resultierende Diagramm Plug-in exportiert.

- *EcoreHelper* - Ermöglicht den einfachen Zugriff auf Pakete oder Klassen.
- *ReasoningHelper* - Zur Suche von Mappings innerhalb einer Ressource.
- *MetaMapping* - Diese Klasse stellt die gefundenen Regeln dar.
- *ReasoningAction* - Diese Klasse führt den eigentlichen Analyseprozess durch.
- *TransformationAction* - Führt eine Beispieltransformation durch und zeigt das Beispielmmodell sowie das transformierte Modell an
- *DiagramPlug-in* - Generiert aus den zusammengefassten Editoren den neuen Editor, und exportiert das zugehörige Diagramm-Plug-in

org.tuwien.mtbe.project.actions.reasoning Enthält alle in das MTBE Plug-in eingebetteten Analysealgorithmen, sowie das Interface für einen solchen.

- *Reasoning* - Interface für einen Analysealgorithmen
- *ContainmentReasoning* - Analysealgorithmus für Containment Reasoning
- *SimpleReasoning* - Analysealgorithmus für Simple Reasoning

org.tuwien.mtbe.project.wizards In diesem Paket befinden sich alle Klassen und Methoden die für das Erstellen eines neuen Diagramm-Plug-ins zuständig sind, sowie Dialogfenster und Wizardseiten des MTBE Projektes.

- *MTBEMerger* - Fügt zwei Ausgangsmodelle, in Form von gmfmap-Dateien, zu einem zusammen. Umfasst die Funktionalität des EcoreMerger sowie des GraphMerger.
 - *MTBEPropertyDialog* - Dialogfenster welches vor der Analyse die Auswahl der Algorithmen ermöglicht.
-

- *MTBEPropertyDialog* - Für jedes Diagramm-File können hiermit die zu verwendeten Analysealgorithmen gespeichert werden.
- *MTBETransformationDialog* - Dialogfenster welches vor der Transformation die Auswahl eines Beispielsmodells ermöglicht.
- *NewGraphWizardPage* - Seite aus dem Wizard zur Auswahl der Ausgangsmodelle.
- *NewMTBEProjectWizard* - Wizard zum Erstellen eines neuen MTBE Projekts.
- *NewMTBEProjectWizardPage* - Seite aus dem Wizard zum Erstellen des neuen MTBE Projekts.

org.tuwien.mtbe.project.dnd Die Klassen dieses Pakets unterstützen die Anordnung der Fenster, es simuliert einen *Drag and Drop* Befehl. Der Code wurde aus dem *org.eclipse.ui.tests* übernommen und angepasst.

- *DragOperations* - Schiebt einen ausgewählten Editor oder eine Ansicht zu einer gegebenen Position.
- *EditorDropTarget* - Ziel des Verschiebens des Editor Fensters, leitet sich von *WorkbenchWindowDropTarget* ab.
- *ExistingWindowProvider* - *Window Provider* für *DragOperation*, implementiert *IWorkbenchWindowProvider*.
- *IWorkbenchWindowProvider* - Interface für den *Window Provider*.
- *WorkbenchWindowDropTarget* - Ziel beim Drag'n'Drop eines Fensters, implementiert *TestDropLocation*.

4.2.2 Projekt Wizard

Der MTBE Projekt Wizard erstellt ein neues MTBE-Eclipseprojekt. Dieses Projekt enthält den Quellcode zur Manipulation des kombinierten Metamodells sowie die notwendigen Teile der MTBE-HOT Komponente. Zusätzlich wird durch einen Generator ein weiteres Plug-in Projekt erzeugt, welches den graphischen Editor implementiert. Beide Projekte werden vom Wizard als fertige Plug-ins exportiert, wobei letzteres nach dem Export gelöscht wird. Kombiniert ermöglichen die beiden Plug-ins ein Beispielsmodell zu erstellen und Korrespondenzen zu definieren. Diese Mappings werden in einem späteren Schritt von der Analysekomponente

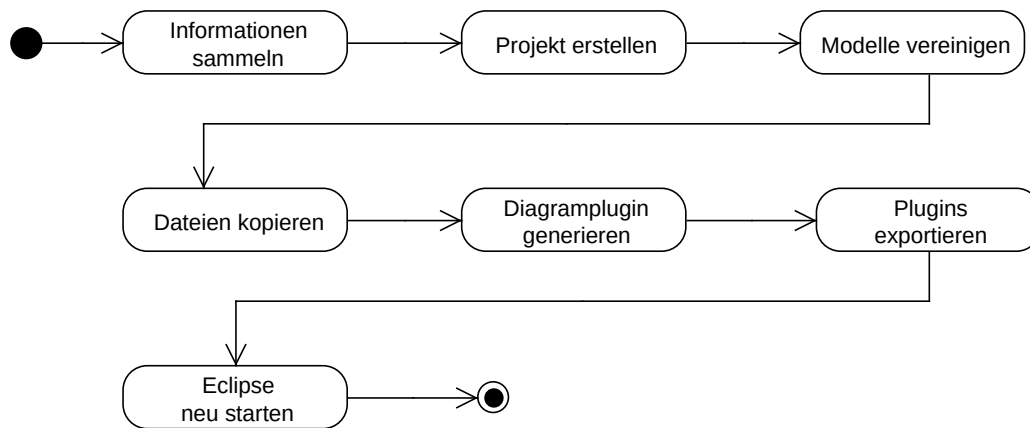


Abbildung 4.3: Arbeitsweise des Wizards

untersucht, um die nötigen ATL-Transformationsregeln zu erstellen. In Abbildung 4.3 ist die Arbeitsweise des Wizards grob skizziert. Die Implementierung ist in den Klassen *NewMtbeProjectWizard*, *NewMtbeProjectWizardPage* und *GraphMerger* des Paketes *org.tuwien.mtbe4.project.wizard* zu finden.

Informationen sammeln Die erste Seite des Wizards benötigt als Input den Namen des zu erstellenden MTBE-Projektes. Dieser ist frei wählbar, darf aber noch nicht als Projektname in dem aktuellen *Workspace* verwendet werden. Die zweite Seite des Wizards enthält Eingabefelder für die *Mapping Definitionen* der beiden Editoren. Diese werden direkt mittels ihres absoluten Pfads angegeben. Alternativ können sie auch durch Verwendung einer der beiden Funktionen *Browse* und *Find in Workspace* ausgewählt werden. Eine Voraussetzung dieser Implementierung ist, dass beide Quell-Editoren vollständig als GMF Projekte umgesetzt sind. Wie aus der Einführung in GMF (Siehe Abschnitt 2.3.1) bekannt ist, besteht ein Editor in GMF aus einem Domänenmodell, einer *Graphical Definition*, einer *Tooling Definition* und einer *Mapping Definition*. Nachdem das letztere Modell die Wurzelemente aller Anderen referenziert, benötigt der Wizard, und später auch die *Merger* Komponente lediglich den Pfad zu diesem Modell. Zusätzlich ist für jede Domäne ein Name anzugeben. Dieser wird verwendet um verschiedenste Ressourcen zu benennen. Auch das Diagramm Plug-in ist später unter dem Namen (*Name des ersten Modells*) 2 (*Name des zweiten Modells*) *Diagram* zu finden. Die genaue Vorgehensweise wie ein Wizard in Eclipse erstellt wird kann verschiedenen Tutorials entnommen werden und soll somit hier nicht näher erläutert werden [22], [35].

Verzeichnis	Inhalt
/src	In diesen Ordner befinden sich die vom Editor generierten Klassen zur Manipulation von Instanzen des gemeinsamen Metamodells. Generiert wird der <i>Model</i> -, der <i>Edit</i> - und der <i>Editorcode</i> . Für das Diagramm wird ein eigenes Plug-In erstellt. Der Quellcode dieses Plug-Ins wird nach dem Export gelöscht.
/diagramms	Dieser Ordner ist für die Beispielmolelle des Benutzers vorgesehen und ist somit nach dem Anlegen eines neuen MTBE-Projektes noch leer.
/model	Dieser Ordner enthält, wie bei GMF Projekten üblich, die Modelle. Dabei handelt es sich um das jeweils zusammengefügte <i>Domain Model</i> , die <i>Graphical Definition</i> , die <i>Tooling Definition</i> , die <i>Mapping Definition</i> und das <i>Generator Model</i> . Falls das automatisierte Zusammenführen der Editoren nicht zum gewünschten Ergebnis kommt, können diese Modelle manuell angepasst, und das Diagramm Plugin neu generiert werden.
/mtbe-hot	Dieser Ordner beinhaltet die Dateien der MTBE-HOT Komponente. Direkt in diesen Ordner wird das <i>Weaving Model</i> abgelegt, das von der Analysekomponente erstellt wird. Zusätzlich befindet sich an dieser Stelle die ATL Ausgabe welches durch ein Ant Skript erzeugt wird und, falls eine der MTBE Perspektiven angezeigt wird, die vom Benutzer bereitgestellte Beispieldatei und das Ergebnis der Vorschautransformation.
/mtbe-hot/hot	Beinhaltet einige Dateien die zum Extrahieren der ATL-Regeln aus dem <i>Weaving Models</i> notwendig sind.
/mtbe-hot/metamodel	Beinhaltet die Metamodelle der Original Editoren, sowie das Metamodell für das <i>Weaving</i> und einige ATL-Metamodelle.
/mtbe-hot/scripts	Beinhaltet alle Ant-Skripte, die für die Transformationen eingesetzt werden.

Tabelle 4.1: Verzeichnisstruktur eines MTBE-Projektes

Projekt erstellen Nachdem alle Informationen korrekt eingegeben wurden, kann der Wizard mit dem Erstellen des neuen MTBE-Projektes fortfahren. Dies beginnt mit dem Aufruf der Methode *performFinish* beim Abschluss des Editors. Der erste Schritt besteht darin das Projekt anzulegen, dessen Klassenpfad zu setzen und die nötige Ordnerstruktur im Projekt aufzubauen. In Tabelle 4.1 wird erläutert wie diese Struktur aussieht und welche Komponenten in welchem Ordner abgelegt werden. Da es sich um ein Java-Projekt handelt, wird dem Projekt

die *JRE System Library* in der Version 1.6 hinzugefügt. Falls in Zukunft neuere Java-Versionen verwendet werden sollten, müsste das an dieser Stelle entsprechend angepasst werden. Die Funktionalität zum Erstellen des Projektes ist in der Methode *doFinishProjectCreation* implementiert, an deren Ende zusätzlich noch das Manifest für dieses Projekt erstellt wird.

Modelle vereinigen Um dem Anwender die Möglichkeit zu bieten Verbindungen zwischen Elementen beider Editoren zu erstellen, muss ein gemeinsamer Editor generiert werden. Dieser muss zusätzlich zu den Elementen dieser Editoren auch die Domainelemente und graphischen Komponenten enthalten, die eine Verbindung ermöglichen. Es müssen alle Modelle die einen GMF Editor ausmachen zusammengeführt und dann die Elemente der Mappings hinzugefügt werden. Dies wird von der *Merger* Komponente durchgeführt deren Arbeitsweise in Abschnitt 4.2.3 erläutert wird.

Dateien kopieren Vor allem für die MTBE-HOT Komponente sind einige Dateien nötig welche im nächsten Schritt in die entsprechenden Ordner des neu angelegten Projektes kopiert werden müssen. Dies geschieht in der Methode *doFinishGraphMerge* direkt nach der Vereinigung der Modelle durch die *Merger*-Komponente. Auch der Wizard ist als Eclipse Plug-in realisiert und somit müssen die notwendigen Dateien aus einem laufenden Plug-in in das neue Projekt kopiert werden. Listing 4.1 zeigt wie so etwas bewerkstelligt werden kann. Im Beispiel wird der Inhalt des *Weaving*-Metamodells in den Ordner *mtbe-hot/metamodel* kopiert.

Listing 4.1: Kopieren von Dateien aus einem laufenden Plug-in

```
1 IWorkspaceRoot root = ResourcesPlug-in.getWorkspace().getRoot();
2 IResource resource = root.findMember(new Path(containerName));
3 IContainer container = (IContainer) resource;
4
5 ...
6
7 IFile file = container.getFile( new
    Path("/mtbe-hot/metamodel/weavingmodel.ecore") );
8 try {
9     InputStream stream = this.getClass().
        getResourceAsStream("/model/weavingmodel.ecore");
10    if (file.exists()) {
11        file.setContents(stream, true, true, monitor);
12    } else {
13        file.create(stream, true, monitor);
14    }
15    stream.close();
16 } catch (Exception e) {
17     ...
18 }
```

Direkt nach dem Zusammenführen der Modelle werden zu allererst die Metamodelle der Originaleditoren kopiert. Diese sind notwendig um später, nach der Analyse dem *Weaving* Modell die *left* und *right* Referenzen auf diese Metamodelle zu setzen. Würde das nicht geschehen, könnten nur Modelle transformiert werden, welche mit dem gemeinsamen Editor erstellt wurden, was sicherlich nicht das Ziel dieses Projektes darstellt. Die beiden Modelle werden im entsprechenden Unterverzeichnis des *hot* Ordners abgelegt. Auch das *Weaving* Metamodell wird in denselben Ordner kopiert. Danach wird das Ant Script, welches die Transformation des *Weaving* Modells zu ATL Regeln und die Transformation zwischen den Modellen ermöglicht im Unterordner *skript* abgelegt. Auch die von diesem Skript benötigten ATL Modelle kommen in die entsprechenden Ordner.

Abschließend werden in den Optionen des Plug-ins die Namen der Metamodelle gespeichert um später darauf zugreifen zu können. Beachtet werden muss hier, dass beim Kopieren davon ausgegangen wird, dass alle GMF Dateien bis auf die Endung denselben Namen besitzen. Des weiteren sei hier darauf hingewiesen, dass die Namen der Metamodelle im laufenden Plug-in abgelegt werden, was dazu führen kann, dass Problemen auftreten können, wenn mehrere unterschiedliche MTBE-Projekte angelegt und analysiert werden.

Diagramm Plug-in generieren Im Schritt *Modelle vereinigen* wurden zwar die nötigen GMF Modelle erstellt und der Code für die Manipulation des Modells generiert, der Quellcode für das Diagramm Plug-in wurde jedoch noch nicht erstellt. Dazu ist es notwendig zuerst ein GMF *Generator Model* zu erstellen. Normalerweise würde dies vom Benutzer durch Auswahl des Punktes *Create GMF Generator Model* im Kontextmenü der *Mapping Definition* erledigt. Zugunsten der Bedienbarkeit haben wir uns entschieden dies programmatisch auszuführen. Leider ist das GMF-Projekt noch sehr jung und daher konstanten Änderungen unterworfen. Die Funktion welche vom erwähnten Menüpunkt ausgeführt wird, ist leider durch Einschränkungen des entsprechenden Plug-ins geschützt. Der Zugriff darauf ist zwar möglich, wird aber nicht empfohlen, da vom GMF-Projekt nicht garantiert werden kann, dass sich diese in zukünftigen Versionen nicht ändern oder in andere Teile verschoben werden. Anstatt diese Funktionalität nachzuprogrammieren, wurden in dieser Arbeit trotz dieser Tatsache die oben beschriebenen Methoden verwendet, um sich auf eine lauffähige Implementierung des gesamten Projektes zu konzentrieren zu können.

Listing 4.2: Erstellen des GMF *Generator Modells*

```

1 URI u;
2
3 try {
4     u = URI.createPlatformResourceURI( container.getFullPath().
        toOSSString()+ "/" + mapName1 + "2" + mapName2 + ".gmfgn", true
        );
5
6     TransformToGenModelOperation transformOperation = new
        TransformToGenModelOperation();
7     transformOperation.setGenURI(u);
8     transformOperation.loadMappingModel(
9         transformationSet,
10        URI.createPlatformResourceURI(
11            container.getFullPath().toString()
12            + "/" + mapName1 + "2" + mapName2 +
13            ".gmfmap", true), null);
14    transformOperation.loadGenModel(
15        transformationSet,
16        URI.createPlatformResourceURI(
17            container.getFullPath().toString()
18            + "/" + mapName1 + "2" + mapName2 + ".genmodel", true), null);
19    transformOperation.executeTransformation(
20        transformationSet, null);
21 } catch (Exception e) {
22     return;
23 }

```

Listing 4.2 zeigt die kritischen Teile des Codes. In Zeile drei wird die URI für das *Generator Model* erstellt. Die gemeinsame *Mapping Definition* und das gemeinsame *Domain Model* dienen als Ausgangsmodelle für die Transformation (Zeile 9 und 15) und in Zeile 20 wird die Überleitung schließlich durchgeführt. Dieser Methode wird als Parameter ein *ResourceSet* übergeben, welches, falls keine Fehler auftreten das fertige *Generator Model* enthält. Im Prinzip könnte dieses schon als Input für den Generator dienen, um eine einheitliche Menüstruktur zu gewährleisten muss aber noch die *Wizard ID* angepasst werden. Dadurch erscheint auch der Wizard, welcher neue MTBE-Diagramme erstellt unter demselben Menüpunkt wie der MTBE-Projektwizard.

Nun kann der Quelltext des Diagrammeditors erzeugt werden. Dabei wird ein neues Plug-in Projekt angelegt, welches lediglich den Code für die graphische Manipulation enthält. Der Quellcode für die Modellmanipulation befindet sich im eben angelegten Projekt.

Plug-ins exportieren Auch dieser Teil wird normalerweise vom Benutzer selbst initiiert indem er im Kontextmenü eines Projektes *Export* wählt. Es ist im Rahmen dieser Arbeit nicht gelungen diesen Schritt völlig transparent zu halten, da wiederum die gleiche, wie schon oben erwähnte Einschränkung bezüglich der Zugriffsberechtigung gilt. Trotzdem werden auch hier diese Methoden verwendet.

Listing 4.3: Exportieren eines Projektes

```

1
2 ImportExportWizard iew = new ImportExportWizard(ImportExportWizard.
    EXPORT);
3
4 // preselect the two projects for export....
5 Vector<IProject> v = new Vector<IProject>();
6 v.add(root.getProject(containerName));
7 v.add(root.getProject(containerName+".diagram"));
8
9 StructuredSelection ss = new StructuredSelection(v);
10 iew.init(workbench, ss);
11
12 IDialogSettings workbenchSettings = WorkbenchPlugin.getDefault().
    getDialogSettings();
13 IDialogSettings wizardSettings = workbenchSettings.getSection("
    ImportExportAction");
14
15 if (wizardSettings == null) {
16     wizardSettings = workbenchSettings.addNewSection("
        ImportExportAction");
17 }
18
19 iew.setDialogSettings(wizardSettings);
20 iew.setForcePreviousAndNextButtons(true);
21
22 Shell parent = workbench.getActiveWorkbenchWindow().getShell();
23
24 WizardDialog dialog = new WizardDialog(parent, iew);
25 dialog.create();
26 dialog.getShell().setSize(Math.max(470, dialog.getShell().getSize().x),
    550);
27 PlatformUI.getWorkbench().getHelpSystem().setHelp(
28     dialog.getShell(), IWorkbenchHelpContextIds.EXPORT_WIZARD);
29 dialog.open();

```

Listing 4.3 veranschaulicht das programmatische Exportieren eines Projektes. Dabei werden in den Zeilen 2 bis 7 die beiden Projekte vorselektiert und danach der *Export Wizard* (siehe Abbildung 4.16) angezeigt. Alle Einstellungen können vom Benutzer belassen werden, trotzdem muss er den Wizard abschließen, da keine Möglichkeit gefunden wurde diesen über das Programm zu steuern. Nach dem erfolgreichen Exportieren wird das vom Generator angelegte Diagramm Plug-in Projekt gelöscht, da der Quellcode dieses Projektes nicht mehr verändert werden muss.

Eclipse neu starten Um die beiden Plug-ins zu laden muss abschließend Eclipse neu gestartet werden. Dies wird dem Benutzer angezeigt. Akzeptiert dieser die Aufforderung wird Eclipse neu gestartet und somit die im vorigen Schritt exportierten Plug-ins geladen. Sobald Eclipse eine Möglichkeit bietet neue Plug-ins dynamisch zu laden sollte dies hier berücksichtigt werden.

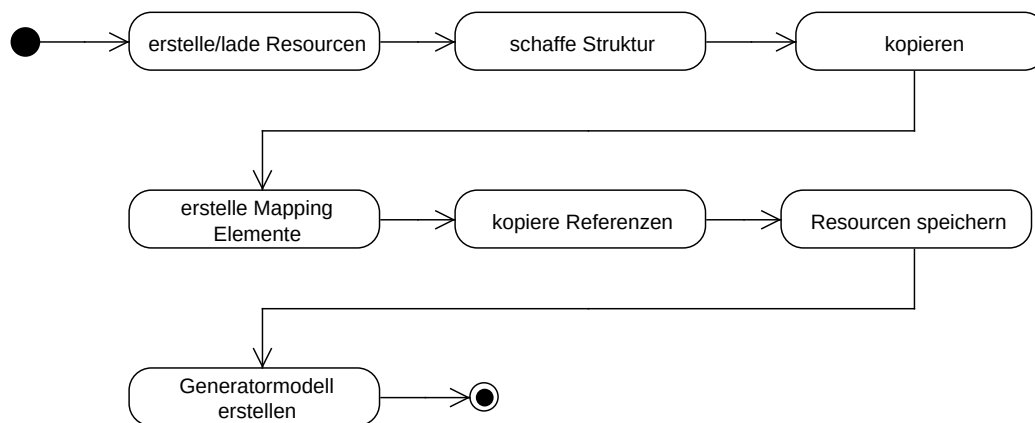


Abbildung 4.4: Übersicht des Zusammenfügens zweier Editoren

4.2.3 Merger

Diese Komponente generiert aus zwei einzelnen GMF-Editoren einen kombinierten Editor welcher die Elemente beider Domänen beherrscht. Dabei wird jeweils eine einzelne Ressource für das Domänenmodell, die *Graphical Definition*, die *Tooling Definition* und die *Mapping Definition* beider Editoren erstellt, welche zusätzlich zu den ursprünglichen Elementen die notwendigen Komponenten zum Erstellen der Mappings enthalten. Dazu müssen die beiden Editoren als GMF Projekte vorliegen und dürfen keine Erweiterungen, wie zum Beispiel *Custom Decoration* Klassen, enthalten. Benötigt werden aber lediglich die Modelle welche die beiden Editoren beschreiben, da nach dem Zusammenfügen der gemeinsame Code mittels der Methoden des Frameworks generiert wird. Somit ist es nicht notwendig den Quellcode der Editoren zu generieren. Ratsam ist es jedoch allemal, denn Fehler in den Editoren finden sich selbstverständlich im gemeinsamen Editor wieder. Abbildung 4.4 veranschaulicht grob die nötigen Schritte, die notwendig sind, um die Modelle zu vereinigen. Erst werden die vorhandenen Ressourcen geladen und diejenigen erstellt die neu hinzukommen. An diesen werden dann Pakete und Klassen angehängt, die die Grundstruktur der Modelle ausmachen. Darauf folgt der eigentliche Kopiervorgang für die Elemente beider Editoren. Im nächsten Schritt werden alle Metamodelle kopiert die nicht direkt in der *Mapping Definition* referenziert sind und schlussendlich werden die erstellten Ressourcen gespeichert.

Anfangs wurde versucht einen generischeren Ansatz zu verfolgen, welcher darin bestand nur von den jeweiligen Metamodellen auszugehen. Dies hätte den Vorteil, dass auch bestehende Editoren verwendet werden könnten, welche nicht in

GMF vorliegen. Im Falle eines nicht auf EMF basierenden Editors müsste man dann lediglich das Metamodell der Domäne nach EMF portieren, was besonders einfach ist wenn dieses schon in MOF definiert ist. Nach einer ersten Implementierung stellte sich aber heraus, dass es einen erheblichen Aufwand darstellt die graphischen Elemente programmatisch zu erstellen. Ein weitaus einschneidender Nachteil eines solchen Ansatzes besteht aber darin, dass es dem Grundgedanken von MTBE widerspricht. Natürlich kann ein Programm anhand des Metamodells, das ja frei definierbar ist, nicht auf die graphischen Notationselemente schließen. Bei wohlbekannten Notationen, wie dies zum Beispiel für die UML Diagramme zutrifft, müssen diese vom Editor generiert werden um den Vorteil des MTBE-Ansatzes auszuspielen, der davon ausgeht, dass der Benutzer mit den jeweiligen Konzepten und Elementen auf der M1 Ebene (siehe auch 2.1.1) vertraut ist. Falls die graphischen Elemente generisch erzeugt werden, weichen diese zwangsläufig von der bekannten Notation ab und müssen somit erst erlernt werden. Nichts desto trotz wäre es ein interessantes Projekt und könnte in Folgeprojekten realisiert werden.

Nachfolgend wird versucht die Arbeitsweise des *Mergers*, welcher in der Klasse *org.tuwien.mtbe4.project.GraphMerger* implementiert ist, bestmöglich darzustellen. Zum Verständnis wird davon ausgegangen dass diese Klasse dem Leser als Quellcode vorliegt. Die einzige *public* Methode dieser Klasse ist *mergeMapFiles* welche vom Wizard aufgerufen wird. Alle anderen Routinen sind Hilfsmethoden und werden falls nötig beim entsprechenden Schritt erläutert.

erstelle/lade Ressourcen Ausgangspunkt sind die vom Wizard gesammelten Informationen. Benötigt werden die beiden *.gmfmap* Dateien, die Bezeichnung der beiden Domänen und der Container in welchem die neu erstellten Dateien gespeichert werden. Da die *Mapping Definition* (enthalten in der Datei mit der Endung *.gmfmap* - wie in Abschnitt 2.3.1 beschrieben) dazu dient die Domäne mit der graphischen Notation zu verbinden, enthält diese Referenzen auf das Domänenmodell, auf die *Graphical Definition* und auf die *Tooling Definition*. Diese müssen daher dem Wizard nicht bekannt gegeben werden, da beim Zugriff auf ein Objekt aus einer anderen Ressource dieses vom *ResourceSet* automatisch nachgeladen wird. Die Methoden am Beginn der Klasse sind für das Laden der Ressourcen zuständig.

In Tabelle 4.2 werden alle im Laufe der Ausführung notwendigen Ressourcen beschrieben, sie werden entweder neu erstellt oder aus einer Serialisierung geladen.

Alle neu erstellten Modelle werden im Verzeichnis */model* des Projektes an-

Variable	neu	Beschreibung	Dateiname
ecoreMerged	x	Wird die Domänenelemente beider Metamodelle enthalten. Zusätzlich enthält es das Metamodell, welches das Mapping beschreibt.	name1+2+name2.ecore
mapMerged	x	Wird die <i>Mapping Definition</i> beider Editoren enthalten. Auch dieser Ressource wird ein <i>Link Mapping</i> hinzugefügt um Elemente beider Editoren zu verbinden.	name1+2+name2.gmfmap
map1		<i>Mapping Definition</i> des ersten Editors	wird nicht kopiert
map2		<i>Mapping Definition</i> des zweiten Editors	wird nicht kopiert
graphMerged	x	Kombinierte <i>Graphical Definition</i> . Für das Mapping wird eine einfache blaue Linie hinzugefügt.	name1+2+name2.gmfgraph
toolMerged	x	Kombinierte <i>Tooling Definition</i> . Enthält je eine <i>Toolgroup</i> für die beiden Domänen und eine für das Mapping	name1+2+name2 .gmftool

Tabelle 4.2: Ressourcen die geladen oder neu erstellt werden müssen

gelegt und können somit nach Belieben angepasst werden. Natürlich muss nach einer manuellen Änderung das Diagramm Plug-in neu generiert und exportiert werden. Dies geschieht wie in Abschnitt 2.3.1 beschrieben.

schaffe Struktur Sind alle Ressourcen geladen oder erstellt, werden die Elemente angehängt, die für das jeweilige Modell notwendig sind und sich nicht aus den vorhandenen Modellen ergeben. Für die *Mapping Definition* ist dies das Wurzelement vom Typ *Mapping* an welches beispielsweise die *Top Node References* angehängt werden. Auch die vorhandenen Modelle besitzen bereits ein solches Wurzelement. Würde aber eines dieser übernommen, hätte dies zur Folge dass auch alle Kindelemente dieses Knotens mit übernommen werden, daher ist es einfacher mittels der *Factory* ein Neues zu generieren als den Baum eines Vorhandenen zu löschen. Darüber hinaus werden die Kindelemente der vorhandenen *Mapping* Wurzelknoten noch benötigt um diese in die neue *Mapping Definition* zu kopieren. Darum werden diese Wurzelknoten in den Variablen *mapping1* und *mapping2* gespeichert um später einfachen Zugriff auf deren Kindelemente zu

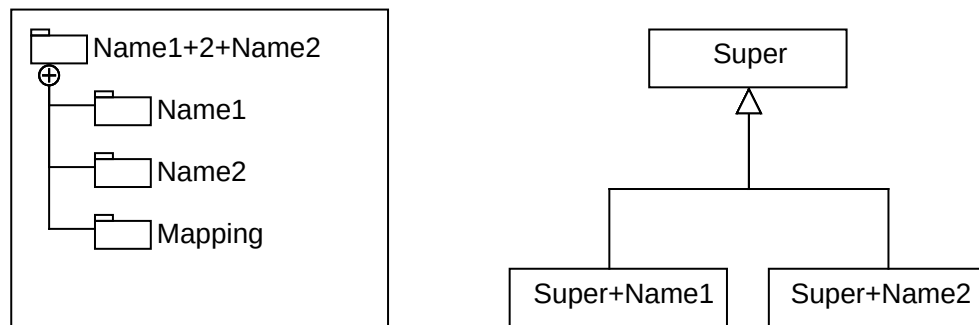


Abbildung 4.5: Pakete und Vererbung im gemeinsamen Metamodell

erlangen. Für die gemeinsame *Graphical Definition* wird eine *Canvas* und eine *Figure Gallery* erstellt, welche die graphischen Elemente beider Editoren beinhalten wird. Da sich die beiden Editoren eine gemeinsame Ressource teilen, muss darauf geachtet werden, dass die einzelnen Editoren nicht dieselben Namen für graphische Elemente verwenden.

Unter einem neu erstellten Wurzelpaket erhält jedes der beiden Metamodelle ein eigenes Paket, welches die Elemente der jeweiligen Domäne enthält. (siehe Abbildung 4.5) Außerdem wird die Klasse *Diagram* erstellt. Dieser werden in einem späteren Schritt Referenzen auf die Elemente hinzugefügt die sonst keinen Container haben. Dies sind diejenigen Elemente die auch bei den vorhandenen Editoren in dem Element der Domäne zu finden sind, welches mittels des *Canvas Mappings* an das Diagramm gebunden ist. Vereinfacht gesagt sind das hauptsächlich jene Knoten, die die Domain Elemente der *Top Node References* beinhalten.

Um bei der späteren Analyse, die vom Benutzer gezeichneten Mappings leichter dem jeweiligen Metamodell zuordnen zu können, werden einige *Super* Klassen erstellt. Dabei gilt, dass jedes Element vom *Super* Element erbt. Dies ist nötig, um bidirektionale Mappings zeichnen zu können, da in GMF bei Referenzen zwischen *Source* und *Target* unterschieden wird. Wird als Typ dieser beiden Referenzen *Super* angegeben, können beliebige Elemente verbunden werden. Dies bringt den Nachteil mit sich, dass auch Elemente innerhalb derselben Domäne verbunden werden können. Diese Implementierung berücksichtigt dies nicht und geht davon aus, dass der Endanwender genügend Erfahrung mitbringt, um beurteilen zu können, dass eine solche Verbindung keinen Sinn machen würde. Zusätzlich erben alle Elemente einer Domäne vom jeweiligen *Super* Element. Dadurch ist es möglich, Elemente sehr einfach dem richtigen Metamodell zuzuordnen.

Listing 4.4: Domain Elemente für das “Simple Mapping”

```

1 EPackage packageMap = EcoreFactory.eINSTANCE.createEPackage();
2 packageMap.setName("mapping");
3 packageMap.setNsPrefix("mapping");
4 packageMap.setNsURI("http://mapping");
5
6 rootPackage.getESubpackages().add(packageMap);
7
8 EClass simpleMapping = EcoreFactory.eINSTANCE.createEClass();
9 simpleMapping.getESuperTypes().add(
10     EcoreFactory.eINSTANCE.createEObject().eClass());
11 simpleMapping.setName("SimpleMapping");
12
13 EReference simpleMapRef1 = EcoreFactory.eINSTANCE.createEReference();
14 simpleMapRef1.setName("part1");
15 simpleMapRef1.setUpperBound(1);
16 simpleMapRef1.setEType(superClass);
17
18 simpleMapping.getEStructuralFeatures().add(simpleMapRef1);
19
20 EReference simpleMapRef2 = EcoreFactory.eINSTANCE.createEReference();
21 simpleMapRef2.setName("part2");
22 simpleMapRef2.setEType(superClass);
23 simpleMapRef2.setUpperBound(1);
24 simpleMapping.getEStructuralFeatures().add(simpleMapRef2);
25
26 packageMap.getEClassifiers().add(simpleMapping);

```

Das dritte Paket beinhaltet die Mapping Elemente. Im Zuge dieser Arbeit wurde lediglich das *Simple Mapping* entwickelt, welches zwei Referenzen vom Typ *Super* besitzt. Somit ist es mittels eines *Simple Mappings* möglich, zwei Elemente zu verbinden. Listing 4.4 veranschaulicht, wie durch die *Reflection API* von EMF alle nötigen Metaelemente des *Simple Mapping* erzeugt werden. Schlussendlich wird noch eine Palette mit drei *ToolGroups* erstellt, die es ermöglichen graphische Elemente zu erstellen und dem Diagramm hinzuzufügen. Erwartungsgemäß existiert für jedes der beiden Domänen eine *Toolgroup*, die die Elemente dieser beinhalten. In der *Toolgroup Name1+2+Name2* beispielsweise befindet sich das Werkzeug zum Erstellen eines Mappings.

kopieren Nun findet das eigentliche Kopieren der vorhandenen Elemente statt. Dabei werden erst alle *Top Node References* des ersten Modells untersucht, dann alle des Zweiten. Die Vorgehensweise ist wie folgt:

1. Jede *Top Node Reference* des Modells wird zu einer *Top Node Reference* im gemeinsamen Modell.
 2. Jede *Top Node Reference* kann ein *Node Mapping* Element besitzen, welches graphische Elemente mit Domain Elementen verbindet. Für dieses *Node*
-

Mapping wird nun die Methode *copyNodeMapping* aufgerufen. Diese erwartet als Parameter die zusammengeführten Modelle, die jeweilige Instanz des *Node Mappings* und die Klasse von der das kopierte *Domain Element* erben soll. Die Vererbung dient, wie schon beschrieben dazu, das Element dem richtigen Metamodell zuordnen zu können.

3. Das Diagrammelement jedes *Node Mappings* wird in die kombinierte *Graphical Definition* übernommen.
4. Das *Domain Element* wird kopiert, sofern dies noch nicht bei einem anderen Aufruf der Methode geschehen ist. Da die *Node Mappings* eine Referenz auf ein Modellelement besitzen, muss der Typ dieser Referenz auf die neue Klasse des gemeinsamen Metamodells gesetzt werden.
5. Gibt es *Compartments* in diesem *Node Mapping* werden auch diese übernommen.
6. Alle *Labels* werden kopiert. Dabei müssen nur die *Featured Label Mappings* auf Domänenelemente überprüft werden, da diese als Einzige solche besitzen können. Die Vorgehensweise ist, wie auch bei allen anderen Elementen die, dass nur kopiert wird, wenn dieses Element nicht schon vorher kopiert worden ist.
7. *Node Mappings* können *Child References* besitzen. Diese beinhalten wiederum ein *Node Mapping* Element, welches selbst wieder *Child References* besitzen kann. Somit muss für alle *Node Mappings* der *Child References* die Methode *copyNodeMapping* rekursiv aufgerufen werden. Dies heißt nichts Anderes, als dass für alle Kindknoten die Schritte 3 bis 9 ausgeführt werden.
8. Sind alle *Node Mappings* und auch die der Kindelemente kopiert, wird noch das Erstellungswerkzeug für den aktuellen Knoten berücksichtigt, falls ein solches vorhanden ist. Für die Prototypimplementierung wird darauf verzichtet weitere *Tools*, wie beispielsweise eine Werkzeugleiste, zu kopieren.
9. Falls ein Domänenelement kopiert wurde, muss dieses von einem der *Super* Elemente erben. Dazu wird dessen Liste von *SuperTypes* der jeweilige *Classifier* hinzugefügt.

Sind alle *Top Node References* und somit alle *Node Mappings* übernommen, werden die *Link Mappings* kopiert. Dieser Vorgang ist etwas einfacher, da auf eine Rekursion verzichtet werden kann. Leider haben in GMF *Link Mappings* kein

vergleichbares Konzept zu den *Child References*. Durch ein solches wäre es einfacher möglich einer Verbindung zusätzliche *Featured Label Mappings* hinzuzufügen und somit auch Multiplizität oder Rollen einfacher darzustellen. Es werden wiederum erst die Links des ersten Editors und dann des Zweiten abgearbeitet.

1. Falls noch nicht im kombinierten Metamodell vorhanden, wird das Domänenelement kopiert.
2. Die graphischen Elemente für diesen Link werden übernommen.
3. Es werden wiederum nur die Erstellungswerkzeuge in die gemeinsame Palette kopiert.
4. Es muss eine *Containment Reference* für diesen Link existieren. Der Typ dieser wird, falls noch nicht erledigt, in das gemeinsame Metamodell kopiert und die Referenz des *Containments* wird angepasst.
5. Links dienen dazu Elemente zu verbinden. Dazu haben sie eine *Source* und ein *Target*. Diese beiden *Domain Elemente* werden gegebenenfalls kopiert und ihr Typ wird auf das neue Element geändert.

erstelle Mapping Elemente Als nächster Schritt wird für die gemeinsame *Mapping Definition* das *Canvas Mapping* erstellt. Dieses verbindet das neue Metamodell mit der gemeinsamen *Graphical Definition* und der gemeinsamen Palette. Damit der Anwender die Verbindungen modellieren kann, wird für das *Simple Mapping* ein *Link Mapping* generiert. Die Elemente werden von den jeweiligen *Factories* erzeugt und in die entsprechenden Modelle eingefügt. Im Metamodell sind die Mappings direkt im Diagramm Element enthalten und können für die Analyse aus diesem als Liste angesprochen werden.

kopiere Referenzen Es ist möglich, dass nicht alle Metamodellelemente einer Domäne von den vorherigen Schritten kopiert wurden. Dies kommt vor, falls keine direkten *Node Mappings* oder *Link Mappings* für diese Elemente existieren, sie aber von anderen Elementen referenziert werden. Daher werden alle kopierten Elemente auf Referenzen untersucht, welche als Typ ein nicht kopiertes *Domain Element* besitzen. Findet sich eine solche, wird dieses kopiert und die entsprechende Referenz angepasst. Da diese Elemente weitere Referenzen besitzen können, wird dieser Schritt solange wiederholt, bis keine neuen Elemente mehr gefunden werden können. Die Implementierung dazu findet sich in der Methode *copyMyReferences*.

Ressourcen speichern Nachdem alle Elemente der unterschiedlichen Modelle in jeweils eine gemeinsame Ressource kopiert wurden, müssen diese Ressourcen noch serialisiert werden. Dies geschieht durch den Methodenaufruf *save* der einzelnen Ressourcen. Nachdem keine zusätzlichen Informationen zur Serialisierung notwendig sind, wird dieser Methode eine leere *Map* übergeben. Wären zusätzliche Informationen gewünscht, wie dies zum Beispiel beim Speichern des *Weaving Modells* notwendig ist, könnten diese in einer entsprechenden *Map* übergeben werden.

Generatormodel erstellen Im Laufe der vorigen Schritte wurde zwar ein gemeinsames Metamodell erstellt, um aber Code zu generieren wird zusätzlich ein Generatormodell benötigt. Dieses dient dazu den Generator zu steuern. Die Methode *writeGenModel* erstellt ein Solches. Normalerweise würde dies mittels eines EMF Wizards erledigt, in Listing 4.5 wird gezeigt wie dies programmatisch erreicht werden kann.

Listing 4.5: Programmatisch Erstellung eines Generatormodells

```

1 //get the root package of our merged metamodel
2 EPackage rootPackage = (EPackage) ecoreMerged.getContents().get(0);
3 //create the resource for the .genmodel file
4 URI genModelURI = URI.createPlatformResourceURI (container.getFullPath
    ().toString() + "/model/" + name2name + ".genmodel", true);
5 Resource genModelResource = Resource.Factory.Registry.INSTANCE.
    getFactory (genModelURI).createResource(genModelURI);
6 //let the Factory produce a Genmodel Object which is the root element
    for the resource and add it to the resource.
7 GenModel genModel = GenModelFactory.eINSTANCE.createGenModel();
8 genModelResource.getContents().add(genModel);
9 resourceSet.getResources().add(genModelResource);
10 //tell the generator where to put the code (everything is in the same
    plugin and under the src folder) and set the right name for the
    plugin
11 genModel.setModelDirectory("/") + container.getProject().getName() + "/"
    src");
12 genModel.setEditDirectory("/") + container.getProject().getName() + "/"
    src");
13 genModel.setEditorDirectory("/") + container.getProject().getName() + "/"
    src");
14 genModel.setModelPlug-inID(container.getProject().getName());
15 org.eclipse.emf.codegen.ecore.Generator.setSD0Defaults(genModel);
16 //the foreign model will be our domain model
17 genModel.getForeignModel().add(ecoreModelURI.toString());
18 genModel.initialize(Collections.singleton(rootPackage));
19 //name1 + name2 is the name of it
20 GenPackage genPackage = (GenPackage) genModel.getGenPackages().get(0);
21 genPackage.setPrefix(name2name);
22 genModel.setModelName(genModelURI.trimFileExtension().lastSegment());
23 genModel.setCanGenerate(true);
24 try {
25     //save the .genmodel

```

```

26     genModelResource.save(Collections.EMPTY_MAP);
27 } catch (IOException e) {
28     e.printStackTrace();
29 }
30 // use the generator the produce the code
31 Generator g = new Generator();
32 g.setInput(genModel);
33 //the model code
34 g.generate(genModel, GenBaseGeneratorAdapter.MODEL_PROJECT_TYPE, new
    BasicMonitor.Printing(System.out));
35 //the edit code
36 g.generate(genModel, GenBaseGeneratorAdapter.EDIT_PROJECT_TYPE, new
    BasicMonitor.Printing(System.out));
37 //and the editor code
38 g.generate(genModel, GenBaseGeneratorAdapter.EDITOR_PROJECT_TYPE, new
    BasicMonitor.Printing(System.out));

```

In den Zeilen 13 bis 15 wird festgelegt, dass der gesamte Quellcode für die Modellmanipulation im *src* Ordner des Projektes abgelegt werden soll. Standardmäßig würden drei verschiedene Plug-ins generiert. In den Zeilen 33 bis 40 wird ein Generator instanziiert und der *Model*, der *Edit* und der *Editor* Code generiert. Dieser Teil ist sicherlich der, der während der Ausführung des Wizards die meiste Zeit in Anspruch nimmt.

4.2.4 Analyse

Dieses Modul analysiert alle vom Benutzer erstellten Mappings und versucht daraus Regeln auf Metamodellebene abzuleiten. Da im Zuge dieser Arbeit vorerst nur ein Prototyp des Frameworks implementiert worden ist, wurde der eigentlichen Analyse noch nicht all zu viel Aufmerksamkeit geschenkt. Sicherlich stellt dieser Teil einen der interessantesten, aber auch komplexesten Aspekte des MTBE-Ansatzes dar. Bei der Implementierung wurde darauf geachtet, besonders diesen Teil erweiterbar zu gestalten, um es Nachfolgeprojekten zu ermöglichen für verschiedene Domänen spezialisierte Algorithmen zu entwickeln und dem Projekt hinzuzufügen. Wie diese Erweiterung bewerkstelligt werden kann, wird in einem späteren Abschnitt beschrieben.

Abbildung 4.6 gibt einen Überblick über die Arbeitsweise der *Analyser* Komponente. Sie ist hauptsächlich in der Klasse *org.tuwien.mtbe4.project.actions.ReasonAction* implementiert und sollte dem Leser wiederum vorliegen, um der nachfolgenden Beschreibung folgen zu können. Angestoßen wird der Prozess vom Benutzer durch die Auswahl *do reasoning* im MTBE-Menü, oder durch Betätigung des entsprechenden Buttons in der Werkzeugleiste. Zur Unterstützung der Algorithmen wurde eine Hilfsklasse

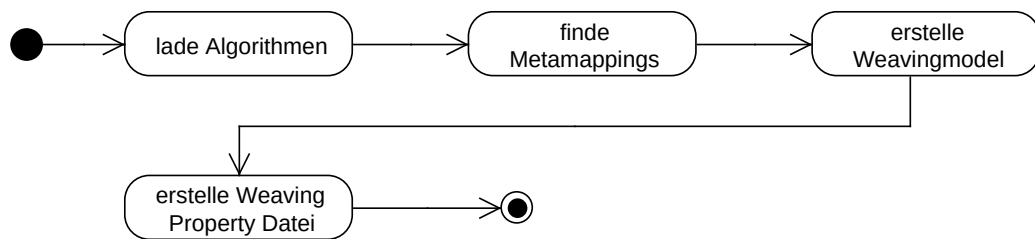


Abbildung 4.6: Arbeitsweise der Analyse

implementiert, mit deren Hilfe es möglich ist, Mappings innerhalb einer Resource zu finden. Eine zweite Hilfsklasse *EcoreHelper*, ermöglicht den einfachen Zugriff auf Pakete oder Klassen einer Ressource.

Lade Algorithmen Der Analyseprozess wurde in Algorithmen aufgeteilt. Dabei gilt, dass jeder Algorithmus einen bestimmten Zweck erfüllt und möglichst alle Mappings, die ein gewisses Kriterium erfüllen, finden soll. Dadurch ist es auch möglich eigene Analysealgorithmen zu entwickeln, die auf eine gewisse Domäne spezialisiert sind und somit das Ergebnis verbessern können. Der Gedanke ist also der, dass nicht eine komplexe Analyse aller Mappings stattfindet, sondern diese Aufgabe auf mehrere spezialisierte Routinen aufgeteilt wird. Da der Benutzer die Möglichkeit erhält zu wählen, welche dieser Algorithmen ausgeführt werden sollen, kann die benötigte Zeit minimal gehalten werden. Es werden nur die Algorithmen ausgeführt, welche auch die Aussicht haben, in einer gewissen Domäne Mappings zu finden. Der erste Schritt der Analyse besteht darin, das betreffende Modell zu finden. Dabei sind für diesen Schritt nur die Modellelemente interessant, die graphische Repräsentation wird nicht analysiert. Um diese zu finden, wird davon ausgegangen, dass der Benutzer entweder diese Datei direkt geöffnet hat oder, was wahrscheinlicher ist, das Diagramm, welches dieses Modell graphisch repräsentiert. Zusätzlich muss die Ressource gespeichert worden sein, damit das Modell auch alle Elemente enthält. Treffen diese Annahmen nicht zu, kann die Analyse nicht durchgeführt werden. Ist das Modell gefunden, wird mit diesem die *Helper* Klasse instanziiert und ein Dialogfenster geöffnet, welches es dem Benutzer ermöglicht, die gewünschten Algorithmen zu wählen. Dabei werden nicht nur die drei, im Zuge dieser Implementierung entwickelten Algorithmen angezeigt, sondern auch diejenigen welche über einen dafür vorgesehenen *Extension point* registriert sind.

Finde Metamappings Ziel des MTBE Ansatzes ist es, von Beispielmappings des Benutzers Rückschlüsse auf Transformationsregeln der Metamodellelemente zu ziehen. Daher werden in diesem Schritt diese Mappings untersucht. Es wird versucht diese Metamappings zu finden und in einem *Weaving Modell* darzustellen. Dieses *Weaving Modell* kann weiterverwendet werden, um daraus ATL Regeln zu extrahieren. Jedem Algorithmus wird eine Liste mit den Bentzermappings, sowie eine Liste mit den bereits von anderen Routinen gefundenen Mappings übergeben. Findet dieser neue Regeln, werden diese der Liste angefügt. Beispielsweise findet das weiter unten erläuterte *SimpleReasoning* Mappings, die ohne Einschränkungen zutreffen. Dies ist dann der Fall, wenn ein bestimmter Typ der ersten Domäne immer mit demselben Typ der zweiten Domäne verbunden wurde.

Listing 4.6: Durchführen der Analyse durch einzelne Algorithmen

```

1    //now go through all the algorithms and if the user selected them,
      we call its reasoning method which will return a List of
      MetaMappings which this reasoning algorithm has found
2    for (ExtensionReasoning er : rs) {
3        try {
4            resoningSelected = Boolean.valueOf(file.
              getPersistentProperty(
5                new QualifiedName("", er.getReasoningId())).toString())
              ;
6        } catch (CoreException e) {
7            e.printStackTrace();
8        }
9        if (resoningSelected) {
10           Reasoning r = er.getReasoning();
11           Vector<MetaMapping> roundMappings = r.doReasoning(model,
              helper, found);
12           metaMappings.addAll(roundMappings);
13       }
14   }
```

Listing 4.6 zeigt den Ausschnitt, welcher nur die vom Benutzer ausgewählten Algorithmen durchführt und dem Vector *metaMappings* alle, von dieser Klasse gefundenen Regeln, hinzufügt. Die gefundenen Regeln werden von der Klasse *MetaMapping* dargestellt. Diese beinhaltet die beiden Klassen des Metamodells und den Typ dieses Mappings. Wie im Abschnitt 5 beschrieben, besteht momentan noch keine Unterstützung für komplexe Bedingungen, also Transformationsregeln, welche nur unter Einschränkungen gelten. Um aber den *ContainmentReasoning* Algorithmus zu implementieren zu können, findet sich im *MetaMapping* auch das *Containment*-Element, sowie für das *FeatureReasoning* zwei Listen von *Features*, welche, wenn gesetzt, den Unterschied zwischen Mappings darstellen. Es sei aber darauf hingewiesen, dass dies keine endgültige Lösung darstellt, sondern vielmehr von einem generellen Ansatz, beispielsweise durch OCL Unterstützung, ersetzt werden sollte.

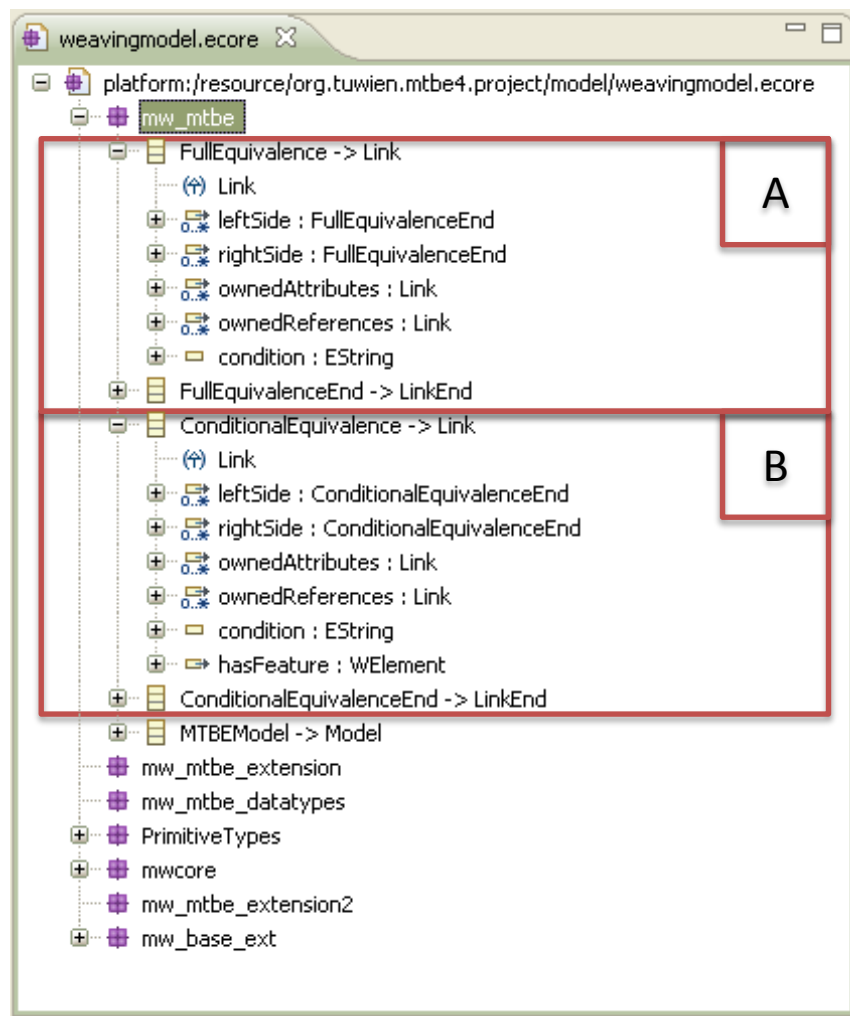


Abbildung 4.7: Das Weaving Meta Model für MTBE

erstelle Weaving Modell Nachdem die einzelnen Algorithmen die entsprechenden Mappings auf Metamodellebene gefunden haben, kann aus dieser Information ein *Weaving Modell* generiert werden, welches wiederum als Input zum Erstellen der ATL Regeln dient. Das *Weaving Modell* muss zu dessen Metamodell konform sein (Abbildung 4.7). Daher wurde dieses Metamodell durch den Wizard (Siehe Abbildung 4.2.2) in den Ordner `/mtbe-hot/metamodel/weavingmodel.ecore` des Projektes kopiert. Von dort wird es als Ressource geladen.

Das ursprüngliche *Weaving Modell* wurde für MTBE geringfügig erweitert. Das Metamodell enthält die Definition der Mappings für die *FullEquivalence* (Abbildung 4.7-A) und *ConditionalEquivalence* (Abbildung 4.7-B). Durch die Klasse *EcoreHelper* werden die nötigen Metamodellelemente, welche im generierten *Weaving Modell* instanziiert werden, geladen. Danach wird die Resource `/mtbe-hot/weavingOut.ecore` angelegt, welche dieses *Weaving Modell* enthalten wird.

Listing 4.7: Setzen der *left* und *right* Referenz des *Weaving Modell*

```

1      EClass modelRefMeta = eHelper.getClass("ModelRef", baseExt);
2      EObject leftModelRefInst = baseExt.getEFactoryInstance().create
        (modelRefMeta);
3      modelInst.eSet(modelMeta.getEStructuralFeature("leftModel"),
        leftModelRefInst);
4      leftModelRefInst.eSet(modelRefMeta.getEStructuralFeature("ref")
        , "/mtbe-hot/metamodel/mmA.ecore");
5      leftModelRefInst.eSet(modelRefMeta.getEStructuralFeature("name"
        ), "leftModel");
6      EObject rightModelRefInst = baseExt.getEFactoryInstance().
        create(modelRefMeta);
7      modelInst.eSet(modelMeta.getEStructuralFeature("rightModel"),
        rightModelRefInst);
8      rightModelRefInst.eSet(modelRefMeta.getEStructuralFeature("ref"
        ), "/mtbe-hot/metamodel/mmB.ecore");
9      rightModelRefInst.eSet(modelRefMeta.getEStructuralFeature("name"
        ), "rightModel");

```

Nach dem obligatorischen Wurzelement werden die Referenzen auf die ursprünglichen Metamodelle als *left* und *right* Referenzen erstellt. In Listing 4.7 ist zu sehen, wie dies erreicht werden kann. Zuerst wird in Zeile 1 die Metaklasse für Modellreferenzen aus dem Metamodell gesucht. Diese wird in Zeile 2 dazu verwendet um eine Instanz davon zu erzeugen. In den Zeilen 3 bis 5 werden über die Reflektion API von EMF die nötigen Attribute *ref*, *name* und *leftModel* gesetzt. Die *left* Referenz verweist dann auf das ursprüngliche Metamodell des ersten Editors. In den Zeilen 6 bis 9 wird die Referenz *rightModel* auf das zweite Metamodell gesetzt. Bevor nun begonnen wird alle gefundenen Mappings abzuarbeiten, werden die Metamodellelemente für die benötigten Weavingelemente ebenfalls entsprechenden Variablen zugeordnet. Nun muss für jedes gefundene Mapping ein *Weavingelement* erstellt werden, welches die Elemente des linken Metamodells mit denen des Rechten verbindet.

Wie bereits erwähnt, existiert für diese Prototypimplementierung noch keine ausgereifte Möglichkeit Bedingungen wie OCL zu modellieren obwohl das *ContainmentReasoning* und das *FeatureReasoning* solche bereits erkennt. Diese einfachen Bedingungen können aber glücklicherweise durch Schachtelung im *Weaving Modell* abgebildet werden.

Listing 4.8: Erstellen einer "Full Equivalence"

```

1      EClass fullEquivalence = eHelper.getClass("FullEquivalence",
        fullCond);
2      EClass fullEquivalenceEnd = eHelper.getClass("FullEquivalenceEnd",
        fullCond);
3      EClass elementRefMeta = eHelper.getClass("ElementRef", baseExt);
4      EObject eo = fullCond.getEFactoryInstance().create(fullEquivalence)
        ;
5      EAttribute nameAttr = eHelper.getAttribute("name", fullEquivalence)
        ;

```

```

6      eo.eSet(nameAttr, mm.getFrom().getName() + "2" + mm.getTo().getName
      ());
7      // the left side reference in the weaving model
8      EStructuralFeature leftSideFeature = fullEquivalence.
      getESTructuralFeature("leftSide");
9      EAttribute leftSideNameAttr = eHelper.getAttribute("name",
      fullEquivalenceEnd);
10     EObject leftSide = fullCond.getEFactoryInstance().create(
      fullEquivalenceEnd);
11     leftSide.eSet(leftSideNameAttr, mm.getFrom().getName());
12     EStructuralFeature leftSideLinkReference = fullEquivalenceEnd.
      getESTructuralFeature("linkLeftSide");
13     EObject leftSideLinkObject = baseExt.getEFactoryInstance().create(
      elementRefMeta);
14     leftSideLinkObject.eSet(elementRefMeta.getESTructuralFeature("name"
      ), mm.getFrom().getName());
15     leftSideLinkObject.eSet(elementRefMeta.getESTructuralFeature("ref")
      , coreUtil.getURI(mm.getFrom()).fragment());
16     leftSide.eSet(fullEquivalenceEnd.getESTructuralFeature("element"),
      leftSideLinkObject);
17     ((List) eo.eGet(leftSideFeature)).add(leftSide);
18     ((List) leftModelRefInst.eGet(modelRefMeta.getESTructuralFeature("
      ownedElementRef"))).add(leftSideLinkObject);
19     // the left side reference in the weaving model
20     ...

```

In Listing 4.8 wird gezeigt, wie eine Seite eines *Weavinglements* erstellt wird. Nachdem die notwendigen Klassen des Metamodells gefunden sind, wird eine *FullEquivalence* instanziiert und deren Name auf die Bezeichnungen der jeweiligen Klassen aus den Metamodellen gesetzt. Dieser Name dient der Übersichtlichkeit im *Weaving Modell*, da der Benutzer dadurch recht einfach sehen kann, welche Metaklassen durch dieses Element verbunden sind. Die restlichen Zeilen des Listings zeigen wie die linke Seite der *FullEquivalence* auf das entsprechende Element des linken Metamodells gesetzt wird. Dazu muss wieder eine Annahme gelten, nämlich die, dass die Namen der Metamodellelemente durch und nach dem Zusammenfügen nicht geändert wurden, was bei eventuellen Nachfolgeprojekten zu berücksichtigen ist. Ist dies nicht der Fall, können diese nicht mehr in den ursprünglichen Metamodellen gefunden werden und somit könnten die Referenzen zwar Domainelemente des gemeinsamen Metamodells, nicht aber die der ursprünglichen Editoren verbinden.

Equivalent zur eben besprochenen Vorgehensweise werden auch die Elemente des rechten Metameodells (*rightSide*) gesetzt.

Listing 4.9: Erstellen einer *Full Equivalence* als *owned Reference*

```

1      EObject eo = createFullEquivalence(eHelper, fullCond, baseExt,
      mm, modelMeta, modelInst, leftModelRefInst,
      rightModelRefInst, modelRefMeta);
2      List ownedAttrInst = (List) modelInst.eGet(ownedElement);
3      ownedAttrInst.add(eo);

```

```

4      EClass sourceElement = mm.getFrom();
5
6      //try to find the feature on the other side if possible...
7      for (EReference eref : sourceElement.getEAllReferences()) {
8          try {
9              EClass targetMeta1 = (EClass) eref.getEType();
10             //now check if there is a full equivalence for this type
11             so we know what element the other side references
12             Vector<MetaMapping> mmRefs = helper.
13                 getAllMetaMappingsFrom(targetMeta1, metaMappings,
14                 MetaMapping.MAPPING_MATCH_ALWAYS);
15             if (mmRefs.size() > 0) {
16                 EClass targetMeta2 = mmRefs.elementAt(0).getTo();
17                 //this ist the type we are looking for in the
18                 references of the other Class
19                 Vector<EReference> references = eHelper.
20                     getReferencesOfType(targetMeta2, mm.getTo());
21                 if (references.size() == 1) {
22                     //if there is more then one reference of this
23                     type it cannot be determined
24                     //to which of those this feature is mapped.
25                     EObject eoc = createFullEquivalenceFromFeature(
26                         eHelper, fullCond, baseExt, mm.getFrom(),
27                         mm.getTo(), eref, references.elementAt(0),
28                         modelMeta, modelInst, leftModelRefInst,
29                         rightModelRefInst, modelRefMeta);
30                     List ownedReferencesInst = (List) eo.eGet(
31                         ownedReferences);
32                     ownedReferencesInst.add(eoc);
33                 }
34             }
35         } catch (ClassCastException cce) {
36             //try the next one
37         }
38     }

```

Nachdem ein *Weaving*-Element erstellt wurde, müssen auch die Referenzen des linken Metamodellelementes auf Referenzen des rechten Metamodells abgebildet werden. Listing 4.9 zeigt, wie dies implementiert wurde. Auch hier findet sich wieder Verbesserungspotential, es ist sogar anzunehmen, dass für Referenzen und Attribute eigene Algorithmen nötig sein werden. Für den Prototyp wurde auf eine vollständige Implementierung verzichtet. In einer Schleife (Zeile 6) werden alle Referenzen des linken Domänenelementes durchlaufen. Falls sich ein Mapping für den Typ dieser Referenz findet, welches ohne Einschränkung gilt (Zeile 10), und der Typ dieses Mappings nur in einer Referenz des rechten Domänenelementes zu finden ist (Zeile 15), wird angenommen dass diese Referenzen dieselbe Bedeutung haben und ein entsprechendes Kindelement an das *Weaving* angehängt (Zeile 18 bis 20) wird. Wie zu sehen, werden durch diese Vorgehensweise wahrscheinlich nicht viele Referenzen gefunden und das resultierende *Weaving Modell* muss vom Benutzer noch angepasst werden. Auch bei den Attributen wurde für

diese Implementierung noch eine sehr einfache Regel implementiert. Nur wenn in beiden Domänenelementen ein Attribut mit dem gleichen Namen existiert, wird eine Verbindung erstellt. Da vor allem Attribute vom Typ *String* als *Labels* in GMF modelliert werden, und diese nicht über ein *SimpleMapping* verbunden werden können, fehlt hier noch eine Lösung, die eine bessere automatisierte Analyse ermöglicht. (Siehe Abschnitt 5)

Sollen komplexere Bedingungen abgebildet werden, muss eine *ConditionalEquivalence* instanziiert werden und dieser, zusätzlich zu den Elementen, die entsprechende Bedingung angehängt werden.

Listing 4.10: Erstellen einer “Conditional Equivalence”

```

1      EObject eoKid = createConditionalEquivalence(eHelper, fullCond,
           baseExt, mmKid, modelMeta, modelInst, leftModelRefInst,
           rightModelRefInst, modelRefMeta);
2      List ownedReferencesInst = (List) eo.eGet(ownedReferences);
3
4      //set the has feature, only the first one is set, since the weaving
           metamodel does not allow differences in more than one feature
           yet
5      if (mm.getSetFeatures() != null && mm.getSetFeatures().size() > 0
           && mm.getSetFeatures().elementAt(0) != null) {
6          EClass linkEndMeta = eHelper.getClass("LinkEnd", baseExt);
7          EObject linkEndInst = baseExt.getEFactoryInstance().create(
           linkEndMeta);
8          EObject elementRefInst = baseExt.getEFactoryInstance().create(
           elementRefMeta);
9          elementRefInst.eSet(elementRefMeta.getEStructuralFeature("ref")
           , "/" + mm.getFrom().getName() + "/" + mm.getSetFeatures()
           .elementAt(0).getName());
10         linkEndInst.eSet(linkEndMeta.getEStructuralFeature("name"), mm.
           getSetFeatures().elementAt(0).getName());
11         linkEndInst.eSet(linkEndMeta.getEStructuralFeature("element"),
           elementRefInst);
12         //since the link end does not contain the references to the
           models we have to add it to the left model element.
13         List ownedRefs = (List) leftModelRefInst.eGet(modelRefMeta.
           getEStructuralFeature("ownedElementRef"));
14         ownedRefs.add(elementRefInst);
15         //now set the feature to this reference...
16         eo.eSet(condEquivalenceHasFeature, linkEndInst);
17     }
18     ownedReferencesInst.add(eoKid);

```

In Listing 4.10 wird ein solches erzeugt. Dazu dient wieder eine Hilfsmethode, die das eigentliche Element erzeugt. Dabei handelt es sich um die gleiche Vorgehensweise wie bei dem *FullEquivalence* Mapping, mit dem Unterschied, dass die Elemente ein anderes Metamodellelement instanziiieren. Diese haben aber dieselben Eigenschaften wie die entsprechenden Elemente der *FullEquivalence*. Wie später beschrieben, findet das *FeatureReasoning* unterschiedliche Mappings aufgrund einer gesetzten Referenz. Vom *Weaving Modell* wird zurzeit jedoch nur die

Unterscheidung aufgrund einer dieser Referenzen unterstützt, nicht deren Kombination. Dies mag für die beiden Beispieleditoren ausreichend sein, sollte aber in zukünftigen Implementierungen erweitert werden. Das *FeatureReasoning* erstellt zwei Mappings, falls Domänenelemente des linken auf unterschiedliche Domänenelemente des rechten Metamodells verweisen, und sich die Instanzen dieser Elemente durch das Vorhandensein einer Referenz unterscheiden.

Somit wird in den Zeilen 6 bis 16 die Eigenschaft *hasFeature* gesetzt, falls im Mappingelement ein solches Vorhanden ist. Da sich der Ausschnitt aus Listing 4.10 in einer Schleife befindet, werden daher zwei *ConditionalReferences* erstellt.

Nachdem alle Mappings als Weavingelemente vorliegen, kann die Ressource serialisiert werden. Hier wird erstmals als Parameter keine leere *Map* verwendet. Damit der *Atlas Model Weaver* das Metamodell findet, ohne dass dieses in Eclipse als Metamodell registriert ist, wird dem Serialisierer mitgeteilt, dass dessen Pfad im Modell übernommen werden soll. Dadurch wird vermieden, dass der Benutzer das Metamodell erst registrieren muss, bevor die generierte Datei mit dem Editor geöffnet werden kann.

erstelle Weaving Property Datei Damit der *Atlas Model Weaver* Editor die generierte Datei öffnen kann, wird für die korrekte Darstellung eine eigene Meta Datei (<file>.amw.prop) benötigt (Siehe Abbildung A.7 im Anhang). Für die Erstellung und Manipulation wird eine Java API bereitgestellt: *org.eclipse.weaver.core.WeaverXMLMetadata*.

Zuerst wird sowohl für das linke als auch für das rechte Modell ein *WovenModelDescriptor* erzeugt und in einer Liste gespeichert (*modelsList*). Jeder dieser *WovenModelDescriptor* Elemente enthält, neben einer eindeutigen Kennzeichnung, einen Anzeigenamen (*setNameProperty*), das zu ladende *Panel*, den Referenznamen, die *Weaving Modell* Referenz und den Pfad des Metamodells.

Listing 4.11: Erstellen einer “Conditional Equivalence”

```

1  WeaverXMLMetadata weaverXMLMetadata = new WeaverXMLMetadata(f);
2  ...
3  // left Model ...
4  modelsList.add(this.createWovenModelDescriptors("0", "leftModel", "
    DefaultWovenPanelExtension", "leftModel", "ModelRef", "null", "
    /" + file.getProject().getName() + "/mtbe-hot/metamodel/mmA.
    ecore"));
5  // right Model ...
6  modelsList.add(this.createWovenModelDescriptors("1", "rightModel", "
    DefaultWovenPanelExtension", "rightModel", "ModelRef", "null",

```

```

        "/" + file.getProject().getName() + "/mtbe-hot/metamodel/mmB.
        ecore"));
7      ...
8      weaverXMLMetadata.createXMLConfigFile(models, weavingModelPR,
        weavingModelPanel, wModelName, metamodelExtension,
        localMetamodelExtension, weavingEcore)

```

Danach wird die Methode *createXMLConfigFile* 4.11 aufgerufen. Dieser wird die erzeugte Liste (*List<WovenModelDescriptor>*) mit den zwei Modellen und der relative Pfad des *Weaving* Modells übergeben. Als *Weaving-Model-Panel* wird die *TransformationWeavingPanelExtension* verwendet und das Wurzelement bekommt den Namen *Model*. Da keine *Extension* verwendet wird, kann sowohl für den Parameter *metamodelExtension* sowie für *localMetamodelExtension* eine leere Liste verwendet werden. Sollten *Extensions* im *km3* Format eingebunden werden, müsste dazu der *plug-in Identifier*, sowie der relative Pfad der entsprechenden *Extension* übergeben werden. Nachdem in diesem Fall keine dies nicht nötig ist, ist lediglich der letzte Parameter *weavingEcore* interessant. Damit kann eine komplette Ecore Ressource als *Extension* geladen werden. Daher wird hier */mtbe-hot/metamodel/weavingmodel.ecore* übergeben.

Das Ergebnis dieser Funktion entspricht dem des *Weaver Wizard*, der auf dieselben Funktionen zurückgreift.

4.2.5 Implementierte Algorithmen

Der folgende Abschnitt beschreibt die im Rahmen der Implementierung des MTBE Prototyps umgesetzten Analysealgorithmen. Jeder Algorithmus muss, um von der Analyse angesprochen werden zu können, das *Interface org.tuwien.mtbe4.project.actions.reasoning.Reasoning* implementieren. Dieses spezifiziert lediglich eine Methode, nämlich *doReasoning*. Dieser Methode werden, wie in 4.6 bereits angesprochen, alle gefundenen Mappings, die Hilfsklasse, welche mit dem gerade vom Benutzer erstelltem Modell initialisiert ist, und die Ressource selber übergeben.

Simple Reasoning Algorithmus

Der *Simple Reasoning* Algorithmus findet Mappings die immer, das heißt ohne Bedingungen, von einer Metaklasse des linken auf eine Metaklasse des rechten Modells zeigen. In der Implementierung der *doReasoning* Methode für das *Simple Reasoning*, werden zu Beginn alle Klassen aus der Analyse ausgeschlossen, für welche durch andere Algorithmen bereits Mappings gefunden wurden. Diese würden zwar ohnehin ignoriert, dieser Schritt hilft aber die Laufzeit zu verkürzen.

Die Analyse selbst ist denkbar einfach. Es werden alle vom Benutzer definierten Mappings durchlaufen und für jedes wird angenommen, dass ein bedingungsloses Mapping vorliegt. Dies gilt bis das Gegenteil bewiesen wird.

Listing 4.12: Finden von Mappings durch das “SimpleReasoning”

```

1      //look in the match Table if we have this object in there already
      ...
2      if (match.get(from.eClass()) != null){
3          //found it, so we have to check if the classes where this
              mapping points too are the same, of not, this matching does
              not always
4          //point to the same class, meaning there is some condition
5          if (match.get(from.eClass()) != to.eClass()){
6              match.remove(from.eClass());
7              removed.add(from.eClass());
8          }
9      }
10     else if (!removed.contains(from.eClass())) {
11         //ok, this one was not in there, put it and if it is not proven
              wrong by another mapping, it is will match always...
12         match.put(from.eClass(), to.eClass());
13     }

```

Listing 4.12 zeigt den relevanten Teil des Codes. Falls im *Hashtable match* bereits ein Eintrag der linken Metaklasse vorhanden ist - und dieser Eintrag nicht dieselbe Metaklasse wie das aktuell untersuchte Mapping besitzt - wurde das Gegenteil bewiesen. Das heißt, es wurde bewiesen, dass zwei, vom Benutzer erstellte Mappings, unterschiedliche Metaklassen verbinden. Dieser Algorithmus findet nur Korrespondenzen die uneingeschränkt gelten. Dadurch kommen alle Verbindungen, die von dieser Metaklasse ausgehen, nicht in Frage. Gibt es noch keinen Eintrag (ab Zeile 11), wird für das aktuelle Mapping angenommen, dass es uneingeschränkt gilt, dies aber wiederum nur solange es nicht von einem späteren Mapping wieder gelöscht wird.

Containment Reasoning Algorithmus

Dieser Algorithmus ist dafür zuständig Mappings zu finden, welche auf der linken Seite dasselbe Domänenmetaelement besitzen, aber auf unterschiedliche Metaelemente des rechten Modells zeigen. Das Unterscheidungsmerkmal stellt das *Containment property* des linken Metamodells dar. Dabei muss aber gelten, dass Elemente innerhalb eines solchen, immer auf ein gleiches Element der rechten Seite zeigen, um gefunden werden zu können. Abbildung 4.8 veranschaulicht ein solches Szenario. Für das UML-Klassendiagramm wird dabei das, in Abschnitt 2.3.1 vorgestellte Metamodell verwendet, mit der Anpassung, dass ein *Property*, falls es eine Rolle darstellt nicht in einer Klasse, sondern in der Assoziation enthalten ist. Das verwendete ER-Modell ist eine Instanz des Metamodells aus

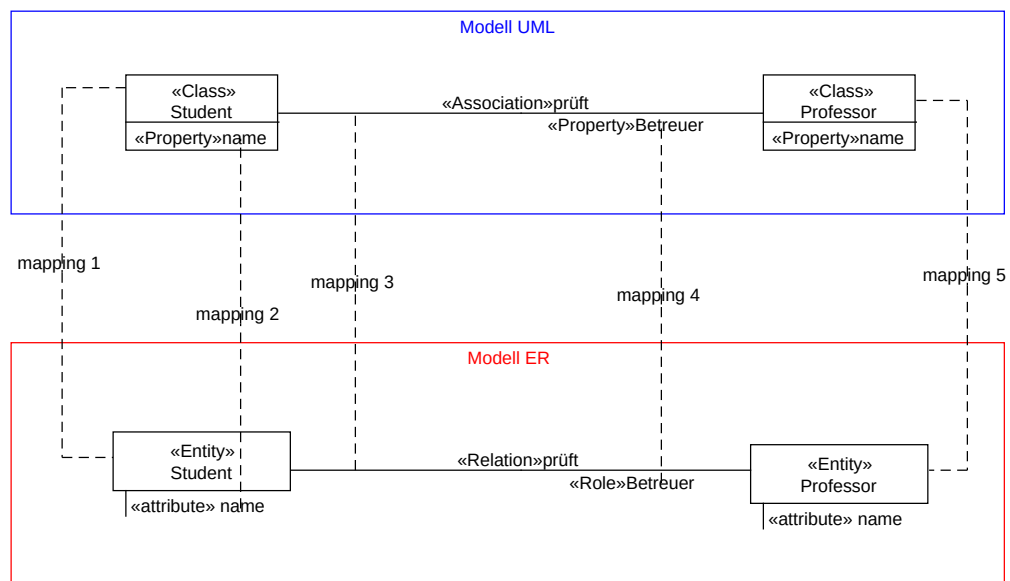


Abbildung 4.8: Beispiel für Benutzermappings

dem Abschnitt EMF 2.6. Wird das Modell 4.8 analysiert, erkennt das *SimpleReasoning*, dass es zwar zwei Mappings (mapping 1 und mapping 5) gibt, welche eine Klasse als Quellelement besitzen, die Annahme, dass eine Klasse immer auf eine Entität abgebildet wird, wird von diesen aber nicht widerlegt. Auch die Annahme, dass einer Assoziation immer eine Relation gegenübersteht (mapping 3), wird nicht falsifiziert. Anders sieht es für das *Property* aus. Mapping 4 widerlegt die Annahme aus mapping 2, dass ein *Property* immer auf ein Attribut zeigt und wird somit vom *SimpleReasoning* nicht beachtet. Werden mapping 3 und 4 im nächsten Schritt nun dem *ContainmentReasoning* übergeben, erkennt dieses, dass zwar das *Property* auf zwei unterschiedliche Domänelemente des ER Modells verweist, dies aber immer dann geschieht, wenn sich das *Property* in unterschiedlichen Containern befindet.

Im Beispiel werden vom *ContainmentReasoning* daher zwei Regeln abgeleitet, nämlich,

1. dass ein *Property* dann zu einem Attribut transformiert werden muss, falls es in einer Klasse enthalten ist, und,
2. dass ein *Property* zu einer Rolle wird, falls es Teil einer Assoziation ist.

Mappings die nicht immer auf dasselbe Element des rechten Metamodells verweisen wenn die Elemente des linken Metamodell im gleichen Typ enthalten sind, können nicht erkannt werden. Für diese muss offensichtlich noch eine zusätzliche

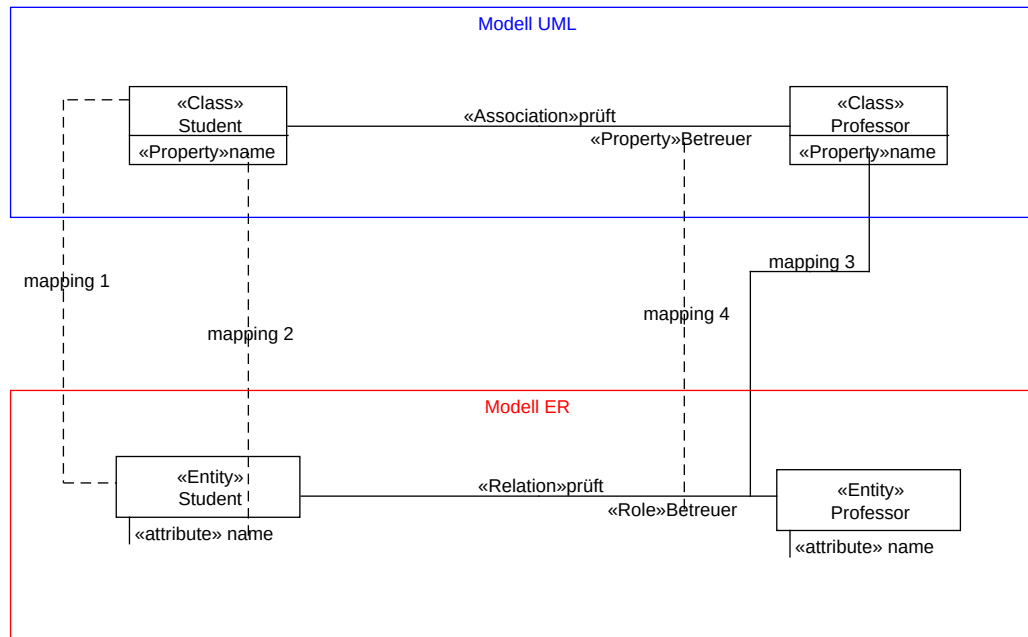


Abbildung 4.9: Ein Mapping das nicht vom ContainmentReasoning gefunden wird

Regel existieren, welche diese weitere Unterscheidung verursacht. Abbildung 4.9 zeigt einen solchen Fall. Dabei handelt es sich in dieser Abbildung höchstwahrscheinlich um einen Fehler des Benutzers. Das mapping 2 verbindet ein *Property* mit einem Attribut und mapping 4 ein *Property* mit einer Rolle. Dies stellt kein Problem dar, da sich diese in unterschiedlichen Containern befinden. Mapping 3 aber zeigt, dass ein *Property*, welches in einer Klasse enthalten ist, zusätzlich mit einer Relation verbunden sein kann. Daher muss eine zusätzliche, noch nicht entdeckte Bedingung existieren, welche mapping 2 und 4 unterscheidet. Das *ContainmentReasoning* ist aber nicht in der Lage diese Bedingung zu erkennen und findet somit in diesem Modell keine Korrespondenzen auf der Metaebene. Es werden viele solcher zusätzlichen Regeln für unterschiedliche Domänen existieren, daher sollte in Nachfolgeprojekten dieses Framework mit zusätzlichen Algorithmen erweitert werden, um möglichst viele Mappings finden zu können und den MTBE Ansatz auch in der Praxis anwendbar zu machen.

Feature Reasoning Algorithmus

Dieser ist wohl der komplexeste der drei, im Zuge dieser Arbeit entwickelten Algorithmen. Ziel des *Feature Reasoning* ist es, zu erkennen, wenn mehrere Mappings von gleichen Elementen des linken Modells auf unterschiedliche Elemente des rechten Modells verweisen. Dabei erkennt das *Feature Reasoning* deren un-

terschiedliche Bedeutung falls eine solche auf Basis von belegten Features des linken Modellelements möglich ist.

Als Kriterium dient nicht wie beim *Containment Reasoning* der Container des zu untersuchenden Elements, sondern dessen gesetzte Referenzen. Es wird nicht differenziert, worauf diese Referenzen zeigen, sondern die Entscheidung basiert lediglich auf der Tatsache, ob eine Referenz tatsächlich belegt wurde. Falls nun, die vorhin getroffene Einschränkung nicht gilt, das Modell aus Abbildung 4.8 also eine Instanz des Metamodells aus Abschnitt GMF ist, wird ersichtlich, dass das *ContainmentReasoning* das Unterscheidungsmerkmal zwischen mapping 2 und mapping 4 nicht finden kann. Beide *Property* Elemente sind in der Klasse *Student* enthalten. Zusätzlich geht aber aus dem Metamodell hervor, dass ein *Property* sowohl eine Referenz auf eine Klasse, als auch auf eine Assoziation besitzt. Dabei wird Letztere nur belegt, falls es sich um eine Rolle handelt. In diesem Szenario unterscheiden sich die beiden Elemente zwar nur durch diese eine Referenz, denkbar ist aber auch, dass sich der Unterschied aus einer Kombination von gesetzten und nicht gesetzten Referenzen zusammensetzt. Dies wird momentan vom *Weaving Modell* noch nicht unterstützt. Daher wurde auch darauf verzichtet solche Kombinationen zu identifizieren. Es könnte aber durch relativ wenige Modifikationen erreicht werden. Es ist anzunehmen, dass künftige Erweiterungen höchstwahrscheinlich auf Attributen und Referenzen aufbauen werden und somit wird auf die Implementierung des *Feature Reasoning* detaillierter eingegangen.

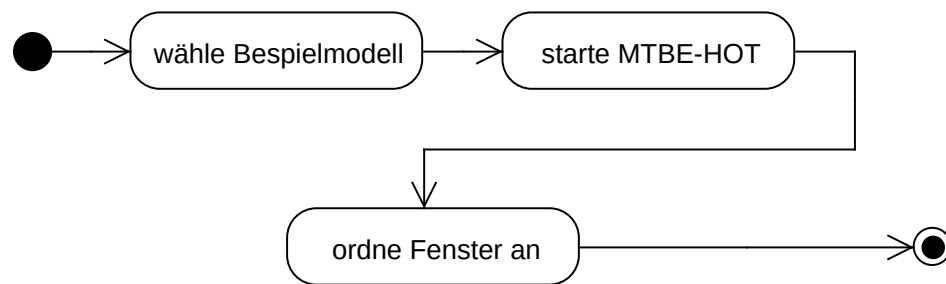
Dieser Algorithmus befindet sich in einem eigenen Plug-in um die Erweiterbarkeit zu demonstrieren, interessant ist für das Verständnis aber lediglich die Klasse *FeatureReasoning*. Um die Funktionsweise zu erläutern dient wieder Beispiel 4.8, wobei auf die Ausführung der anderen Algorithmen verzichtet wird. Zu Beginn stehen also alle Mappings (1 bis 5) in der Liste *mappings*.

1. Gleich zu Anfang liefert die *ReasonHelper* Klasse alle Klassen des linken Metamodells, welche in den Mappings vorkommen. Die Korrespondenzen aus Beispiel 4.8 würden somit *Class*, *Property* und *Association* ergeben. Obwohl sowohl die *Class*, als auch das *Property* mehrfach in den Mappings enthalten sind, werden diese nur einmal in die Liste aufgenommen.
2. Für jede dieser Klassen wird eine Schleife durchlaufen:
3. Wiederum liefert der *ReasonHelper* eine Liste. Dieser gruppiert die vorhandenen Mappings nach dem Typ des rechten Metamodells und gibt diese in einer Liste aus Listen zurück. Im Beispiel wäre das:
 - für *Class*: ([mapping1, mapping5])

- für *Association*: ([mapping3])
- für *Property*: ([mapping2],[mapping4])

Natürlich ist die Analyse in einem Beispiel mit wenigen Mappings noch relativ einfach. Bereits hier könnte festgestellt werden, dass lediglich das *Property* auf unterschiedliche Domänenelemente des rechten Metamodells verweist, da mehr als eine Liste zurückgegeben wird. Für die beiden Anderen könnte hier abgebrochen werden, es ist aber ohnedies anzunehmen, dass vorrangegangene Algorithmen diese Mappings bereits erfolgreich identifiziert haben und diese somit nicht in der Liste wären.

4. Das *FeatureReasoning* findet Mappings, welche sich nur durch die belegten *Features* unterscheiden, daher dürfen sich die linken Domänenelemente eines Mappings in einer der Listen, in diesem Punkt nicht unterscheiden. Dies würde bedeuten, dass ein weiteres Unterscheidungsmerkmal besteht, welches nicht identifiziert werden kann. Daher werden nun alle Mappings der einzelnen Listen untersucht. Das jeweils erste Mapping dient als Prototyp, alle anderen dürfen von diesem nicht abweichen. Im Beispiel trifft dies zu, da ohnehin nur ein Element in den einzelnen Listen existiert. Es würde aber auch zutreffen, wenn Korrespondenzen für mehrere Rollen existieren würden.
5. Nun wird geprüft, ob mehrere Prototypen für eine Klasse existieren und, um Fehler in der weiteren Ausführung zu vermeiden, ob die linken Klassen tatsächlich alle vom gleichen Typ sind. Hier fallen nun *Class* und *Association* weg, da für diese nur ein Prototyp gefunden wurde.
6. Im nächsten Schritt werden die gesetzten *Features* verglichen. Dazu werden die Prototypen jeder Liste, also im Prinzip das erste Element jeder Liste, durchlaufen und aufsummiert, welche der *Features* gesetzt sind und welche nicht. Ergibt die Summe eines *Features* die Anzahl der Prototypen oder 0, besteht kein Unterschied, ansonsten wird für jede Liste ein Mapping Element erzeugt und der Typ auf *mapping feature difference* gesetzt. Diesen werden in den entsprechenden Listen alle Features angehängt, die entweder gesetzt, oder nicht gesetzt, den Unterschied ausmachen. Im Beispiel würden zwei Mappings erzeugt, wobei beide als *from* das *Property* besitzen. *to* weist aber beim Einen auf die *Rolle* und beim anderen auf das *Attribut* und in der Liste der gesetzten *Features* existiert bei der Rolle das *association Feature*, wohingegen dieses *Feature* beim Attribut in den nicht gesetzten *Features* vorhanden ist.

Abbildung 4.10: Arbeitsweise der Methode *TransformationAction*

4.2.6 Transformation

Diese Komponente erstellt aus dem generierten *Weaving* Modell eine ATL-Datei und transformiert damit ein geladenes Beispiel Modell in das jeweilige andere Modell, um eine Vorschau bereitzustellen. In Abbildung 4.10 wird die Arbeitsweise dieser Komponente veranschaulicht. Der Quellcode ist in der Klasse *org.tuwien.mtbe4.project.actions.TransformationAction* zu finden.

Der Benutzer wählt über die Menüleiste oder über die Buttons in der Werkzeugleiste die Richtung der Transformation. Dafür stehen ihm die Funktionen *do transformation left-right* beziehungsweise *do transformation right-left* im MTBE Menü zur Verfügung.

Transformation Dialog Bevor eine Transformation durchgeführt wird, wird ein Dialog geöffnet, in welchem das gewünschte Beispieldiagramm angegeben werden kann. Dieses wird in der MTBE-Perspektive gemeinsam mit dem Ergebnis der Transformation angezeigt. Der Pfad auf das Beispiel wird gespeichert, um eine einfachere Bedienung bei mehrfacher Ausführung zu ermöglichen.

MTBE-HOT Die *Higher Order Transformation* wurde aus dem Paper [40] übernommen und durch einige kleinere Anpassungen in das Framework integriert. Sie befindet sich in einem frühen Entwicklungsstadium und liefert daher noch nicht das gewünschte Ergebnis. Gesteuert wird sie in der Methode *updateMTBEHot*. Mit Hilfe des *AntRunners* werden erst das Skript *run.xml* ausgeführt, welches aus dem *Weaving* Modell ATL Regeln extrahiert. Dem Skript werden die Parameter *metaPath* und *hotPath* übergeben. Im *metaPath* sind die beiden Metamodelle, sowie die *Mapping Language* in Form des Ecore Modells *weavingOut.ecore* zu finden.

Skript Das Script *run.xml* (Siehe Anhang A.8) transformiert das generierte *Weaving* Modell in ein ATL Modell. Dieses kann danach verwendet werden um die ATL Regeln zu erstellen.

Das zweite Skript *transformation.xml* (Siehe Anhang A.9) führt die Beispieltransformation durch. Es transformiert das gewählte Modell mit den zuvor generierten ATL-Regeln und zeigt das Ergebnis in den MTBE Perspektiven an.

Fenster anordnen Abschließend wird die Methode *arrangeWindows(IProject fProject, IFile file, Boolean leftRight)* aufgerufen, welche, je nach Perspektive das Ergebnis der Beispieltransformation sowie das *Weaving Modell* und den ATL Code öffnet. Diese Ressourcen werden neben und unter dem Editor angezeigt. Die Methode wird im Zusammenhang mit der Funktionsbeschreibung der Perspektiven genauer erläutert. (Abschnitt 4.2.7)

4.2.7 MTBE-Perspektiven

Das MTBE Plug-in erweitert Eclipse um zwei Perspektiven, welche die Arbeit an einem MTBE Projekt erleichtern sollen.

MTBE Developer Perspektive Der Einsatz dieser Perspektive ist für einen Entwickler gedacht, der eine schnelle Transformation mittels MTBE durchführen möchte. Diese Perspektive besteht aus dem Diagrammeditor, in dem Diagramme aus beiden Anwendungsdomänen mit den entsprechenden Mappings gezeichnet werden können. Unter dem Diagrammeditor befinden sich noch zwei weitere Fenster, in denen eine Vorschau der Transformation angezeigt wird. Sie dient zum Testen der Vollständigkeit der eingezeichneten Beispiel Mappings. Je nach dem in welche Richtung die Transformation durchgeführt werden soll, ist im einen Fenster das Beispielmmodell des Benutzers und im anderen das Ergebnis der Transformation auf Basis der eingezeichneten Mappings zu sehen.

MTBE Debug Perspektive Diese Eclipse Perspektive bietet einen tieferen Einblick in MTBE. Zusätzlich zu den Fenstern der Developer Perspektive werden noch der *ModelWeaver* und der generierte ATL Code angezeigt. Im *ModelWeaver* kann der Benutzer die durch die Analyse gefundenen Mappings bearbeiten und erweitern. Dadurch kann Wissen über Domänen eingebracht werden ohne das Plug-in durch einen eigenen Algorithmus zu erweitern.

Die Perspektiven werden im *plugin.xml* definiert.

Listing 4.13: Definition der *Perspectives-Extensions* im plugin.xml

```

1  <extension point="org.eclipse.ui.perspectives">
2      <perspective
3          name="MTBE_Developer"
4          icon="icons/reasonButton.gif"
5          class="org.tuwien.mtbe4.project.MTBEPerspectiveDev"
6          id="org.tuwien.mtbe4.project.DevPerspective">
7      </perspective>
8  </extension>
9
10 <extension point="org.eclipse.ui.perspectives">
11     <perspective
12         name="MTBE_Debug"
13         icon="icons/reasonButton.gif"
14         class="org.tuwien.mtbe4.project.MTBEPerspective"
15         id="org.tuwien.mtbe4.project.DebugPerspective">
16     </perspective>
17 </extension>
18 </plugin>

```

Der genaue Aufbau der Perspektive wird in der Methode *MTBEPerspectiveDev* und *MTBEPerspective* beschrieben. Die Fenster für die Beispieltransformation werden erst beim Durchführen der Transformation geladen und unter dem Diagrammeditor geöffnet. Zuständig für die Anordnung der Fenster ist die Methode *arrangeWindows(IProject fProject, IFile file, Boolean leftRight)* in der Klasse *TransformationAction*. Der Parameter *Boolean leftRight* gibt an, in welche Richtung die Transformation durchgeführt wurde, da davon die Anordnung der Fenster abhängt. Abhängig davon, welche der beiden Perspektiven aktiv ist, werdend die entsprechenden Fenster geöffnet und angeordnet. In der *Debug Perspektive* werden zusätzlich zu der Beispieltransformation das *Weaving Modell* und der ATL Code geladen.

Zur Anordnung der Fenster werden die Klassen aus dem Paket *org.tuwien.mtbe4.project.dnd* verwendet. Der Code wurde aus dem *org.eclipse.ui.tests* ¹ übernommen und angepasst. Im Grunde wird beim Anordnen ein *Drag'n'Drop* Befehl simuliert, sodass die Fenster geteilt und angeordnet werden.

Da bei diesem Vorgang auf die interne API zugegriffen wird, kann es bei einer neuen Version von Eclipse unter Umständen zu Problemen kommen. Siehe dazu Abschnitt 5.

Screenshot 4.11 und 4.12 veranschaulichen den Aufbau der zwei Perspektiven. Abbildung 4.11 zeigt die MTBE Debug Perspektive und Abbildung 4.12 die MTBE Developer Perspektive.

¹[http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.tests/Eclipse UI Tests/org/eclipse/ui/tests/api/Bug95357Test.java?hideattic=0&view=markup](http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.tests/Eclipse%20UI%20Tests/org/eclipse/ui/tests/api/Bug95357Test.java?hideattic=0&view=markup)

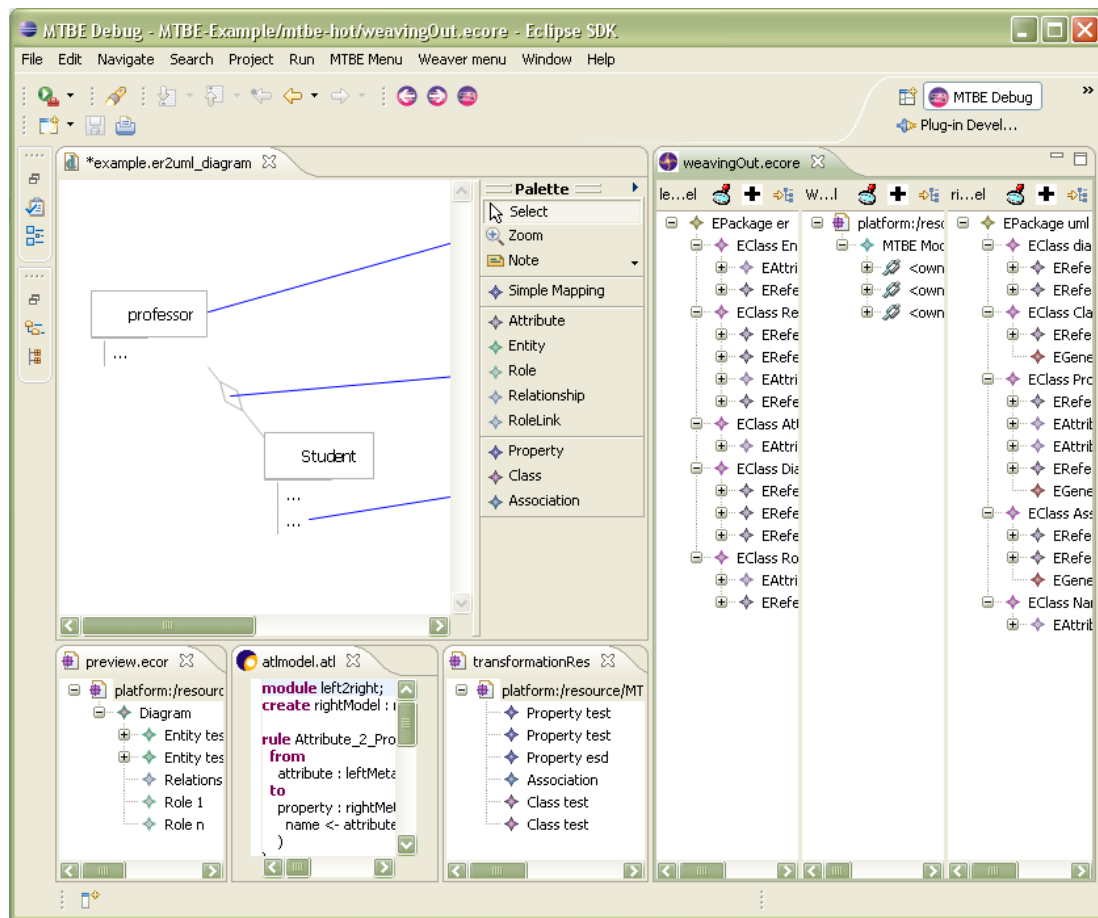


Abbildung 4.11: MTBE Debug Perspective

4.2.8 Erweiterbarkeit

Analysealgorithmen Bei der Realisierung der MTBE Implementierung wurde darauf geachtet passende Schnittstellen für die Erweiterung um neue Analysealgorithmen anzubieten.

Dazu wurde ein Extension Point *Reasoning* angelegt, mit dem zugehörigen Schema *reasoning.exsd* 4.14.

Listing 4.14: Extension Point des MTBE Plug-ins

```

1 <!ELEMENT extension (algorithm*)*>
2 <!--ATTLIST extension
3 point CDATA #REQUIRED
4 id CDATA #IMPLIED
5 name CDATA #IMPLIED-->
6
7 <!--ELEMENT algorithm EMPTY-->
8 <!--ATTLIST algorithm
9 name CDATA #REQUIRED
10 class CDATA #REQUIRED
11 id CDATA #REQUIRED
12 description CDATA #IMPLIED-->

```

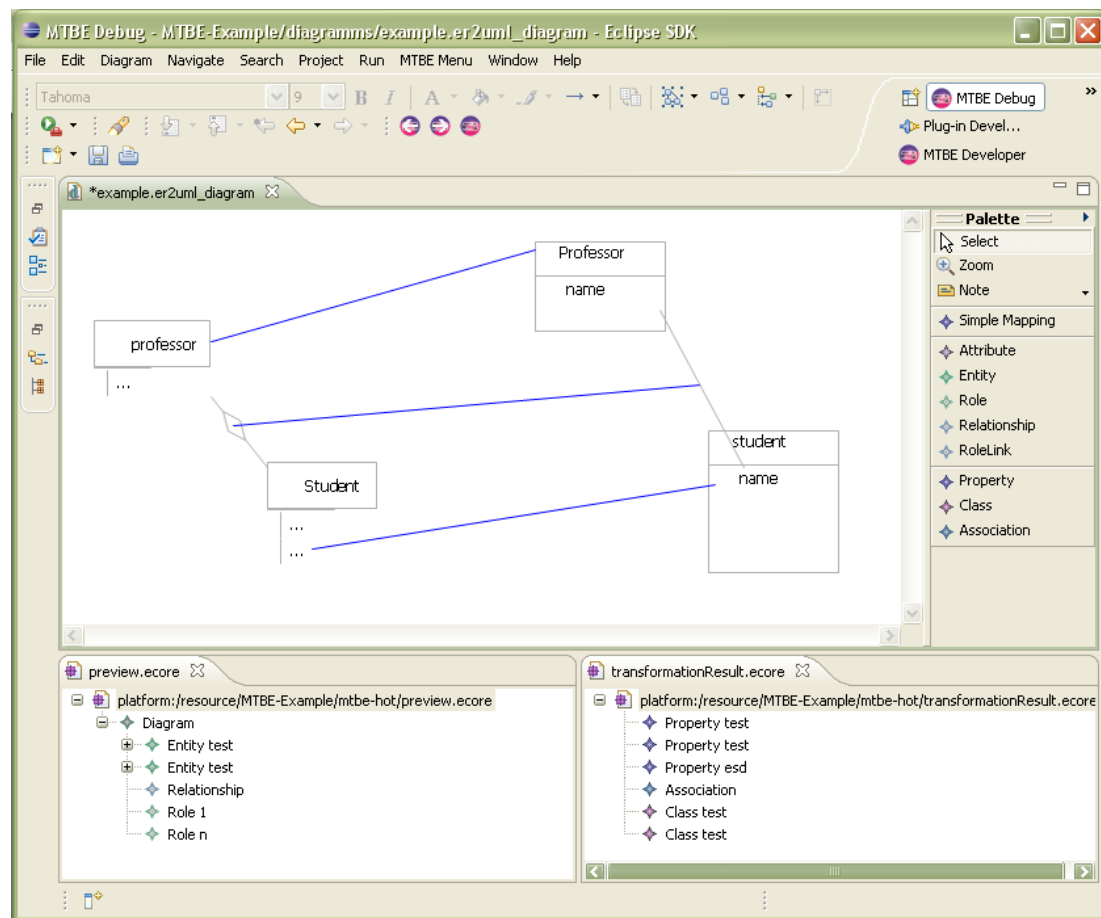


Abbildung 4.12: MTBE Developer Perspective

Erweiterung des MTBE Plug-ins Soll nun ein neuer Analysealgorithmus zum MTBE Plug-in hinzugefügt werden, muss dieser entsprechend dem Schema 4.14 im *plugin.xml*-File hinzugefügt werden. Der Algorithmus selbst muss das Interface *Reasoning* implementieren und in das entsprechende Paket *org.tuwien.mtbe4.project.actions.reasoning* gelegt werden. Folgendes Listing 4.15 zeigt die Einträge für die in den Prototyp dieser Arbeit umgesetzten Algorithmen *Simple Reasoning* und *Containment Reasoning*.

Listing 4.15: Hinzufügen neuer Algorithmen im plugin.xml

```

1
2   <extension
3       id="reasoningalgorithm"
4       name="Reasoning_Algorithm"
5       point="org.tuwien.mtbe4.project.reasoning">
6       <algorithm
7           class="org.tuwien.mtbe4.project.actions._
8               reasoning.SimpleReasoning"
9           id="SimpleReasoning"
10          name="Simple_Algorithm">
11      </algorithm>
12      <algorithm

```

```

12         class="org.tuwien.mtbe4.project.actions.␣
           reasoning.ContainmentReasoning"
13         id="ContainmentReasoning"
14         name="Containment␣Reasoning">
15     </algorithm>
16 </extension>

```

Zuständig für die Kontrolle der Analyse-Algorithmen während der Laufzeit ist die Klasse *ExtensionReasoning*, sie liefert alle im *plugin.xml* definierten Algorithmen als Vektor von *ExtensionReasoning* Objekten zurück, jedes dieser Objekte enthält eine eindeutige Id, den Namen, den Typ und eine kurze Beschreibung. Der Namen und die Beschreibung werden in der Dialogbox zur Auswahl der Algorithmen für den Benutzer angezeigt. (Siehe Abbildung 4.19) Nach einem Neustart der Eclipse Umgebung können alle auf diese Weise hinzugefügten Algorithmen, über die gewohnten MTBE Benutzerschnittstellen angesteuert werden. Wichtig ist das beim Neustart Eclipse mit dem Parameter *-clean* gestartet wird, denn nur dann wird das Plug-in neu geladen und die Änderungen übernommen.

Erweiterung mit einem eigenen Plug-in Die notwendigen *Plug-in Dependencies* sind neben dem *org.eclipse.ui* und dem *org.eclipse.core.runtime*, das im Rahmen dieser Arbeit entwickelte *org.tuwien.mtbeX.project* sowie noch *org.eclipse.emf* und *org.eclipse.emf.ecore*.

Dem *Reasoning Algorithm Extension Point* muss nun noch ein oder je nach Wunsch mehrere Algorithmen mit dem Schema *reasoning.exsd* (Listing 4.14) hinzugefügt werden sowie die entsprechende Klasse, die das Interface *Reasoning* implementiert, erstellt werden.

Auch diese Algorithmen können nach einem Neustart der Eclipse Umgebung sofort eingesetzt und über die gewohnten MTBE Benutzerschnittstellen angesteuert werden.

Der *FeatureReasoning* Algorithmus wurde auf diese Weise implementiert.

Vorschläge für Erweiterungen Zusätzlich zur Erweiterung mittels neuer Analysealgorithmen bietet MTBE ausgehend vom vorgeschlagen Framework sowie dieser Arbeit noch verschiedene Ansatzpunkte. Siehe dazu den Abschnitt 5.+

4.3 Installationsanleitung

Folgende Schritte sind notwendig um das MTBE Plug-in zu verwenden.

Installation Eclipse Download und Installation von Eclipse. Durch das Entpacken der Eclipse Zip-Datei ², entsteht automatisch der Eclipse Ordner mit allen Unterordnern und benötigten Dateien. MTBE wurde mit der Eclipse Version 3.4.0 (Build id: I20071101-2000) entwickelt und getestet.

Installation EMF und GMF Entpacken der jeweiligen Zip-Dateien, sie beinhalten alle einen Eclipse Ordner mit den ergänzenden Dateien zum Eclipse SDK.

- Eclipse Modeling Framework EMF 2.4 ³
- Graphical Editing Framework (GEF) 3.3.1 ⁴
- Eclipse Graphical Modeling Framework (GMF) 2.0.1 ⁵

Installation ATLAS Model Management Architecture Download und Installation des ATL Plug-ins.

- ATL Bundle 2.0 Standard Version 2.0RC2 ⁶

Installation des MTBE Plug-ins Entpacken der MTBE Zip-Datei, sie beinhaltet das Plug-in in der für Eclipse gewünschten Ordnerstruktur. Kopieren von *org.tuwien.mtbe.project_1.x.x.jar*⁷ in den Eclipse-Plug-in Ordner.

Eclipse neu starten Abschließend muss Eclipse neu gestartet werden, eventuell mit dem Parameter “-clean”.

Weitere Einstellungen und Anmerkungen Um die volle Funktionalität des MTBE Plug-ins zu gewährleisten, müssen noch folgende Einstellungen vorgenommen werden:

Anpassen des Ant Classpath In den Eclipse Einstellungen (Window>Preferences) im Unterpunkt “Ant>Runtime” muss den *Global Entries* das externe Jar file *ebnfextractor.jar* hinzugefügt werden. Es befindet sich in den Libraries des Plug-ins “org.eclipse.m2m.atl.engine_2.0.0”.

²<http://www.eclipse.org/downloads/>

³<http://www.eclipse.org/modeling/emf/downloads/>

⁴<http://www.eclipse.org/gef/downloads/>

⁵<http://download.eclipse.org/modeling/gmf/downloads/>

⁶<http://www.eclipse.org/modeling/m2m/downloads/>

⁷<http://www.modelcvs.org/prototypes/mtbe.html#prototype>

Registrieren der Metamodelle Um die Beispieltransformation richtig anzeigen zu können, müssen die zugehörigen Metamodelle registriert werden. Beim Erstellen des zusammengefassten Diagrammeditors werden diese in den Ordner “mtbe-hot/metamodel” mit den Namen *mmA* und *mmB* kopiert.

4.4 Benutzerdokumentation

4.4.1 Beschreibung der Funktionalität

Mit dem *Model Transformation By-Example* Plug-in können ohne konkretes Wissen über Metamodelle und die Transformationsregeln einfach und schnell Modell-zu-Modell Transformationen durchgeführt werden. Es können Transformationen von einem beliebigen Modell in jedes beliebige andere Modell transformiert werden. Die einzige Bedingung ist, dass von beiden Modellen ein GMF-Editor und ein EMF-Editor vorhanden ist.

Der Benutzer erstellt ein neues Project - “New MTBE Project ...” - wählt sein Ausgangs und sein Zielmodell in Form des *gmfmap-Files* und gibt jedem einen Namen. Das Plug-in generiert automatisch einen neuen Editor in dem beide Modelle gemeinsam gezeichnet werden können. Der Benutzer kann in diesem Editor beispielhaft Modelle aus beiden Problemdomänen zeichnen, und diese dann mit Mappings verbinden. Aus den eingezeichneten Mappings können nun mittels ausgewählten Algorithmen Transformationsregeln generiert werden. Um zu testen ob die eingezeichneten Beispielmappings ausreichen, kann der generierte Transformationsregelsatz auf ein ausgewähltes Modell ausgeführt werden, wahlweise von rechts nach links oder links nach rechts.

4.4.2 Beschreibung der Benutzeroberfläche

Dem Benutzer stehen über das MTBE Menu drei Funktionen zur Verfügung, dargestellt in Abbildung 4.13 (1). Zum einen der Befehl “do reasoning”, welcher das aktuelle Diagramm-File mit den zwei Modellen und den Mappings analysiert und daraufhin ein *Weaving Modell* erstellt. Des Weiteren stehen dem Benutzer die zwei Funktionen “do transformation left-right” und “do transformation right-left” zur Verfügung. Diese Befehle generieren aus dem erstellten *Weaving Modell* den ATL Code und führen eine Beispieltransformation durch, je nach Funktion von rechts nach links oder links nach rechts. Jeder dieser Befehle ist außerdem noch über die Symbolleiste (Abbildung 4.13) an steuerbar, siehe Abbildung 4.13 (2).



Abbildung 4.13: Screenshot der MTBE Menuleiste

MTBE bietet zwei Perspektiven, *MTBE Developer* Perspektive und die *MTBE Debug* Perspektive, die je nach Anwendungsfall eingesetzt werden können, dargestellt Abbildung 4.13 (3).

- *MTBE Developer Perspektive* - Diese Eclipse Perspektive ist für einen Entwickler gedacht, der nur eine Transformation mittels MTBE durchführen möchte. Diese Perspektive besteht aus dem Diagrammeditor und den zwei Fenstern für die Transformation.
- *MTBE Debug Perspektive* - Die MTBE Debug Perspektive bietet zusätzlich zur Ansicht der Developer Perspektive noch den *Model Weaver* und den generierten ATL Code. Der *Model Weaver* ermöglicht dem Benutzer die durch MTBE generierten Mappings zu bearbeiten oder noch weitere eigene Mappings hinzuzufügen.

4.4.3 Programmooptionen

Bevor eine Analyse ausgeführt wird öffnet sich ein Dialogfeld in dem der Benutzer die für ihn relevanten Analysealgorithmen auswählen kann. In der aktuellen Version dieses Prototyps stehen drei Algorithmen zur Auswahl, der *Simple Reasoning*, *Containment Reasoning* und der *Feature Reasoning* Algorithmus. Je nach Anwendungsfall und Entwicklungsstatus werden weitere Algorithmen zur Auswahl stehen.

Das Plug-in merkt sich die Auswahl des Benutzers, sie wird jeweils in Kombination mit der zu analysierenden Datei abgespeichert. Über die rechte Maustaste kann in den *Properties* die Auswahl der Algorithmen angesehen und verändert werden.

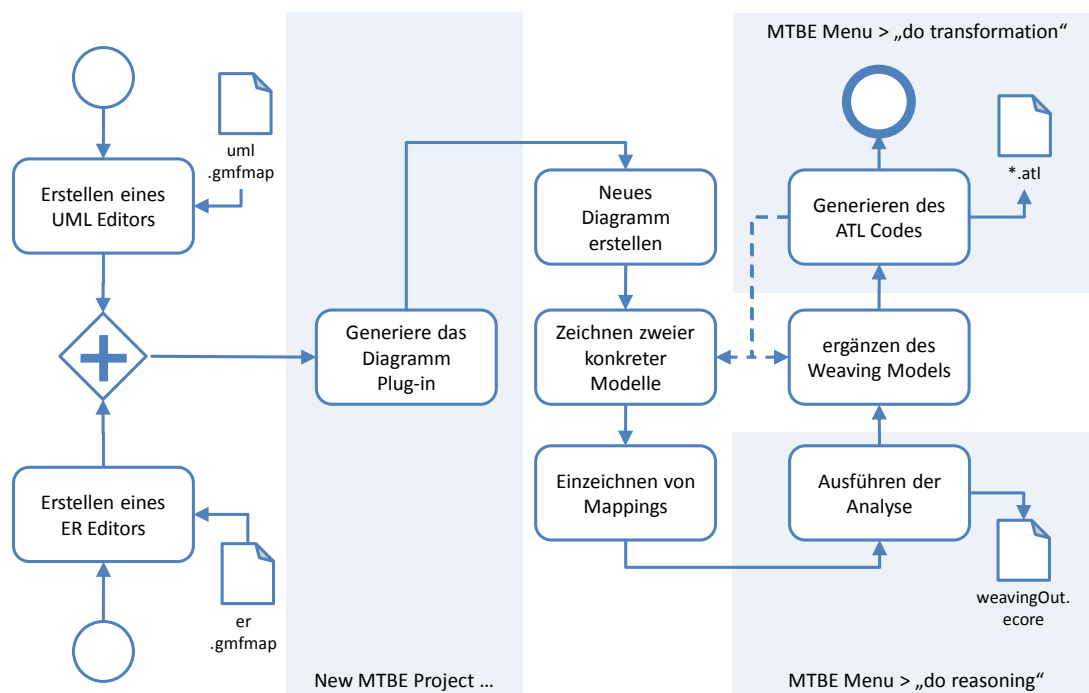


Abbildung 4.14: Überblick über den MTBE Prozess

4.4.4 Konfigurationsmöglichkeiten

- *Analysealgorithmen* - Im *plugin.xml* des MTBE Plug-ins können Analysealgorithmen hinzugefügt oder entfernt werden. Eine detaillierte Beschreibung hierzu findet sich im Abschnitt 4.2.8.
- *Perspektiven* - Die zu Verfügung gestellten Perspektiven können in den Methoden *MTBEPerspectiveDev* und *MTBEPerspective* nach Bedarf erweitert werden.

4.5 Beispiel

4.5.1 Einleitung

Folgender Abschnitt beschreibt die konkrete Anwendung des MTBE Plug-ins anhand eines einfachen Beispiels. Durchgeführt wird eine Transformation von einem UML in ein ER Diagramm und die umgekehrte Transformation, ausgehend von zwei von rein deklarativen erstellten GMF Editoren für UML sowie ER. In welcher Reihenfolge der MTBE Prozess abläuft und wie was und wann was erstellt wird, ist in Abbildung 4.14 dargestellt.

Ausgehend von den zwei Dateien *er.gmfmap* und *uml.gmfmap* wird ein neues Diagramm Plug-in erzeugt. In diesem ist es möglich Diagramme auf Basis bei-

der Metamodelle zu zeichnen. Zwischen diesen beiden konkreten Modellen aus der gleichen Problemdomäne werden Korrespondenzen eingezeichnet, aus denen dann im Weiteren über ausgewählte Analysealgorithmen ein *Weaving Modell* erzeugt wird. Das *Weaving Modell* zwischen den beiden Matamodellen kann nun bei Bedarf noch manuell erweitert werden. Im nächsten Schritt kann der ATL Code erzeugt werden, je nach Benutzereingabe für eine UML2ER Transformation oder eine ER2UML Transformation. Mittels einer Beispieltransformation kann der ATL Code getestet werden, bei Bedarf können wieder neue Diagrammelemente oder neue Mappings hinzugefügt und die Analyse erneut ausgeführt werden. In den nächsten Punkten wird auf die einzelnen Schritte im Detail eingegangen.

4.5.2 Erstellen der Ausgangseditoren

Um eine Transformation mit MTBE zwischen zwei Modellen ausführen zu können, muss von beiden Modellen ein GMF Editor vorhanden sein. Entwickelt wurde das MTBE Plug-in dahin, dass jeder GMF Editor verwendet werden kann (Mögliche Einschränkungen siehe 4.6).

Wie ein GMF Editor erstellt wird, wurde schon im Kapitel 2.3.1 am Beispiel des UML Klassendiagramms beschrieben. In folgendem Beispiel werden ein UML Editor und ein ER Editor herangezogen, jeweils basierend auf einem vereinfachten Metamodell. Details zu diesen Editoren finden sich im Abschnitt 2 dieser Diplomarbeit Arbeit.

4.5.3 Generieren des Diagramm Plug-ins

Im ersten Schritt erstellt der Benutzer ein neues Projekt. Dazu wählt er in der Menüleiste “File >New >Other...” und wählt dort im Ordner MTBE den Unterpunkt “New MTBE Project”. Siehe Abbildung 4.15 (1). Im Wizard Fenster “New MTBE Project” muss nun der Name des neuen Projektes eingegeben werden Abbildung 4.15 (2). Daraufhin erscheint das Fenster “MTBE Graph Wizard”, dargestellt in Abbildung 4.15 (3). Hier müssen nun die beiden GMF Editoren in Form der *GMF Map* (*.gmfgraph) ausgewählt werden. Beiden Modellen muss ein Name zugeordnet werden, der dann im Folgenden zur Benennung der *Packages* aber auch des Diagramm-Plug-ins herangezogen wird. Mit dem Betätigen des “Finish” Buttons, wird der Code angestoßen der für das Zusammenfassen der Editoren zuständig ist. Nachdem die beiden Editoren zusammengefasst worden sind, öffnet sich ein weiteres Dialogfenster Abbildung 4.16, über welches das neue Diagramm Plug-in als exportiert werden soll. Unser Wizard selektiert die beiden zu exportierenden Datei vor, daher muss, wenn der Pfad korrekt ist, nur mehr der

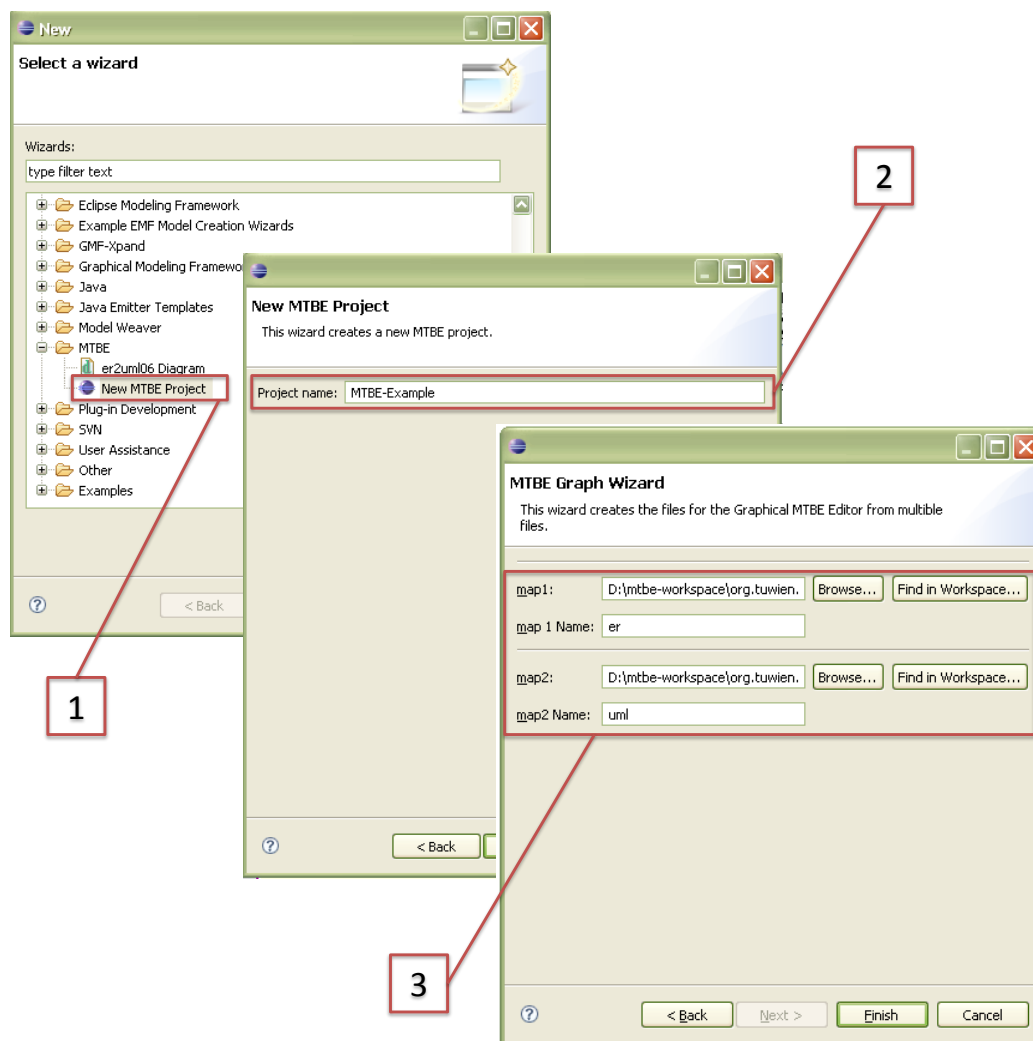


Abbildung 4.15: MTBE Wizard - Erstellen eines neuen MTBE Projektes

“Finish” Button gedrückt werden. Es folgt eine Aufforderung zum Neustart der Eclipse Umgebung. Nach dem Neustart kann das Diagramm Plug-in verwendet werden.

4.5.4 Zeichnen der Modelle

Das im letzten Schritt erzeugte Diagramm Plug-in befindet sich nun im “New”-Dialog im Verzeichnis MTBE. Der Benutzer erstellt nun ein neues Diagramm **.er2uml* mit dem Namen “example”, das MTBE Plug-in stellt dazu den Ordner “Diagramm” zur Verfügung. Dieser Ordner erhält nun die zwei Dateien *example.er2uml* sowie *example.er2uml_diagramm*.

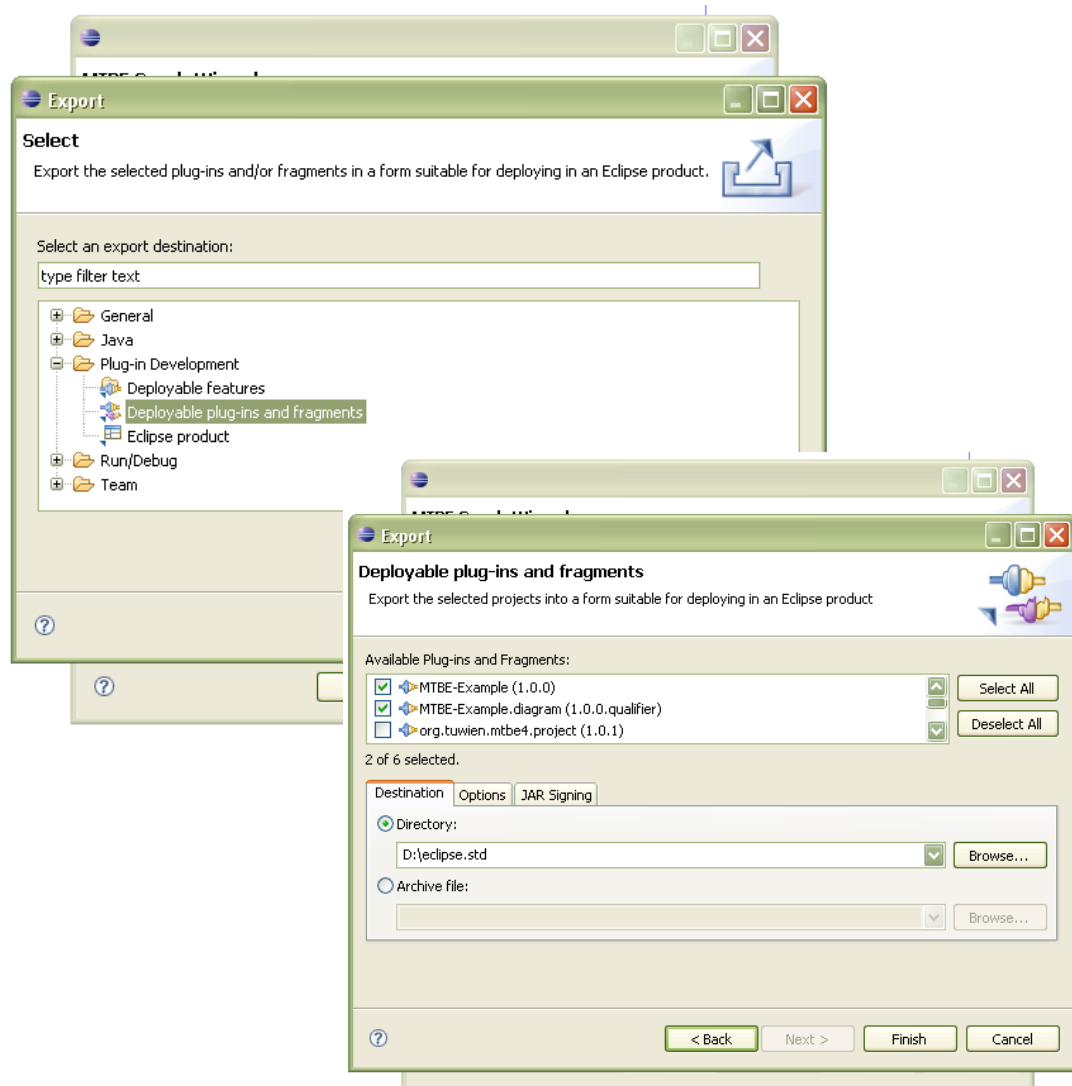


Abbildung 4.16: MTBE Wizard - Exportieren des neuen Diagramm Plug-in

Zum Zeichnen stehen dem Benutzer auf der rechten Seite die Werkzeug Paletten, gruppiert nach den zwei Ausgangsmodellen zur Verfügung, sowie ein Typ um Korrespondenzen einzuzichnen, das *Simple Mapping*. Wir zeichnen nun zwei einfache Diagramme, je eines in UML (Abbildung 4.17 A) und eines in ER (Abbildung 4.17 B). Wichtig ist es, dass alle für meine Transformation essentiellen Sprachkonzepte enthalten sind.

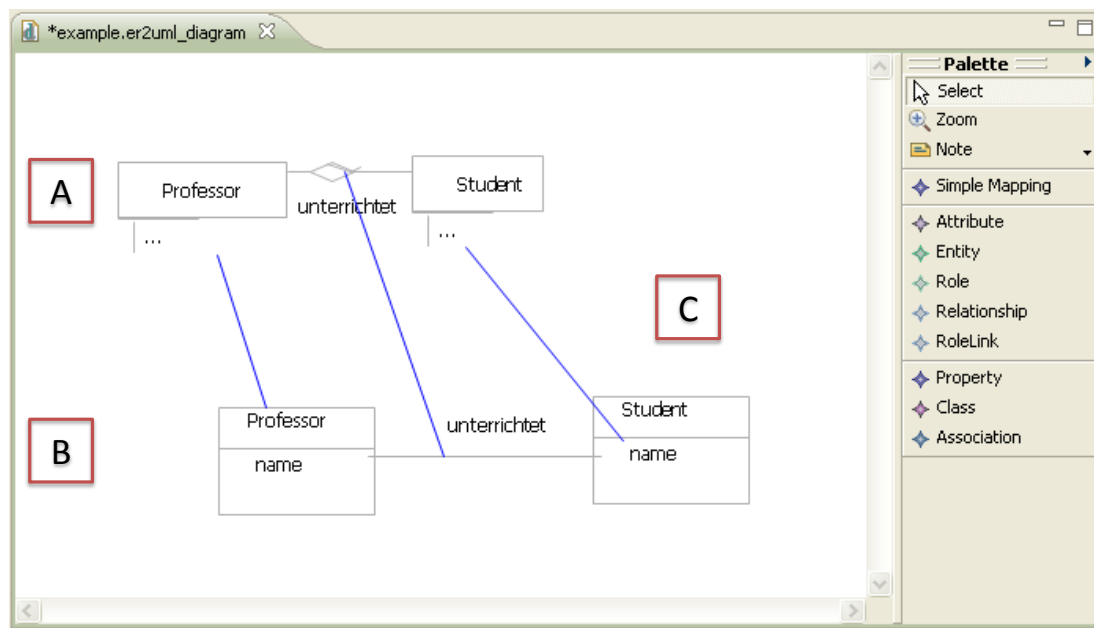


Abbildung 4.17: Der neue Diagrammeditor für beide Modelltypen (*example.er2uml_diagramm*)

4.5.5 Einzeichnen der Mappings

Wurden die zwei Modelle gezeichnet, kann nun begonnen werden Mappings zwischen ihnen einzuzichnen Abbildung 4.17 C. Es werden nun folgende Korrespondenzen gesetzt:

- *Klasse 2 Entity*,
- *Attribut 2 Property*
- *Relationship 2 Association*.

Dieses Beispiel beschränkt sich auf diese drei Mappings um diese Ausführung für Erläuterung einfach zu halten. Abbildung 4.18 stellt die zwei gezeichneten Modelle mit den drei Mappings als Baum dar. Neben den einzelnen Modellelementen beider Ausgangseditoren (4.18-A), sind im mit B markierten Bereich die drei von uns eingezeichneten Mappings gespeichert. Jedes dieser *Simple Mappings* enthält zwei *Properties*, welche die Korrespondenz zwischen den zwei Elementen darstellt.

4.5.6 Ausführen der Analyse

Auf Basis der Datei in Abbildung 4.18 wird nun die Analyse durchgeführt. Dazu muß nur mehr der “do reasoning” Button betätigt werden. Bevor die Analyse gestartet wird, erscheint noch ein Dialogfeld (Abbildung 4.19) über welches ausgewählt werden kann, welche Algorithmen auf das gezeichnete Diagramm aus-

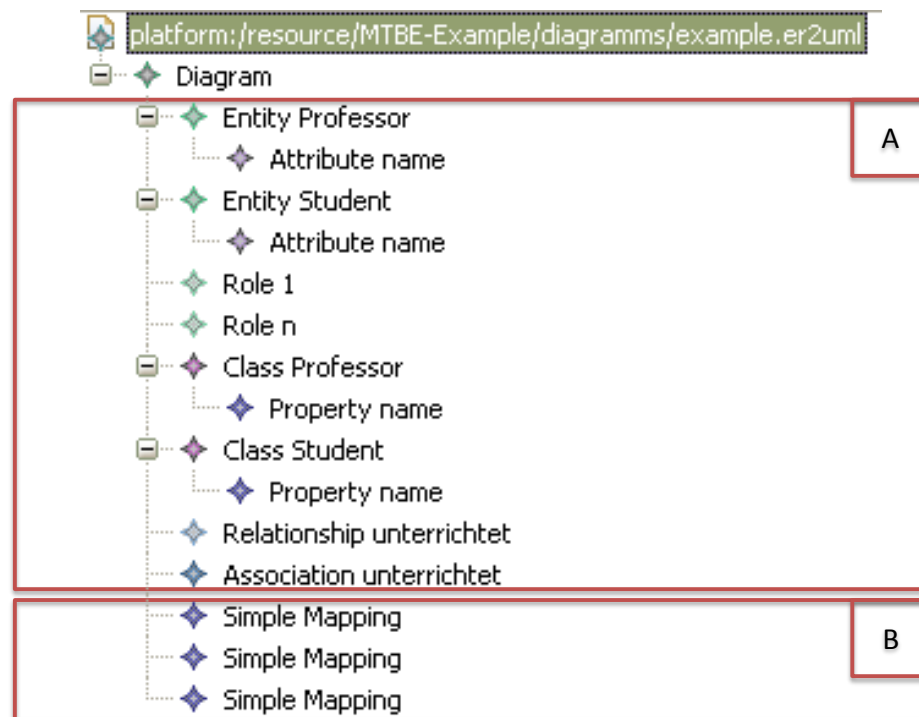
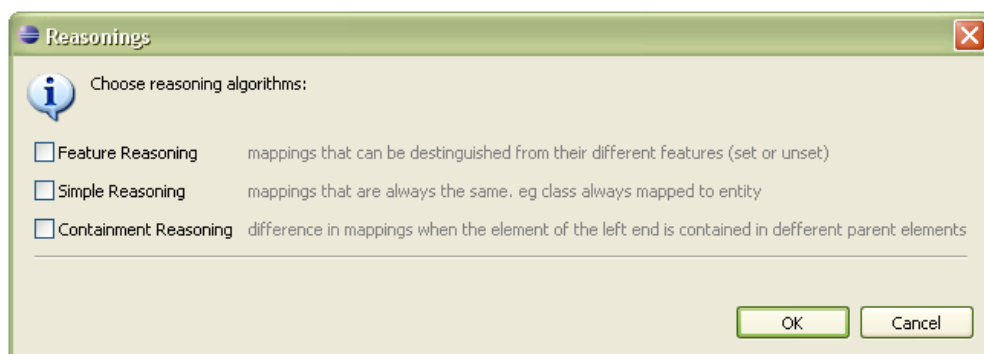
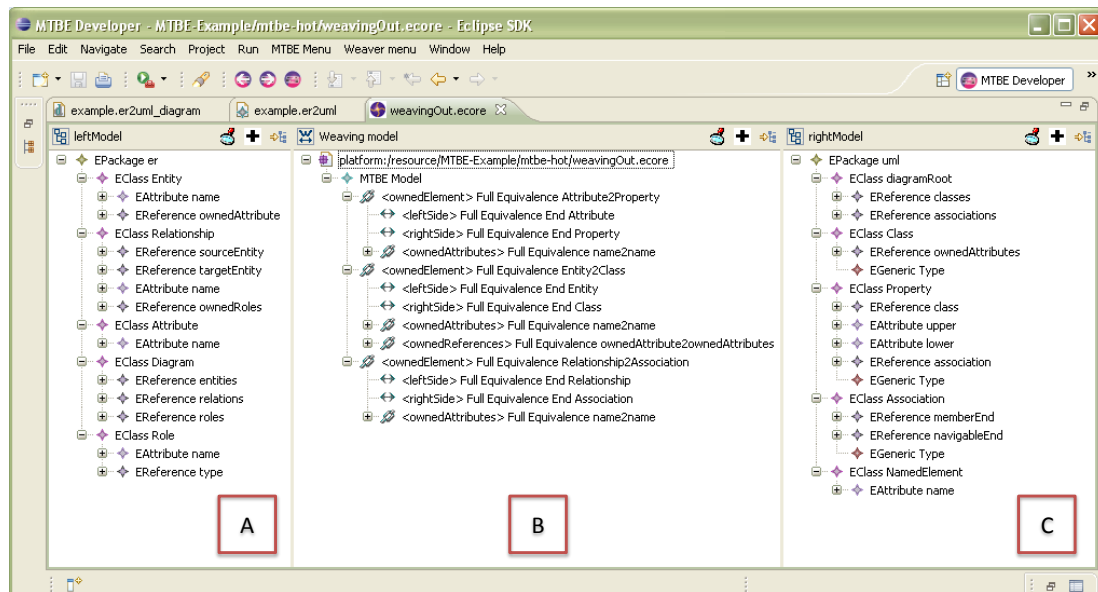
Abbildung 4.18: Unser Diagramm in Baumdarstellung (*example.er2uml*)

Abbildung 4.19: Dialogbox für die Auswahl der Algorithmen

geführt werden sollen. In der aktuellen Version des MTBE Plug-ins stehen die Analysealgorithmen *Simple Reasoning*, *Containment Reasoning* und *Feature Reasoning*. Für dieses Beispiel werden alle drei Algorithmen gewählt. Das Ergebnis der Analyse ist ein *Weaving Modell* zwischen den beiden Metamodellen.

Abbildung 4.20: Das generierte *Weaving Modell*

4.5.7 Weaving Modell

Das generierte *Weaving Modell* ist unter dem Namen “weavingOut.ecore” im Ordner *mtbe-hot* zu finden, in Abbildung 4.20 dargestellt im *Weaving Editor*. Links ist das Metamodell des ER Beispiel Diagrammes zu sehen (Abbildung 4.20 A) und rechts das des UML Diagrammes (Abbildung 4.20 C), in der Mitte befindet sich das aus den zuvor eingezeichneten Mappings generierte *Weaving Modell* (Abbildung 4.20 B).

Durch die Analysealgorithmen wurden folgende Äquivalenzen gefunden.

- Attribute2Property
- Entity2Class
- Relationship2Association

4.5.8 Generieren des ATL Codes

Der nächste Schritt beschreibt das Generieren des ATL Codes, je nach Benutzereingabe kann der Code für eine Transformation in die eine oder andere Richtung generiert werden.

- *ER2UML* Um den Regeln für eine Transformation von ER zu UML zu erhalten, wird der Button “do transformation right-left” betätigt. Es erscheint das Dialogfenster “Left2Right Transformation” in Abbildung 4.21 in welchem ein Modell für das im nächsten Schritt folgende Testen der

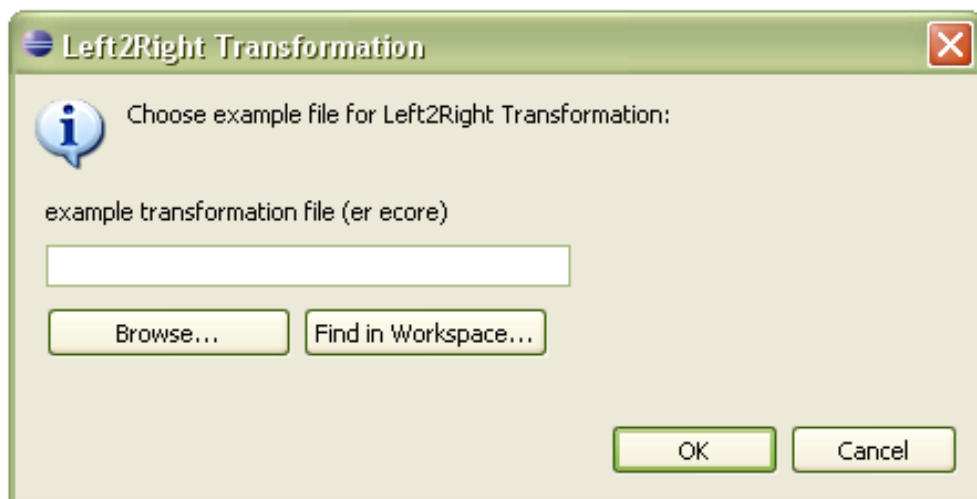


Abbildung 4.21: Dialogbox für die Auswahl eines Modells zum Testen der Transformation

Transformation gewählt werden kann. Nach betätigen des “Ok”-Buttons wird zuerst das Script *run.xml* ausgeführt, dieses generiert aus dem *Weaving Modell* die Transformationsregeln in Form eines ATL Codes. Dann wird das Script *transformation.xml* ausgeführt, welches eine Beispieltransformation mit dem gewählten Diagramm durchführt. In den beiden Fenstern unter dem gezeichneten Diagramm werden nun das Ausgangsmodell sowie das transformierte Modell angezeigt.

Folgendes Listing 4.16 zeigt den durch MTBE generierten ATL Code.

Listing 4.16: ATL Beispieltransformation UML2ER

```

1      module left2right;
2      create rightModel : rightMetaModel from leftModel :
        leftMetaModel;
3
4      rule Entity_2_Class {
5          from
6              entity : leftMetaModel!Entity
7          to
8              class : rightMetaModel!Class (
9                  name <- entity.name
10             )
11     }
12
13     rule Relationship_2_Association {
14         from
15             relationship : leftMetaModel!Relationship
16         to
17             association : rightMetaModel!Association (
18                 name <- relationship.name
19             )

```

```

20     }
21
22     rule Attribute_2_Property {
23         from
24             attribute : leftMetaModel!Attribute
25         to
26             property : rightMetaModel!Property (
27                 name <- attribute.name
28             )
29     }

```

- *UML2ER* Um den Regeln für eine Transformation von UML zu ER zu erhalten, muss der Button “do transformation left-right” gedrückt werden. Auch für diese Richtung kann wieder ein Modell zum Testen der Transformation gewählt werden.

Zuerst wird wiederum das Script *run.xml* ausgeführt, dann das Script *transformation.xml*. Der Unterschied zur ER2UML Transformation liegt lediglich darin, dass die Metamodelle vertauscht werden. In den beiden Fenstern unter dem Diagramm werden nun links das transformierte Modell und rechts daneben das Ausgangsmodell angezeigt.

Folgendes Listing 4.16 zeigt nun den durch MTBE generierten ATL Code, diesmal für die entgegengesetzte Transformationsrichtung.

Listing 4.17: ATL Beispieltransformation ER2UML

```

1     module left2right;
2     create rightModel : rightMetaModel from leftModel :
3         leftMetaModel;
4     rule Class_2_Entity {
5         from
6             class : rightMetaModel!Class
7         to
8             entity : leftMetaModel!Entity (
9                 name <- class.name )
10    }
11    rule Association_2_Relationship {
12        from
13            association : rightMetaModel!Association
14        to
15            relationship : leftMetaModel!Relationship (
16                name <- association.name )
17    }
18    rule Property_2_Attribute {
19        from
20            property : rightMetaModel!Property
21        to
22            attribute : leftMetaModel!Attribute (
23                name <- property.name )
24    }

```

4.5.9 Test der Transformation

Wurde eine der beiden *MTBE Perspektiven* ausgewählt, wird automatisch beim Ausführen des Befehles “do transformation left-right” beziehungsweise “do transformation right-left” eine Beispieltransformation ausgeführt. Die Transformation wird auf das zuvor (Abbildung 4.21) gewählte Modell angewandt, und dieses mit dem Ergebnis unter dem gezeichneten Diagramm angezeigt. Entsprechend dem iterativen Ansatz von *Model Transformation By-Example* könnte nun bei Bedarf Anpassungen des *Weaving Modell* erfolgen beziehungsweise weitere Mappings zwischen Modellen eingezeichnet werden.

4.6 Kritische Betrachtung

GMF Entwicklungsstatus

GMF ist ein noch sehr junges Projekt. Daher ist es noch starken Veränderungen unterworfen. Um eine möglichst einfache Benutzung zu gewährleisten wurden einige GMF und Eclipse interne Funktionen verwendet, für die nicht garantiert wird, dass sie in dieser Form in späteren Veröffentlichungen vorhanden sein werden. Dabei handelt es sich um den Zugriff auf die *TransformToGenmodelOperation* welche es ermöglicht aus einer *Mapping Definition* das entsprechende *.genmodel* zu erzeugen. Auch wenn diese Funktion in einer der nächsten Version von GMF in ein anderes Paket verschoben wird oder sich die Parameter ändern, ist es anzunehmen, dass es einfacher ist dies anzupassen als eine Änderung in GMF mit der eigenen Implementierung zu kompensieren. Auch der *ImportExportWizard* sollte normalerweise nicht programmatisch angesprochen werden. Trotzdem wurde er zugunsten einer einheitlichen Bedienung integriert.

Graphical Definition

Diese Implementierung erstellt nicht nur eine gemeinsame *Mapping Definition* und ein kombiniertes *Domain Model* sondern fügt auch die restlichen, für ein GMF Projekt nötigen Ressourcen zusammen. Dies hat den Vorteil dass die ursprünglichen Editoren nicht mehr zwangsweise installiert sein müssen. Da aber für die *Graphical Definition* eine gemeinsame *Figure Gallery* erstellt wird, ist zu beachten, dass es zu doppelten Namen und somit zu Problemen beim *GraphMerger* kommen kann. Dies ist nur durch eine Anpassung der beiden ursprünglichen *Graphical Definitions* zu verhindern. Es wäre zu überlegen in Zukunft diese Komponenten durch eine Referenz in die *Mapping Definition* einzubinden um dieses Problem zu lösen.

MTBE Perspektive

Die Vorschau der Beispieltransformation benötigt auf Grund von Usability-Aspekten drei beziehungsweise vier Editoren gleichzeitig. Theoretisch ist Eclipse auch dafür ausgerichtet mehrere Editoren nebeneinander und untereinander anzuzeigen. Im momentanen Entwicklungsstatus von Eclipse gibt es leider keine entsprechende API, die diese Funktion unterstützt, es wurde daher in diesem Prototyp auf das Simulieren eines “Drag and drop” Befehls zurückgegriffen, um die Fenster nach geforderten Wünschen anzuordnen. Auf diese Variante wurde zurückgegriffen um trotz Einschränkungen durch die Eclipse API die Vorteile der Darstellung mit mehreren Editoren zu demonstrieren.

Eigene Algorithmen

Um Entwicklern die Möglichkeit zu bieten Wissen über Domänen einzubringen wurde eine einfache Möglichkeit geschaffen den Analyseprozess zu erweitern. Dabei kann eigene Logik dazu beitragen aus den Mappings der Benutzer Verbindungen auf der Metaebene zu finden. Im vorliegenden Prototyp liegt das Hauptaugenmerk auf einem erweiterbaren Framework. Um Verbindungen auf Metaebene welche Bedingungen (*conditions*) beinhalten richtig zu beschreiben ist es in vielen Fällen notwendig neben Erweiterungsalgorithmen auch Änderungen in Codeabschnitten vorzunehmen welche das *Weaving Modell* erstellen. Dies ist einer der Punkte die in späteren Weiterentwicklungen adressiert werden sollten (siehe auch Abschnitt 5).

Komplexe Editoren

Der entwickelte Prototyp zeigt wie man aus zwei einfachen Editoren ein Plug-in generiert, welches Komponenten zur Erstellung und Analyse von Beispielmappings bereitstellt. Wie in jedem Softwareprojekt mussten aus Zeitgründen einige Funktionen unberücksichtigt bleiben. Das Ergebnis ist eine lauffähige Implementierung, welche die wichtigsten Eigenschaften von GMF berücksichtigt. Editoren die den vollen Funktionsumfang von GMF wie beispielsweise *FeatureInitializer* oder *Custom Decorations* verwenden, können bislang noch nicht eingesetzt werden.

Erweiterbarkeit

Die *Higher Order Transformation* Komponente wurde durch ein *Ant Script* realisiert. Auch diese Komponente unterliegt konstanten Erweiterungen. Das Framework verwendet den *AntRunner* um die nötigen Skripten

te anzustoßen. Ändern sich die Parameter müssen diese in der Klasse *org.twien.mtbe4.project.actions.TransformationAction* angepasst werden. Sollte sich dies als unzureichend herausstellen, insbesondere falls die Parameter vom Benutzer verändert werden sollen, wäre es von Vorteil diese Schnittstelle in einen Konfigurationsdialog auszulagern. Die Implementation ist eng mit den Möglichkeiten des *Weaving Modells* verknüpft. Anstatt eine weitere künstliche Schnittstelle einzuführen, welche nach Änderungen im *Weaving Modell* ohnehin adaptiert werden müsste, wurde das *Weaving Modell* direkt als Repräsentation der Korrespondenzen auf Metaebene gewählt. Es wird zwar Reflektion verwendet um die nötigen Elemente zu instanziiieren, gröbere Änderungen im Metamodell führen aber notwendigerweise zu Änderungen im Quelltext des Plug-ins. Der Prototyp stellt dem Benutzer ein sogenanntes *Simple Mapping* zur Verfügung mit dem sich Domäneelemente verknüpfen lassen. Dabei können jeweils zwei Elemente durch ein Mapping verbunden werden. Oft reicht dies aber nicht aus um die Bedeutung eines Mappings vollständig abzubilden. Zusätzlich sollte die vorhandene *Mapping Language* erweitert werden um zusätzlich $n : m$ Verbindungen zu ermöglichen oder Einschränkungen darzustellen. Beispielsweise wäre es von Vorteil *XOR* Bedingungen zu unterstützen. Zusätzliche Typen von Mappings könnten es ermöglichen sowohl graphisch als auch semantisch zwischen Mappings zu unterscheiden welche Domäneelemente betreffen und solchen welche die Bedeutung von Referenzen oder Attributen erfassen.

Kapitel 5

Ausblick

Im Zuge dieser Diplomarbeit wurde ein Prototyp für MTBE implementiert. Um das Programm in der Praxis vernünftig einsetzen zu können besteht noch Bedarf an einigen Erweiterungen. Vor allem der Analyse wurde noch viel zu wenig Aufmerksamkeit geschenkt, obgleich diese einer der Kernelemente des MTBE Ansatzes ist. Darum wurde für diesen Punkt eine einfache Erweiterungsmöglichkeit geschaffen. Nachfolgend sind einige, der unserer Meinung nach wichtigsten Aspekte, die eine sinnvolle Erweiterung dieses Prototyps darstellen.

Zusätzliche Algorithmen

In dieser Arbeit wurde ein Framework vorgestellt welches zwei unabhängige Editoren zusammenfügt und eine grundlegende Analyse von Mappings durchführen kann. Dies funktioniert für die beiden einfachen, in dieser Arbeit vorgestellten, Editoren. Von den vorhandenen Analysealgorithmen werden aber zurzeit nur recht einfache Zusammenhänge erkannt. Daher können durch den erwähnten Erweiterungsmechanismus zusätzliche Algorithmen hinzugefügt werden. Zusätzlich zu diesen kann auch der Analyseteil des Projektes erweitert werden, falls beispielsweise zusätzliche *Metamapping* Typen definiert werden, oder bestimmte Mappings ein von der Implementierung abweichendes Einfügen in das *Weaving Modell* erforderlich machen. Dies kann durch adaptieren der Klasse `reasonAction` bewerkstelligt werden, mit steigenden Anforderungen könnte dafür auch ein Extension Point geschaffen werden. Wir gehen davon aus, dass für eine verbesserte Analyse am ehesten das Untersuchen der Referenzen zielführend ist. Daher ist auch das *FeatureReasoning* grundsätzlich auf Erweiterbarkeit ausgelegt. Dieser findet zurzeit alle Referenzen, verwendet dabei aber für die Unterscheidung lediglich die Tatsache ob diese belegt sind. Vorstellbar ist aber auch die Analyse der referenzierten Typen, wofür der bestehende Algorithmus erweitert werden kann.

Mapping Language

Der Prototyp stellt dem Benutzer ein sogenanntes *Simple Mapping* zur Verfügung mit dem sich Domäneelemente verknüpfen lassen. Dabei können jeweils zwei Elemente durch ein Mapping verbunden werden. Oft reicht dies aber nicht aus um die Bedeutung eines Mappings vollständig abzubilden. Zusätzlich sollte die vorhandene *Mapping Language* erweitert werden um zusätzlich $n : m$ Verbindungen zu ermöglichen oder Einschränkungen darzustellen. Beispielsweise wäre es von Vorteil *XOR* Bedingungen zu unterstützen. Zusätzliche Typen von Mappings könnten es ermöglichen sowohl graphisch als auch semantisch zwischen Mappings zu unterscheiden welche Domäneelemente betreffen und solchen welche die Bedeutung von Referenzen oder Attributen erfassen.

Attribut- und Referenzanalyse

Zusätzlich zu den Mappings des Benutzers wird auch implizit vorhandene Information ausgewertet um das *Weaving Modell* zu erstellen. Dabei wird von diesem lediglich eine sehr einfache Analyse durchgeführt die sicherlich noch erweitert werden kann. Für Referenzen eines gemappten Elements des linken Metamodells, muss ebenfalls ein geeignetes Mapping vorliegen wobei für das rechte Ende dieser Korrespondenz lediglich eine Referenz im ursprünglichen rechten Metaelement vorliegen darf. Abbildung 5.1 zeigt ein Beispiel für welches die Analyse die Referenz auflösen kann. Nachdem die Analysealgorithmen die beiden Korrespondenzen *mapping1* und *mapping2* erfolgreich auf Korrespondenzen der Metamodellbene abstrahiert haben, wird ein *Weavingelement* für *Class* zu *Entity* erzeugt. Für die Referenz *ownedAttributes* der *Class* wurde während der Analyse eine Korrespondenz auf *Attribute* gefunden. Somit kann dem erstellten *Weavingelement* eine *owned Reference* angehängt werden. Würde in diesem Beispiel *mapping1* nicht existieren oder es gäbe eine weitere Korrespondenz welche das *Property* mit einem anderen Element als *Attribute* welches in der Entität enthalten ist verbindet, könnte diese Referenz nicht eindeutig zugeordnet werden. Für die Attribute wird zusätzlich noch deren Name überprüft, im angeführten Beispiel haben beide Attribute die Bezeichnung *name* und können somit zugeordnet werden. Für die Erweiterung dieser Analyse ist eine Erweiterung des Frameworks durch einen zusätzlichen *Extension Point* vorstellbar.

Ähnlich wie bei der Analyse der Mappings können spezialisierte Algorithmen die jeweiligen Metaelemente genauer analysieren und mit Hilfe von Informationen über die Domänen bessere Resultate erzielen. Zusätzlich sollten die Mappings

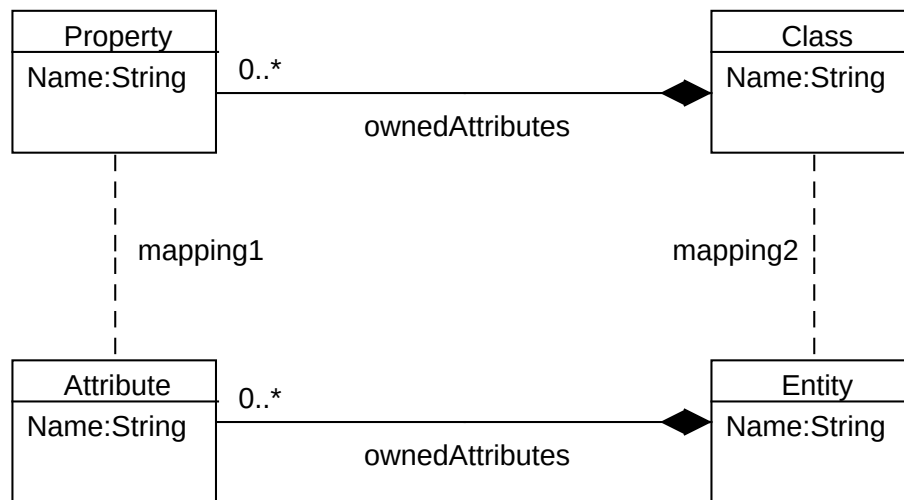


Abbildung 5.1: Beispiel Attribut- und Referenzanalyse

auf deren *Containment Features* geprüft werden um einerseits die bereitgestellte Information des Benutzers zu nutzen und um andererseits über das Ausschlussprinzip automatisch zusätzliche Mappings zu finden.

Kombination der Analyse

Durch die Konzentration der einzelnen Algorithmen auf eine klar abgegrenzte Aufgabe wird es manchmal dazu führen, dass Information nicht genutzt wird obwohl diese implizit vom Benutzer bereitgestellt wurde. Vor allem komplexere Bedingungen hängen von vielen unterschiedlichen Kriterien ab. Um diese Informationen zu extrahieren schlagen wir eine Kombinationsmöglichkeit der einzelnen Algorithmen vor. Die offensichtlichste Lösung wäre den einzelnen *doReasoning* Methoden eine Liste der zusätzlich zur Verfügung stehenden Analysemethoden bereitzustellen. Dazu wäre auch sinnvoll diese mit zusätzlichen Metainformationen anzureichern. Dabei könnte es sich um den Typ der gefundenen Mappings oder auch Informationen über den Autor usw. handeln.

Eine andere Lösung bestünde darin nach der Analyse einen weiteren Schritt einzuführen in welchem die Algorithmen explizit kombiniert werden. Der Vorteil dieser Methode bestünde darin, dass die sich einzelne Algorithmen nur auf ihre Aufgabe konzentrieren müssten. Es wäre auch durchaus vorstellbar hier eine weitere Schnittstelle zu schaffen die es erlaubt mittels Definitionen die Algorithmen zu kombinieren. Es könnte eine Sprache entwickelt werden, die zum Beispiel durch die Definition von *has FEATURE x AND containedIn y* die entsprechen-

den Mappings liefert welche dann analysiert werden können um zusätzliche nicht vom Benutzer direkt bereitgestellte Informationen zu gewinnen. Der Vorteil der Kombination liegt in der Aufgabenteilung. Dieser Algorithmus kann auf vorhandene Routinen zurückgreifen und kann beispielsweise auf die Analyse der *Features* verzichten. Dadurch ist es möglich mit verhältnismäßig geringem Aufwand Algorithmen zu entwickeln, die Wissen über die Domänen einbringen.

Vollständige GMF Unterstützung

Bei der Implementierung des Prototypen wurden alle GMF Konzepte berücksichtigt die von den beiden einfachen Editoren verwendet werden. Darüber hinaus existieren einige Möglichkeiten in GMF die vom *GraphMerger* noch ignoriert werden. Dazu zählen beispielsweise der *FeatureInitializer* mit dem es möglich ist bei der Erstellung von graphischen Elementen zusätzliche Domänenelemente zu erzeugen. In die *Tooling Definition* werden nur die Erstellungswerkzeuge übernommen und Einträge in eine Werkzeugleiste übergangen oder *Custom Creation*-Werkzeuge zur Zeit gänzlich ignoriert. Auch wird GMF ständig verbessert und erweitert und so werden sich wahrscheinlich in der nächsten Ausgabe bereits Neuerungen finden für die das MTBE-Projekt angepasst werden muss.

Unterstützung von Bedingungen

Wie bereits mehrfach erwähnt werden Bedingungen in Mappings nur bedingt unterstützt. Es existiert zwar im *Weaving Metamodel* in der *ConditionalEquivalence* die Referenz *hasFeature* welche aber nur auf den speziellen Fall anwendbar ist wenn sich aus den Mappings ergibt, dass eine Unterscheidung nur durch die Belegung eines *Features* möglich ist. Dies mag in unserem einfachen Beispiel der Fall sein, eine generelle Lösung fordert aber mehr Flexibilität. Ein möglicher Ansatz wäre die Unterstützung von OCL im *Weaving Modell*. Dabei müssten OCL Bedingungen von den einzelnen Analysealgorithmen erstellt werden und von der Analysekomponente ins Modell übernommen werden. Ein anderer Ansatz ist die Bereitstellung unterschiedlicher Metamodelle für das *Weaving* und ein modularer Aufbau der Erstellung dieses Modells durch weitere *Extension Points*.

Graphische Vorschau

In der MTBE-Perspektive wird eine Vorschau der Transformation angezeigt. Diese dient dem Benutzer als Indikator wie gut das Ergebnis durch die vorhandenen Mappings ist und ob noch zusätzliche Informationen benötigt werden oder ob das *Weaving Modell* zusätzlich angepasst werden muss. Der Prototyp verwendet hier-

zu den baumbasierten Editor und nicht den graphischen Editor. Da diese jedoch existieren müssen und die Lesbarkeit des Ergebnisses verbessert würde, wäre es von Vorteil die ursprünglichen Editoren zu verwenden. Bei der Transformation wird zwar lediglich ein Domänenmodell erzeugt, es ist aber höchstwahrscheinlich recht einfach möglich daraus eine entsprechende graphische Repräsentation zu generieren welche von einem der Editoren angezeigt werden kann.

Kapitel 6

Zusammenfassung

Die Implementierung des Prototyps zeigt neben den Potentialen dieses Ansatzes auch einige Probleme des Frameworks auf Basis vom *Graphical Modeling Frameworks* auf. Diese resultieren vielfach aus dem frühen Entwicklungsstadium dieses Projektes sowie des Design-Entscheidungen der Entwickler vor allem bei der Erstellung des GMF Metamodells und der Implementierung der Code-Generatoren. Trotzdem wurde gezeigt, dass die Implementierung des Frameworks mit den vorgestellten Technologien möglich ist. Weiterentwicklungen in diesen erweitern auch die Möglichkeiten des Frameworks.

Zusätzlich belegt die Implementierung, dass sich benutzerdefinierte Mappings auf Instanzebene zu Korrespondenzen auf Metamodellebene abstrahieren lassen. Dabei hängt die Qualität dieser Abstraktion stark vom eingebrachten Wissen über die eingesetzten Domänen ab. Es wurden Analysealgorithmen vorgestellt welche mit dem Ziel entwickelt wurden von diesem Wissen unabhängig zu sein um auf den allgemeinen Einsatzbereich des MTBE Ansatzes hinzuweisen. Dabei wurden sehr einfache Metamodelle für die UML und ER Domänen verwendet. Für komplexere Modelle wurde die Möglichkeit geschaffen den Analyseprozess über einen *Extension Point* zu erweitern und domänenspezifisches Wissen in den Vorgang einzubringen.

Zusätzlich wurde eine *Higher Order Transformation* Komponente eingebunden welche laufend Beispieltransformationen durchführt um einen Iterativen Prozess zu ermöglichen. Indem der Anwender ständig das Ergebnis abgleichen kann, müssen nur so viele Mappings bereitgestellt werden bis das gewünschte Ergebnis durch den Analyseprozess erreicht wird.

Um das das MTBE Plug-in tatsächlich in der Praxis vernünftig einsetzen zu können besteht auf jeden Fall noch Bedarf an einigen Erweiterungen. Einige Aspekte wurden ausgeklammert und Andere nur oberflächlich behandelt. Vor allem der Analyse wurde noch viel zu wenig Aufmerksamkeit geschenkt, obgleich

diese einer der Kernelemente des MTBE Ansatzes ist. Darum wurde speziell dafür eine einfache Erweiterungsmöglichkeit geschaffen. Aber auch für die Komponenten *Mapping Language* und *Higher Order Transformation* bieten wir mögliche Schnittstellen zur Erweiterung.

Neben dem MTBE Ansatz der *Business Infomatics Group* wurde auch ein ähnlicher Ansatz von Dániel Varró [48] vorgestellt. Der Unterschied zwischen diesen Ansätzen ist, dass Varrós Ansatz auf der abstrakten Ebene ansetzt, im Gegenteil dazu arbeitet MTBE auf der konkreten Ebene.

Zu unserer MTBE Implementierung kann festgestellt werden, dass der gewählte Ansatz durchaus das Potential besitzt weiterentwickelt zu werden. Es ist unter anderem denkbar zusätzliche Möglichkeiten des GMF Frameworks zu unterstützen, um schlussendlich auch mittels weiteren Analysealgorithmen mehrere vollständige Editoren anzubieten.

Anhang A

Anhang

A.1 EMF ER Metamodell

Listing A.1: Annotiertes Interface - Diagram

```

1 package org.tuwien.mtbe.er;
2 import org.eclipse.emf.common.util.EList;
3 import org.eclipse.emf.ecore.EObject;
4 /**
5  * @model
6  */
7 public interface Diagram extends EObject {
8     /**
9      * @model containment="true"
10     */
11     public EList<Entity> getEntities();
12     /**
13      * @model containment="true"
14     */
15     public EList<Relationship> getRelationships();
16     /**
17      * @model containment="true"
18     */
19     public EList<Role> getRoles();
20 }

```

Listing A.2: Annotiertes Interface - Entity

```

1 package org.tuwien.mtbe.er;
2 import java.util.List;
3 import org.eclipse.emf.ecore.EObject;
4 /** @model */
5 public interface Entity extends EObject {
6
7     /** @model required="true" */
8     public String getName();
9
10    /** @model type="Attribute" containment="true" */
11    public List getOwnedAttributes();
12 }

```

Listing A.3: Annotiertes Interface - Relationship

```

1 package org.tuwien.mtbe.er;
2 import org.eclipse.emf.common.util.EList;
3 import org.eclipse.emf.ecore.EObject;
4 /**
5  * @model
6  */
7 public interface Relationship extends EObject {
8     /**
9      * @model required="true"
10     */
11     public String getName();
12     /**
13      * @model required="true"
14     */
15     public Entity getSourceEntity();
16     /**
17      * @model required="true"
18     */
19     public Entity getTargetEntity();
20     /**
21      * @model lower="2" upper="2"
22     */
23     public EList<Role> getOwnedRoles();
24 }

```

Listing A.4: Annotiertes Interface - Role

```

1 package org.tuwien.mtbe.er;
2 import org.eclipse.emf.ecore.EObject;
3 /**
4  * @model
5  */
6 public interface Role extends EObject {
7     /**
8      * @model required="true"
9     */
10     public String getName();
11     Entity getType();
12     void setType(Entity value);
13 }

```

Listing A.5: Annotiertes Interface - Attribute

```

1 package org.tuwien.mtbe.er;
2 import org.eclipse.emf.ecore.EObject;
3 /**
4  * @model
5  */
6 public interface Attribute extends EObject {
7     /**
8      * @model required="true"
9     */
10
11     public String getName();
12 }

```

Listing A.6: ER Metamodell als XML Schema

```

1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <xsd:schema xmlns="http://org.tuwien.mtbe.er.ecore" xmlns:xsd="
    http://www.w3.org/2001/XMLSchema" targetNamespace="http://org.
    tuwien.mtbe.er.ecore">
3    <xsd:complexType name="Diagram">
4      <xsd:sequence>
5        <xsd:element maxOccurs="unbounded" minOccurs="0" name="
            entities" type="Entity"/>
6        <xsd:element maxOccurs="unbounded" minOccurs="0" name="
            relationships" type="Relationship"/>
7        <xsd:element maxOccurs="unbounded" minOccurs="0" name="
            roles" type="Role"/>
8      </xsd:sequence>
9    </xsd:complexType>
10   <xsd:complexType name="Entity">
11     <xsd:sequence>
12       <xsd:element maxOccurs="1" minOccurs="1" name="name" type="
            xsd:string"/>
13       <xsd:element maxOccurs="unbounded" minOccurs="0" name="
            ownedAttributes" type="Attribute"/>
14     </xsd:sequence>
15   </xsd:complexType>
16   <xsd:complexType name="Attribute">
17     <xsd:attribute name="name" type="xsd:string" use="
            required"/>
18   </xsd:complexType>
19   <xsd:complexType name="Relationship">
20     <xsd:sequence>
21       <xsd:element name="name" type="xsd:string"/>
22       <xsd:element maxOccurs="1" minOccurs="1" name="
            sourceEntity" type="Entity"/>
23       <xsd:element maxOccurs="1" minOccurs="1" name="
            targetEntity" type="Entity"/>
24       <xsd:element maxOccurs="2" minOccurs="2" name="
            ownedRoles" type="Role"/>
25     </xsd:sequence>
26   </xsd:complexType>
27   <xsd:complexType name="Role">
28     <xsd:sequence>
29       <xsd:element name="name" type="xsd:string"/>
30       <xsd:element maxOccurs="1" minOccurs="0" name="type"
            type="Entity"/>
31     </xsd:sequence>
32   </xsd:complexType>
33 </xsd:schema>

```

A.2 GMF und weitere Ausschnitte

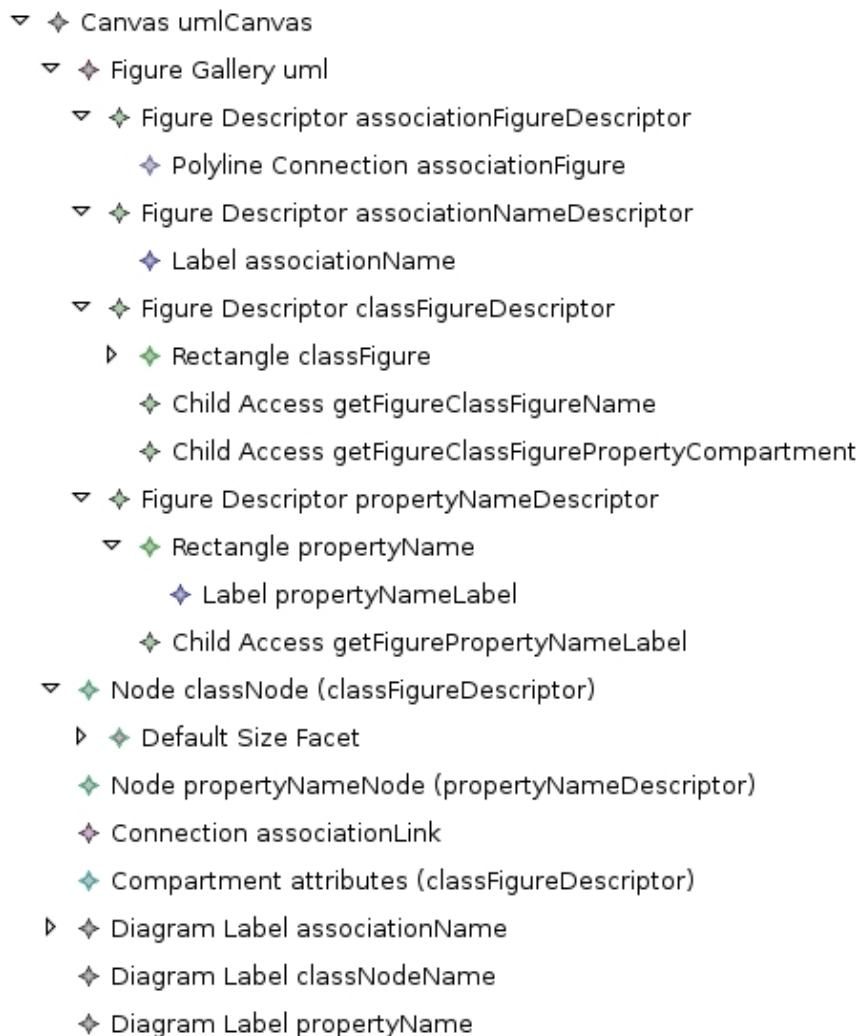


Abbildung A.1: Graphical Definition für UML Editor

Listing A.7: amw.prop-File für den Weaving Editor

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <weaver>
3 <weaving>
4 <relative-path>/test04/mtbe-hot/weavingOut.ecore</relative-path>
5 <panel>TransformationWeavingPanelExtension</panel>
6 <wmodel>Model</wmodel>
7 </weaving>
8 <metamodels>
9 <plugin/>
10 <local/>
11 <complete>/test04/mtbe-hot/metamodel/weavingmodel.ecore</complete>
12 </metamodels>
13 <woven id="0">
14 <model-relative-path>null</model-relative-path>
15 <metamodel-relative-path>/test04/mtbe-hot/metamodel/mmA.ecore</
    metamodel-relative-path>

```

```

16 <reference>leftModel</reference>
17 <title>leftModel</title>
18 <panel>DefaultWovenPanelExtension</panel>
19 <wmodelref>ModelRef</wmodelref>
20 </woven>
21 <woven id="1">
22 <model-relative-path>null</model-relative-path>
23 <metamodel-relative-path>/test04/mtbe-hot/metamodel/mmB.ecore</
    metamodel-relative-path>
24 <reference>rightModel</reference>
25 <title>rightModel</title>
26 <panel>DefaultWovenPanelExtension</panel>
27 <wmodelref>ModelRef</wmodelref>
28 </woven>
29 </weaver>

```

Listing A.8: run.xml Script

```

1 <project name="Execute_MTBE_HOT" default="AMW2ATL_model">
2   <!-- the properties are set outside by the caller of this script,
      in this example AntRunner in the Reasoning.java class -->
3   <!--
4   <property name="sourcePath" value="/org.tuwien.mtbe.hot/metamodel/"
      />
5   <property name="workingPath" value="/org.tuwien.mtbe.hot/" />
6   <property name="targetPath" value="/org.tuwien.mtbe.hot/model/" />
7   <property name="transformationPath" value="/org.tuwien.mtbe.hot/" />
8   -->
9   <target name="AMW2ATL_model">
10     <am3.loadModel modelHandler="EMF" name="left_model" metamodel="
        MOF" path="${metaPath}${leftIs}..ecore" />
11     <am3.loadModel modelHandler="EMF" name="right_model" metamodel="
        MOF" path="${metaPath}${rightIs}..ecore" />
12     <am3.loadModel modelHandler="EMF" name="ATL" metamodel="MOF"
        nsuri="http://www.eclipse.org/gmt/2005/ATL" />
13     <am3.loadModel modelHandler="EMF" name="AMW" metamodel="MOF"
        path="${metaPath}weavingmodel.ecore" />
14     <am3.loadModel modelHandler="EMF" name="amw_model" metamodel="
        AMW" path="${hotPath}weavingOut.ecore" />
15     <echo message="generating ATL_model from weaving_model" />
16     <!-- Transform AMW into ATL - OUT : ATL from IN : AMW, source:
        MOF, target: MOF; -->
17     <am3.atl path="${hotPath}hot/amw2atl_mtbe_2.atl">
18       <inModel name="MOF" model="%EMF" />
19       <inModel name="AMW" model="AMW" />
20       <inModel name="ATL" model="ATL" />
21       <inModel name="left" model="left_model" />
22       <inModel name="right" model="right_model" />
23       <inModel name="IN" model="amw_model" />
24       <outModel name="OUT" model="ATL_OUT" metamodel="ATL" />
25     </am3.atl>
26     <am3.saveModel model="ATL_OUT" path="${hotPath}atlmodel.ecore"
      />
27     <antcall target="extract" inheritall="true" inheritrefs="true"/
      >
28   </target>
29   <target name="extract">
30     <property name="source" value="${hotPath}atlmodel.ecore"/>

```

```

31     <property name="target" value="\${hotPath}atlmodel.atl"/>
32     <am3.loadModel modelHandler="EMF" name="IN" metamodel="ATL"
        path="\${source}" />
33     <am3.loadModel modelHandler="EMF" name="TCS" metamodel="MOF"
        path="\${metaPath}TCS.ecore" />
34     <am3.loadModel modelHandler="EMF" name="ATL.tcs" metamodel="TCS"
        " path="\${metaPath}ATL-0.2-TCS.ecore">
35     </am3.loadModel>
36     <am3.saveModel model="IN" path="\${target}">
37         <extractor name="ebnf">
38             <param name="format" value="ATL.tcs"/>
39         </extractor>
40     </am3.saveModel>
41 </target>
42 </project>

```

Listing A.9: transformation.xml Script

```

1 <project name="Execute_MTB_EHOT" default="TRANSFORMATION">
2     <target name="TRANSFORMATION">
3         <am3.loadModel modelHandler="EMF" name="left_model" metamodel="
            MOF" path="\${metaPath}\${leftIs}..ecore" />
4         <am3.loadModel modelHandler="EMF" name="right_model" metamodel="
            MOF" path="\${metaPath}\${rightIs}..ecore" />
5         <am3.loadModel modelHandler="EMF" name="IN" metamodel="
            left_model" path="\${hotPath}preview.ecore" />
6         <am3.atl path="\${hotPath}atlmodel.atl">
7             <inModel name="leftMetaModel" model="left_model" />
8             <inModel name="rightMetaModel" model="right_model" />
9             <inModel name="leftModel" model="IN" />
10            <outModel name="OUT" model="OUT" metamodel="right_model" />
11        </am3.atl>
12        <am3.saveModel model="OUT" path="\${hotPath}transformationResult
            ..ecore" />
13    </target>
14 </project>

```

Literaturverzeichnis

- [1] ANNEKE, KLEPPE, WARMER JOS und BAST WIM: *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley Professional, April 2003.
- [2] ATLAS GROUP, *ATL User Manual*.
<http://www.eclipse.org/m2m/atl/doc/>, letzter Besuch: 2007-12-28.
- [3] BECKER, KARIN, MICHELLE DE OLIVEIRA CARDOSO, CAREN MORAES NICHELE und MICHELE FRIGHETTO: *Mail-By-Example: A Visual Query Interface for Email Management*. In: *International Conference on Advanced Visual Interfaces*, Seiten 280–281, Palermo, 2000.
- [4] BUDINSKY, FRANK, DAVID STEINBERG, ED MERKS, RAYMOND ELLERSICK und TIMOTHY J. GROSE: *Eclipse Modeling Framework*. Addison-Wesley, 2004.
- [5] CYPHER, ALLEN: *Watch What I Do – Programming by Demonstration*. MIT Press, Cambridge, USA, 1993.
- [6] CZARNECKI, KRZYSZTOF und S. HELSEN: *Feature-based Survey of Model Transformation Approaches*. IBM Syst. J., 45, 2006.
- [7] ECLIPSE WIKI, AMW. <http://wiki.eclipse.org/AMW>, letzter Besuch: 2008-02-02.
- [8] ECLIPSE.ORG, *Eclipse.org Home*. <http://www.eclipse.org>, letzter Besuch: 2007-10-09.
- [9] ERICH, GAMMA, HELM RICHARD, JOHNSON RALPH und VLISSIDES JOHN: *Design patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [10] FABRO, MARCOS DIDONET DEL, JEAN BEZIVIN, FREDERIC JOUAULT, ERWAN BRETON und GUILLAUME GUELTAS: *AMW: A Generic Model Weaver*. In: *Proceedings of the 1ère Journée sur l’Ingénierie Dirigée par les Modèles (IDM05)*, Paris, France, 2005.

- [11] FRANK, BUDINSKY, STEINBERG DAVE, MERKS ED, ELLERSICK RAY und GROSE TIMOTHY J.: *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley Professional, August 2003.
 - [12] FRANKEL, DAVID S.: *Model Driven Architecture*. Wiley Publishing, 2003.
 - [13] GAMMA, ERICH und KENT BECK: *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addison Wesley Professional, 2003.
 - [14] GERBER, ANNA und KERRY RAYMOND: *MOF to EMF: There and Back Again*. In: *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, Seiten 60–64, Anaheim, California, 2003.
 - [15] GRUHN, VOLKER, DANIEL PIEPER und CARSTEN RÖTTGERS: *MDA. Effektives Softwareengineering mit UML2 und Eclipse*. Springer-Verlag, Berlin Heidelberg, 2006.
 - [16] HITZ, MARTIN und GERTI KAPPEL: *UML@Work*. dpunkt-Verlag, 3. Auflage, 2005.
 - [17] INRIA TEAM ATLAS, *Project-ATLAS:Model Management*. <http://ralyx.inria.fr/2005/Raweb/atlas/uid31.html>, letzter Besuch: 2007-12-28.
 - [18] LECHNER, STEPHAN und MICHAEL SCHREFL: *Transformers-by-example: Pushing Reuse in Conceptual Web Application Modelling*. In: *Proceedings of the 2004 ACM Symposium on Applied Computing*, Seiten 1654–1661, Nicosia, Cyprus, 2004.
 - [19] LEE, DONGWON, WENLEI MAO, HENRY CHIU und WESLEY W. CHU: *Designing Triggers with Trigger-By-Example*. Knowledge and Information Systems, 7, 2005.
 - [20] LITTLE, SUZANNE und JANE HUNTER: *Rules-By-Example - A Novel Approach to Semantic Indexing and Querying of Images*. In: *International Semantic Web Conference*, Seiten 534–548, Hiroshima, Japan, 2004.
 - [21] MENS, TOM und PIETER VAN GORP: *A Taxonomy of Model Transformation*. Electr. Notes Theor. Comput. Sci., 152, 2006.
 - [22] MOORE, BILL, DAVID DEAN, ANNA GERBER, GUNNAR WAGENKNECHT und PHILIPPE VANDERHEYDEN: *Eclipse Development - Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM, 2004.
-

-
- [23] MYERS, BRAD A.: *Visual Programming, Programming by Example, and Program Visualization: a Taxonomy*. SIGCHI Bull., 17, 1986.
 - [24] OBJECT MANAGEMENT GROUP (OMG): *MDA Guide version 1.0.1*, Juni 2003.
 - [25] OBJECT MANAGEMENT GROUP (OMG): *MOF Core Specification, v2.0*, January 2006.
 - [26] OBJECT MANAGEMENT GROUP (OMG): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, Juli 2007.
 - [27] OBJECT MANAGEMENT GROUP (OMG), *OMG*. <http://www.omg.org/>, letzter Besuch: 2007-10-07.
 - [28] OBJECT MANAGEMENT GROUP (OMG), *OMG's MetaObject Facility*. <http://www.omg.org/mof/>, letzter Besuch: 2007-10-07.
 - [29] OBJECT MANAGEMENT GROUP (OMG): *Unified Modeling Language: Superstructure*, February 2007.
 - [30] OMG, OBJECT MANAGEMENT GROUP: *Meta Object Facility (MOF) Core Specification*, Jänner 2006.
 - [31] OMONDO.COM, *The Modeling Eclipse UML Model Driven Tool*. <http://www.omondo.com>, letzter Besuch: 2007-10-09.
 - [32] ÖZSOYOGLU, GÜLTEKIN und HUAQING WANG: *Example-Based Graphical Database Query Languages*. Computer, 26(5):25–38, 1993.
 - [33] RAMAKRISHNAN, RAGHU und JOHANNES GEHRKE: *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 2002.
 - [34] REITSMA, JAAP, *GMF Tutorial*. http://wiki.eclipse.org/index.php/GMF_Tutorial, letzter Besuch: 2008-02-02.
 - [35] SCARPINO, MATTHEW, STEPHEN HOLDER, STANFORD NG und LAURENT MIHALKOVIC: *SWT/JFace in Action: GUI Design with Eclipse 3.0 (In Action series)*. Manning Publications Co., Greenwich, USA, 2004.
 - [36] SCHMIDT, DOUGLAS C.: *Guest Editor's Introduction: Model-Driven Engineering*. IEEE Computer, 39, 2006.
-

- [37] SMITH, DAVID CANFIELD: *Pygmalion: A Creative Programming Environment*. Doktorarbeit, Stanford University, Stanford, CA, USA, 1975.
 - [38] STAHL, THOMAS, MARKUS VÖLTER, SVEN EFFTINGE und ARNO HAASE: *Modellgetriebene Softwareentwicklung*. dpunkt.verlag, 2007.
 - [39] STROMMER, MICHAEL, MARION MURZEK und MANUEL WIMMER: *Applying Model Transformation By-Example on Business Process Modeling Languages*. In: *Advances in Conceptual Modeling – Foundations and Applications*, Seiten 116–125, Auckland, New Zealand, 2007.
 - [40] STROMMER, MICHAEL und MANUEL WIMMER: *A Framework for Model Transformation By-Example: Concepts and Tool Support*. In: *Proceedings of the 46th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'08)*, Zurich, Switzerland, July 2008.
 - [41] TANSEL, A. U., M. E. ARKUN und G. ÖSOYOĞLU: *Time-by-Example Query Language for Historical Databases*. IEEE Trans. Softw. Eng., 15, 1989.
 - [42] THE ECLIPSE FOUNDATION, *The Eclipse Modeling Framework (EMF) Overview*. <http://dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html>, letzter Besuch: 2007-10-03.
 - [43] THE ECLIPSE FOUNDATION, *Atlas Model Weaver (AMW)*. <http://www.eclipse.org/gmt/amw/>, letzter Besuch: 2007-12-28.
 - [44] THE ECLIPSE FOUNDATION, *Eclipse Graphical Modeling Framework Project (GMF)*. <http://www.eclipse.org/modeling/gmf/>, letzter Besuch: 2007-12-12.
 - [45] THE ECLIPSE FOUNDATION, *Eclipse Modeling Framework Project (EMF)*. <http://www.eclipse.org/modeling/emf/>, letzter Besuch: 2007-12-12.
 - [46] THE ECLIPSE FOUNDATION, *Graphical Editing Framework (GEF)*. <http://www.eclipse.org/gef/>, letzter Besuch: 2007-12-12.
 - [47] THE ECLIPSE FOUNDATION, *UML2 Projekt*. <http://www.eclipse.org/uml2/>, letzter Besuch: 2008-02-04.
 - [48] VARRÓ, DANIEL: *Model Transformation by Example*. In: *Lecture Notes in Computer Science : Model Driven Engineering Languages and Systems*, Seiten 410–424, Genova, Italy, 2006.
-

-
- [49] WIMMER, MANUEL, MICHAEL STROMMER, HORST KARGL und GERHARD KRAMLER: *Towards Model Transformation Generation By-Example*. In: *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, Seite 285b, Hawaii, USA, 2007.
- [50] ZLOOF, MOSHÉ M.: *Query-by-example: The Invocation and Definition of Tables and Forms*. In: *VLDB '75: Proceedings of the 1st International Conference on Very Large Data Bases*, Seiten 324–343, Framingham, Massachusetts, 1975.
- [51] ZLOOF, MOSHÉ M.: *Query-by-Example: A Data Base Language*. IBM Systems Journal, 16, 1977.
-