Lutz Bichler

Codegeneratoren für MOF-basierte Modellierungssprachen

Codegeneratoren für MOF-basierte Modellierungssprachen

Von der Fakultät für Informatik
der Universität der Bundeswehr München
zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften
genehmigte

DISSERTATION

von

Diplom-Informatiker
Lutz Bichler

1. Gutachter: Prof. Dr. Andy Schürr
2. Gutachter: Prof. Dr. Mark Minas

Datum des Kolloquiums: 26.11.2004

Kurzfassung

Umfang und Funktionalität eingesetzter Softwaresysteme nehmen beständig zu. Zudem wird Software immer häufiger in Bereichen verwendet, die spezielle Anforderungen an die Software-Qualität stellen. Modellbasierte Softwareentwicklung wird als viel versprechender Ansatz angesehen, die daraus entstehenden Anforderungen zu bewältigen. In den vergangenen Jahren wurde eine Vielzahl modellbasierter Ansätze zur Softwareentwicklung vorgestellt. Viele dieser Ansätze verwenden die *Unified Modeling Language* (UML), um die Modelle zu notieren. In letzter Zeit gewinnt außerdem die *Model Driven Architecture* (MDA) zunehmend an Bedeutung, die in erheblich stärkerem Maße als andere Ansätze auf die automatische Abbildung von Modellen auf den Code der Zielplattform setzt. Domänenspezifische Anpassungen und Erweiterungen von Modellierungssprachen wie der UML sind die Voraussetzung für das Erreichen der Ziele der MDA, weil sich die Modelle verschiedener Anwendungsdomänen meist nur durch angepasste Modellelemente effizient beschreiben lassen. Eine Möglichkeit, entsprechende Anpassungen und Erweiterungen für die UML zu definieren, ist die Metamodellierungssprache *Meta Object Facility* (MOF), die u. a. zur Definition der UML verwendet wird.

In dieser Arbeit wird ein Baukasten beschrieben, der die Entwicklung von Codegeneratoren für Modellierungssprachen erleichtert, die durch ein MOF-Modell definiert sind. Im Unterschied zu anderen Ansätzen zur Entwicklung von Codegeneratoren für Modellierungssprachen können nicht nur die Abbildungen der Elemente der Eingabesprache auf die zu erzeugenden Artefakte, sondern auch die Eingabesprache selbst durch den Anwender festgelegt werden. Auf diese Weise wird eine erheblich flexiblere Unterstützung der automatischen Codeerzeugung aus Modellen erreicht. Da der Baukasten mit Ausnahme der Abbildungen zwischen den Elementen der zu unterstützenden Modellierungssprache und den Artefakten der gewünschten Zielsprache alle Komponenten eines Codegenerators bereitstellt, bzw. generieren kann, ist die Aufgabe des Anwenders auf ein Minimum beschränkt. Der Anwender muss zur Implementierung eines Codegenerators für eine MOF-basierte Modellierungssprache lediglich die Abbildungen zwischen den Sprachelementen und Artefakten der gewünschten Implementierungstechnologie definieren bzw. implementieren.

Der Baukasten basiert auf Standardtechnologien und verwendet ausschließlich die Mechanismen, die die MOF bereitstellt. Er kann daher in nahezu jeder objektorientierten Implementierungstechnologie realisiert werden. Die Realisierung im Rahmen dieser Arbeit erfolgte durch die Programmiersprache *Java*, die Dokumentenbeschreibungssprache *Extensible Markup Language* (XML) und die Transformationssprache *Extensible Stylesheet Language Transformations* (XSLT). Die Verwendung standardisierter Technologien erleichtert den Umgang mit den Elementen des MOmo-Baukastens, weil die verwendeten Technologien einer großen Zahl von Entwicklern bereits bekannt sind, so dass der Einarbeitungsaufwand minimiert werden kann.

Inhaltsverzeichnis

Kurzfassung **i**

1 Einleitung **1**
1.1 Motivation 1
1.2 Problemstellung und Lösungsansatz 2
1.3 Organisation der Arbeit 5

2 Modellbasierte Softwareentwicklung **7**
2.1 Modelle 8
2.1.1 Klassifikation nach Anwendungsbereichen 8
2.1.2 Klassifikation nach Eigenschaften 10
2.1.3 Begriffsdefinition 10
2.2 Metamodelle 12
2.3 Modellierung 16
2.3.1 Rational Unified Process 17
2.3.2 Model Driven Architecture 20
2.4 Zusammenfassung 23

3 Literaturüberblick **25**
3.1 Klassifikation 26
3.2 Modelltransformation 27
3.2.1 Direkte Programmierung 28
3.2.2 Deklarative Ansätze 29
3.2.3 Graphtransformationsbasierte Ansätze 30

3.2.4 Weitere Ansätze . . . 33
3.2.5 Bewertung . . . 35
3.3 Codegenerierung . . . 36
3.3.1 Direkte Programmierung . . . 36
3.3.2 Schablonenbasierte Ansätze . . . 36
3.3.3 Bewertung . . . 38
3.4 Zusammenfassung . . . 39

4 Der MOmo-Baukasten 41
4.1 Anforderungsdefinition . . . 42
4.1.1 Anwendungsfallkategorien . . . 42
4.1.2 Anforderungen . . . 46
4.2 Entwurf . . . 48
4.2.1 Aufbau von MOmo-Generatoren . . . 50
4.2.2 Schablonenablaufumgebung . . . 51
4.2.3 Hinzufügen optionaler Komponenten . . . 52
4.3 Beispiel . . . 54
4.4 Zusammenfassung . . . 63

5 Meta Object Facility 65
5.1 Metadatenarchitektur . . . 66
5.2 Struktur . . . 68
5.3 Modellierungselemente . . . 70
5.3.1 Klassen . . . 71
5.3.2 Assoziationen . . . 78
5.3.3 Datentypen . . . 80
5.3.4 Pakete . . . 82
5.3.5 Bedingungen . . . 89
5.4 Zusammenfassung . . . 90

6 Extensible Markup Language 93
6.1 Bestandteile . . . 94
6.1.1 Elemente und Attribute . . . 94
6.1.2 Namensräume . . . 97
6.1.3 Referenzen . . . 99
6.2 Dokumentklassen . . . 102

6.2.1 Aufbau eines XML-Schemas . . . 103
6.2.2 Elementdeklarationen . . . 104
6.2.3 Typdefinitionen . . . 107
6.3 Zusammenfassung . . . 111

7 XML Metadata Interchange 113
7.1 Schemata . . . 116
7.1.1 Aufbau . . . 116
7.2 Dokumente . . . 124
7.2.1 Aufbau . . . 124
7.2.2 Produktion . . . 126
7.3 Zusammenfassung . . . 129

8 Generierte Komponenten 131
8.1 Aufbau . . . 132
8.2 Modellabhängige Bestandteile . . . 133
8.2.1 Pakete . . . 134
8.2.2 Klassen . . . 136
8.2.3 Assoziationen . . . 151
8.2.4 Datentypen . . . 153
8.3 Modellunabhängige Bestandteile . . . 155
8.3.1 Reflexion . . . 155
8.3.2 XMI . . . 158
8.4 Zusammenfassung . . . 159

9 Generische Komponenten 161
9.1 Ansatz . . . 161
9.2 Steuerkomponente . . . 163
9.2.1 Die Reader-Schnittstelle . . . 164
9.2.2 Modul-Schnittstelle . . . 164
9.2.3 Kontextgenerator-Schnittstelle . . . 164
9.2.4 Schablonenausführer-Schnittstelle . . . 165
9.2.5 Formatierer-Schnittstelle . . . 165
9.3 Schablonenablaufumgebung . . . 166
9.3.1 Schablonenmechanismen . . . 166
9.3.2 Kontextgenerator . . . 174

9.3.3 Schablonenausführer 177
9.4 Zusammenfassung 180

10 Anwendungen **183**
10.1 Petri-Netze 183
10.1.1 Definition 184
10.1.2 MOmo-Codegenerator 185
10.1.3 Entwicklung einer Waschmaschinensteuerung 197
10.2 MOF 202
10.2.1 Codegenerator 202
10.2.2 Laufzeitverhalten 204
10.3 Zusammenfassung 205

11 Schlussbemerkungen **207**
11.1 Zusammenfassung 207
11.2 Ausblick 210
11.2.1 Erweiterungen des MOmo-Baukastens 210
11.2.2 Anwendung des MOmo-Baukastens 211

A Aufbau der Konfigurationsdatei **213**

Abbildungsverzeichnis **217**

Tabellenverzeichnis **221**

Literaturverzeichnis **223**

Abkürzungsverzeichnis **232**

Kapitel 1

Einleitung

1.1 Motivation

Die Anforderungen an moderne Softwaresysteme steigen kontinuierlich an. Immer mehr Funktionalität muss in immer kürzeren Zeitzyklen entwickelt werden. Zusätzlich wird Software in immer mehr Bereichen eingesetzt, so dass die Qualität von Softwaresystemen immer wichtiger wird. So ist etwa im Bereich der eingebetteten Systeme eine ständige Zunahme des Softwareanteils am Gesamtsystem zu beobachten, weil immer mehr Funktionalität durch Software auf frei programmierbarer Hardware anstelle spezieller Hardware implementiert wird. Dies gilt für alle Bereiche der eingebetteten Systeme, insbesondere auch für sicherheitskritische Bereiche, so dass zudem völlig neue Anforderungen an Softwaresysteme gestellt werden. Neben der Steigerung des Umfangs moderner Softwaresysteme nehmen daher auch die qualitativen Anforderungen an diese Systeme zu.

Das Problem, immer mehr Software in immer kürzerer Zeit bei steigenden qualitativen Anforderungen produzieren zu müssen, kann nur durch Anwendung neuer Prozesse und Techniken gelöst werden, die eine Erhöhung des Abstraktionsniveaus bei der Softwareentwicklung ermöglichen. Dazu müssen automatisierbare Anteile der Softwareentwicklung identifiziert und geeignete Werkzeuge zur Automatisierung entwickelt werden.

In den Analyse- und Entwurfsphasen heute üblicher Softwareentwicklungsprozesse werden Modelle des zu entwickelnden Softwaresystems erstellt. In der Implementierungsphase werden diese Modelle auf eine bestimmte Implementierungstechnologie abgebildet, um das System zu realisieren. Ein naheliegender und vielversprechender Schritt zur Reduzierung des Entwicklungsaufwandes ist daher, die Abbildung eines Entwurfs auf eine Implementierungstechnologie zu automatisieren.

Die Vision, Softwaresysteme auf hoher Abstraktionsebene zu modellieren und anschließend automatisch in lauffähige Programme zu transformieren, wird bereits längere Zeit verfolgt. Durch die Einführung der sog. *Model Driven Architecture* (MDA) ist allerdings in letzter Zeit ein breites Interesse für diese Idee ausgelöst worden. Die MDA definiert einen groben Rahmen, der die

Transformation eines Modells in eine lauffähige Implementierung in zwei Schritte unterteilt. Im ersten Schritt wird aus einem plattformunabhängigen Modell ein plattformspezifisches Modell erzeugt und im zweiten Schritt das plattformspezifische Modell in die lauffähige Implementierung transformiert.

Die MDA schreibt nicht vor, wie ein plattformunabhängiges Modell in ein plattformspezifisches Modell zu überführen ist. Aufgrund der großen Akzeptanz der *Unified Modeling Language* (UML) ist der meistgenannte Ansatz zur Transformation plattformunabhängiger in plattformspezifische Modelle die Verwendung sog. *UML-Profile*. Ein UML-Profil besteht im Wesentlichen aus einer Menge sog. Stereotypen, die Elemente plattformunabhängiger Modelle auf die Modelle einer Implementierungstechnologie abbilden. Ein weitergehender Ansatz ist die Abbildung des plattformunabhängigen Modells auf ein Modell, das sich ausschließlich aus Repräsentationen der Elemente der gewünschten Implementierungstechnologie zusammensetzt.

Die Realisierung einer „automatischen Softwareentwicklung“ erfordert geeignete Werkzeuge. Daraus resultieren zwei wesentliche Probleme, die die modellgetriebene Entwicklung von Softwaresystemen in vielen Anwendungsbereichen bislang behindern: (zu) hoher Aufwand für die Entwicklung von Werkzeugen für domänenspezifische Modellierungssprachen und mangelnde Anpassbarkeit der verfügbaren Werkzeuge. Ein Lösungsansatz, der zur Behebung dieser Probleme beitragen kann, wird im Folgenden skizziert.

1.2 Problemstellung und Lösungsansatz

Die in Abschnitt 1.1 dargestellten Probleme können nicht durch ein einzelnes Werkzeug oder einen Ansatz gelöst werden. Stattdessen bedarf es einer Werkzeugkette bzw. ineinandergreifender Ansätze, die jeweils ein Teilproblem lösen. In dieser Arbeit wird ein Lösungsansatz zur Erzeugung von Quelltexten einer Programmiersprache aus Modellen vorgestellt, der sich mit Ansätzen zur Modelltransformation und grafischen Diagrammeditoren kombinieren lässt, um die modellbasierte Entwicklung von Softwaresystemen zu ermöglichen. In Abb. 1.1 ist die Aneinanderreihung der drei Arbeitsschritte bzw. Werkzeuge dargestellt, die zur Automatisierung der Implementierungsphase benötigt werden.

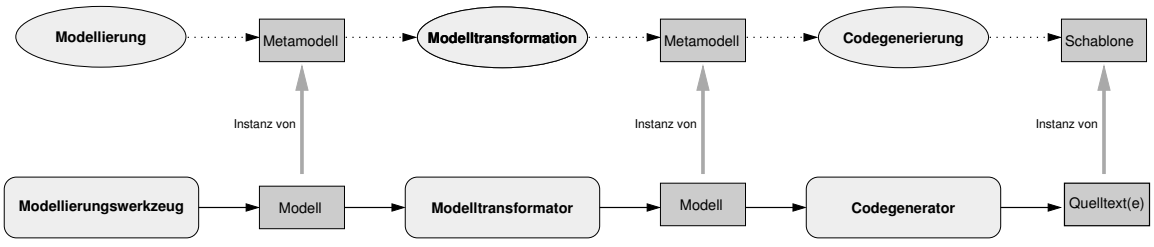


Abbildung 1.1: Automatisierung der Implementierungsphase

Die obere Reihe zeigt die Tätigkeiten und die Artefakte, die zu ihrer Ausführung benötigt werden. Die untere Reihe ordnet jeder Tätigkeit das entsprechende Werkzeug zu. Jede Tätigkeit

ist durch eine Ellipse und jedes Artefakt durch ein Rechteck dargestellt. Beziehungen zwischen Tätigkeiten und Artefakten sind durch gestrichelte Pfeile dargestellt, die von der Tätigkeit zum verwendeten Artefakt gerichtet sind. Während des ersten Schrittes, der *Modellierung* eines Softwaresystems, wird eine Modellierungssprache benutzt, um das System zu beschreiben. Die Syntax von Modellierungssprachen wie der UML werden durch ein sog. *Metamodell* beschrieben. Die Modelle, die mit einer so definierten Sprache erstellt werden, bestehen aus Instanzen der Elemente des Metamodells. Solche Instanziierungen sind in Abb. 1.1 durch graue Pfeile dargestellt. Auch Zeichenketten, die manuell erstellten Code enthalten, können als Instanzen eines Metamodells betrachtet werden und lassen sich so in den automatisch erzeugten Code integrieren. Auf diese Weise können die Synchronisationprobleme verringert werden, die bei der manuellen Integration von manuell erstelltem und generiertem Code in der Regel auftreten. Der zweite Schritt ist die *Modelltransformation*, die eine Abbildung zwischen zwei Metamodellen ausdrückt. Ein Metamodell beschreibt die Elemente der Modellierungssprache, und ein weiteres Metamodell repräsentiert die Elemente der Implementierungstechnologie, die zur Realisierung des Systems verwendet werden soll. Der dritte Schritt, die *Codegenerierung*, führt eine Abbildung der Elemente des Metamodells der Implementierungstechnologie auf Quelltext derselben Implementierungstechnologie durch. Die automatische Durchführung der letzten zwei Schritte ermöglicht die gewünschte Anhebung des Abstraktionsniveaus bei der Softwareentwicklung.

In letzter Zeit ist eine Vielzahl flexibler Ansätze zur Durchführung von Modelltransformationen entwickelt worden. Die meisten dieser Ansätze führen allerdings nur Transformationen zwischen Modellen durch, um z. B. den Datenaustausch zwischen Werkzeugen zu ermöglichen, und sind nicht zur Transformation eines Modells in eine lauffähige Implementierung vorgesehen. Außerdem sind viele Ansätze aufgrund des hohen Implementierungsaufwandes für die Werkzeuge selbst nur prototypisch oder überhaupt nicht realisiert.

Um die Implementierungsphase automatisieren zu können, werden neben Werkzeugen zur Durchführung von Abbildungen allgemeiner Modelle auf plattformspezifische Modelle auch Generatoren benötigt, die plattformspezifische Modelle in Code der Implementierungstechnologie übertragen. Die Realisierung solcher Codegeneratoren ist sehr aufwändig, weil die Abstraktionsebene im Vergleich zu Modelltransformationen niedriger ist. Das Ziel dieser Arbeit ist, den Aufwand für die Implementierung von Codegeneratoren, die den letzten Schritt der in Abb. 1.1 dargestellten Werkzeugkette durchführen, soweit wie möglich zu reduzieren. Um dieses Ziel zu erreichen, wurde der *MOmo*[1]*-Baukasten* entwickelt, dessen Bestandteile zum Aufbau einer Klasse von Codegeneratoren verwendet werden können. Im Folgenden werden solche Codegeneratoren als *MOmo-Codegeneratoren* bezeichnet. Abb. 1.2 zeigt ein Beispiel für den Ablauf der automatischen Codegenerierung mit Hilfe eines MOmo-Codegenerators.

MOmo-Codegeneratoren verarbeiten Modelle, die durch XMI-Dokumente repräsentiert werden. XMI definiert eine Abbildung zwischen der Modellierungssprache MOF und der Dokumentenbeschreibungssprache XML, so dass sich jedes Modell, das durch die Elemente einer MOF-basierten Modellierungssprache beschrieben wird, durch ein XMI-Dokument darstellen lässt. In Abb. 1.2 wird die Eingabe durch einen Editor für Petri-Netze [Pet62] symbolisiert, der das Modell eines Netzes als XMI-Dokument abspeichern kann. Der Editor steht stellvertretend

[1] MOF-basierte Metamodellierung

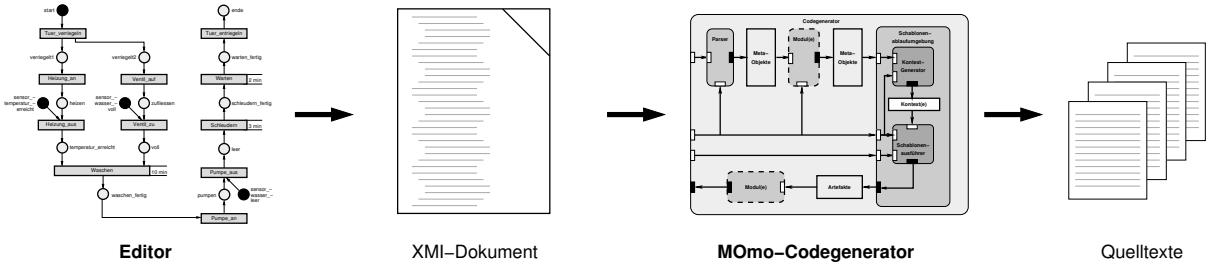


Abbildung 1.2: Automatische Codegenerierung mit einem MOmo-Codegenerator

für alle Editoren, die das Modellieren mit einer MOF-basierten Modellierungssprache ermöglichen, inbesondere also für alle UML-Modellierungswerkzeuge. Das XMI-Dokument dient als Eingabe für den MOmo-Codegenerator, der eine Menge von Quelltexten erzeugt, die in eine lauffähige Implementierung des Netzes übersetzt werden können.

Um eine derartige Transformation durchführen zu können, muss ein MOmo-Codegenerator eine Reihe von Komponenten enthalten, deren Entwicklung von den Bestandteilen des MOmo-Baukastens unterstützt werden muss. Abb.1.3 zeigt ein Schema des Aufbaus der Codegeneratoren, die mit Hilfe des MOmo-Baukastens erstellt werden können.

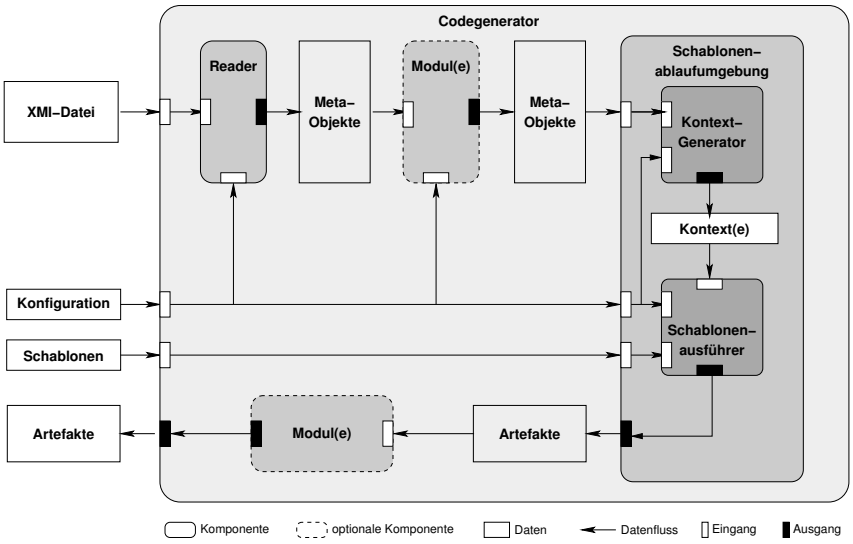


Abbildung 1.3: Schema eines MOmo-Codegenerators

MOmo-Codegeneratoren bestehen mindestens aus den Komponenten einer generierten Metamodellimplementierung, drei generischen Komponenten und einer beliebigen Anzahl von Schablonen. Die Metamodellimplementierung, eine Java™-Implementierung der Elemente eines MOF-Modells, legt den Sprachumfang der Modellierungssprache fest, die der Codegenerator „versteht“. Das Metamodell wird durch die *Meta Object Facility* (MOF) beschrieben. Die Implementierung kann durch einen im Baukasten enthaltenen MOF-Codegenerator automatisch erzeugt werden und umfasst auch einen *Parser*, der XMI-Dokumente einlesen und in Instanzen

der Elemente der Metamodellimplementierung umwandeln kann. Der MOF-Codegenerator ist ebenfalls nach dem in Abb. 1.3 dargestellten Schema aufgebaut.

Auf den Instanzen der Metamodellimplementierung lassen sich durch benutzerdefinierte *Module* beliebige Manipulationen durchführen, bevor die Metaobjekte an die Schablonenablaufumgebung weitergeleitet werden. Die Ablaufumgebung für die Schablonen wird durch generische Komponenten bereitgestellt. Da diese Komponenten, *Kontextgenerator* und *Schablonenausführer*, ausschließlich Mechanismen der MOF verwenden, sind sie unabhängig von der Modellierungssprache, die der Codegenerator in Quelltexte übersetzen soll, und können für alle MOmo-Codegeneratoren unverändert verwendet werden. Durch den MOmo-Baukasten lässt sich daher ohne großen Aufwand eine Ablaufumgebung für Schablonen erstellen, die Elemente oder Gruppen von Elementen einer MOF-basierten Modellierungssprache in Quelltexte einer Programmiersprache übersetzen. Zusätzlich werden lediglich Vorschriften benötigt, die die Artefakte beschreiben, die für einzelne Modellelemente oder Modellelementgruppen erzeugt werden sollen. Diese sind die einzigen Bestandteile eines MOmo-Codegenerators, die manuell erstellt werden müssen. Die resultierenden Artefakte können durch eine zweite Art benutzerdefinierter Module weiterverarbeitet werden. Die wichtigste Anwendung dieser Module ist die Formatierung der erzeugten Quelltexte.

1.3 Organisation der Arbeit

Kapitel 2 enthält eine Einführung in die modellbasierte Softwareentwicklung. Zunächst werden die Bedeutungen der Begriffe *Modell* und *Metamodell* für die weitere Arbeit festgelegt. Anschließend erfolgt eine Beschreibung des Vorgehens während der modellbasierten Entwicklung eines Softwaresystems. Stellvertretend für die heute gängigen Prozesse, die zur Entwicklung ausgeführt werden, wird der *Rational Unified Process* beschrieben. Abschließend enthält das Kapitel eine Beschreibung der *Model-Driven Architecture*, die eine weitergehende Automatisierung der Softwareentwicklung vorsieht und eine schrittweise Überführung eines Modells in eine Implementierungstechnologie durch Modelltransformatoren und Codegeneratoren angibt.

Kapitel 3 gibt einen Überblick über die aktuellen Entwicklungen in den Bereichen Modelltransformation und Codegenerierung, um die Entwicklung des MOmo-Baukastens zu motivieren. Aufbauend auf den Ergebnissen aus Kapitel 3 beschreibt Kapitel 4 die Anforderungen an die Codegeneratoren, die mit Hilfe des MOmo-Baukastens realisiert werden sollen, und erläutert die Arbeitsweise der verschiedenen Elemente des MOmo-Baukastens. Nach der groben Beschreibung der Funktion der einzelnen Bestandteile des Baukastens folgen in den weiteren Kapiteln detaillierte Beschreibungen der zugrunde liegenden Standards und der Arbeitsweise der wichtigsten Bestandteile.

In den Kapiteln 5-7 werden die Grundlagen des MOmo-Baukastens beschrieben. Kapitel 5 beschreibt die Grundbausteine der Meta-Modellierungssprache MOF, die zur Definition von Modellen der abstrakten Syntax von Modellierungssprachen, insbesondere der *Unified Modeling Language* (UML), verwendet wird. Erweiterungen und Anpassungen der UML können durch

Erweiterungen bzw. Veränderungen des UML-Metamodells beschrieben werden, so dass die MOF eine wesentliche Grundlage der UML-basierten Modellierung ist.

Die Kapitel 6 und 7 beschreiben die Empfehlung *Extensible Markup Language* (XML) des *World Wide Web Commitee* (W3C) und den Standard *XML Metadata Interchange* (XMI) der *Object Management Group* (OMG). XML ist ein allgemein verwendbares Dokumentformat zur Beschreibung strukturierter Daten. Die XMI-Spezifikation beschreibt eine Abbildung der Elemente der MOF auf XML-Dokumente. Jegliche Daten, deren Struktur durch ein MOF-Modell beschrieben wird, können durch XMI-Dokumente repräsentiert werden. Insbesondere können daher UML-Modelle durch XMI-Dokumente dargestellt und zwischen verschiedenen Werkzeugen ausgetauscht werden.

Der MOmo-Baukasten basiert auf einer Implementierung der Meta-Modellierungssprache MOF. Die Implementierung wird in der Programmiersprache Java™durchgeführt, so dass eine Abbildung der Elemente der MOF auf Java-Konstrukte erforderlich ist. Kapitel 8 beschreibt die Abbildungen der grundlegenden Bausteine der MOF auf Java-Elemente. Die gewählte Abbildung läßt sich problemlos auf andere objektorientierte Programmiersprachen übertragen und kann daher auch für andere Implementierungen des MOmo-Konzeptes verwendet werden.

Kapitel 9 beschreibt die generischen Komponenten des MOmo-Baukastens, die auf der Basis der Java-Abbildung der MOF entwickelt wurden, um eine Grundlage für die Implementierung von Codegeneratoren für MOF-basierte Modellierungssprachen bereitzustellen. Die *Steuerkomponente* regelt den Ablauf eines Codegenerierungsprozesses und wird für jeden MOmo-Codegenerator unverändert verwendet. Die Komponenten der *Schablonenablaufumgebung* sind generisch und verarbeiten jede Modellierungssprache, die durch ein MOF-Modell definiert ist. Zusätzlich werden lediglich Vorschriften benötigt, die die Artefakte beschreiben, die für einzelne Modellelemente oder Modellelementgruppen erzeugt werden sollen. Diese sind die einzigen Bestandteile eines MOmo-Codegenerators, die manuell erstellt werden müssen.

Kapitel 10 beschreibt die Erfahrungen, die bei der Anwendung des MOmo-Baukastens gewonnen wurden. Zunächst wird eine Beispielimplementierung eines einfachen Codegenerators für Petri-Netze [Pet62] mit Hilfe des Baukastens vorgestellt. Daraufolgend wird die Anwendung des MOmo-Baukastens zur Entwicklung von Codegeneratoren für die OMG-Modellierungssprachen MOF und UML beschrieben. Abschließend werden die Laufzeiten angegeben, die der MOmo-MOF-Codegenerator für die Erzeugung ausgewählter Metamodellimplementierungen benötigt. Abschließend werden die Ergebnisse der Arbeit in Kapitel 11 zusammengefasst. Außerdem wird ein Ausblick auf weitergehende Anwendungsgebiete und Arbeiten gegeben.

Kapitel 2

Modellbasierte Softwareentwicklung

Der Funktionsumfang, der in modernen technischen Geräten durch Software realisiert wird, nimmt ständig zu. In vielen Anwendungsgebieten werden softwarebasierte Systeme in zunehmendem Maße auch in sicherheitskritischen Teilsystemen eingesetzt. Durch diese Entwicklung steigen Komplexität und Anforderungen an die Qualität von Software gleichzeitig an. Eine Möglichkeit, den erhöhten Anforderungen gerecht zu werden, ist der Einsatz modellbasierter Softwareentwicklungsmethoden und -werkzeuge.

Die modellbasierte Softwareentwicklung verwendet Modelle des zu entwickelnden Softwaresystems, um in jeder Phase der Softwareentwicklung von unwichtigen Details abstrahieren zu können. Dies ermöglicht die Reduzierung der Komplexität und stellt eine wichtige Grundlage dar, um die oben genannten Anforderungen zu erfüllen. Die Modelle, die in verschiedenen Entwicklungsphasen benutzt werden, stehen zueinander in Beziehung, verbessern die Kommunikation über das System, erhöhen das Verständnis zwischen Entwickler und Benutzer/Auftraggeber und steigern die Wiederverwertbarkeit von Systemkomponenten. Der Code zu entwickelnder Systeme kann zudem (teil-) automatisch aus den Modellen erzeugt werden, so dass die Entwicklungszeit verkürzt wird. Zudem kann lauffähiger Code bereits aus den unvollständigen Modellen früher Entwicklungsphasen generiert werden. Daraus lassen sich mit vergleichsweise geringem Aufwand Prototypen erstellen, die die Kommunikation zwischen Anwender und Entwickler weiter vereinfachen.

In diesem Kapitel wird in die modellbasierte Software-Entwicklung eingeführt. Der Fokus liegt auf dem Einsatz objektorientierter Modellierungssprachen und -methoden, insbesondere der *Unified Modeling Language* (UML) [UP03b]. Die Abschnitte 2.1 und 2.2 legen die Bedeutung der Begriffe *Modell* und *Metamodell* für die weitere Arbeit fest. Auf Grundlage der Begriffsdefinitionen beschreibt Abschnitt 2.3 das Vorgehen bei der modellbasierten Software-Entwicklung. Den Schwerpunkt der Beschreibung bilden zwei ausgewählte Prozesse, die stellvertretend für den Stand der Technik und die zukünftige Entwicklung näher betrachtet werden.

2.1 Modelle

Im Mittelpunkt des modellbasierten Vorgehens zur Entwicklung eines Software-Systems stehen die Modelle, die in den verschiedenen Phasen der Entwicklung erstellt werden. Der Modellbegriff ist aber nicht auf die Software-Entwicklung beschränkt, sondern es werden in nahezu allen Bereichen der Wissenschaft und Technik Modelle benutzt, um die unwichtigen Eigenschaften eines Untersuchungsgegenstandes auszublenden und/oder die wichtigen Eigenschaften hervorzuheben.

Eine allgemeine Definition des Begriffs *Modell* lautet nach [Lor04]: „Als Modell eines Objektes oder Systems (des Originals) wird ein ähnliches Objekt oder System verstanden, mit dessen Hilfe man Aufgaben lösen kann, deren Lösung am Original nicht möglich oder nicht zweckmäßig ist. Ähnlich müssen die Eigenschaften sein, die bezüglich der Aufgabe wesenlich sind."

Aus dieser ersten allgemeinen Definition wird zum einen deutlich, dass die Auswahl der relevanten Eigenschaften für die sinnvolle Verwendung von Modellen eine wichtige Rolle spielt. Zum anderen zeigt die Definition, dass es viele verschiedene Arten von Modellen geben kann, und die Modelle mit ganz unterschiedlichen Zielen erstellt und eingesetzt werden. Aufgrund der unterschiedlichen Charakteristika der Anwendungsgebiete und Zielstellungen unterscheiden sich die Eigenschaften der Modelle, die zum Erreichen dieser Ziele verwendet werden, deutlich. Im Folgenden soll zunächst ein Überblick über die typischen Anwendungsgebiete von Modellen gegeben werden. Darauf folgt eine Betrachtung der Eigenschaften, die zur Klassifizierung von Modellen geeignet sind. Aufbauend auf diesen Informationen wird der Modellbegriff für die weitere Arbeit festgelegt.

2.1.1 Klassifikation nach Anwendungsbereichen

Modelle werden mit verschiedenen Zielen in verschiedenen Anwendungsbereichen angewendet. Im Folgenden sollen zunächst einige Definitionen des Modellbegriffs aus allgemeinen Nachschlagewerken vorgestellt werden, um einen Überblick über die Bedeutungen des Modellbegriffs in verschiedenen Anwendungsgebieten zu gewinnen. Die vorgestellten Definitionen stammen aus der Brockhaus-Enzyklopädie [Bro98] und Meyers Enzyklopädischem Lexikon [Mey76] und umfassen neben einer allgemeinen Definition die sieben Anwendungsgebiete Logik und Mathematik, Malerei und Bildhauerei, Mode, Naturwissenschaften, Technik, Wirtschaftswissenschaften sowie Soziologie, Psychologie und Politikwissenschaften:

1. *allgemein:* Muster, Entwurf, z. B. in der Baukunst ein Architekturmodell; Vorbild, Beispiel

2. *Logik und Mathematik:* ein Bereich (meist eine Menge), dessen Mitglieder und deren Verknüpfungen eine durch Axiome beschriebene abstrakte Struktur besitzen.

   Die Modell-Theorie versucht mithilfe von Modellen die Widerspruchsfreiheit von Axiomensystemen sowie die Unabhängigkeit der Axiome voneinander zu beweisen.

3. *Malerei und Bildhauerei:* Naturgegenstand, besonders der Mensch (aber auch Tier oder Pflanze), der als Vorbild künstlerischer Gestaltung dient.

   In der Bildhauerei auch ein stereometrisch genaues Vorbild des endgültigen Werkes.

4. *Mode:* als Einzelstück gefertigtes Kleidungsstück; kann abgewandelt als Vorlage für die serienweise Herstellung (Konfektion) dienen

5. *Naturwissenschaften:* ein Abbild der Natur unter Hervorhebung für wesentlich erachteter Eigenschaften und unter Außerachtlassen als nebensächlich angesehener Aspekte.

   Modelle entstehen aus der Wechselwirkung zwischen Hypothesenbildung und Beobachtung oder (in den exakten Naturwissenschaften) messendem Experiment. Modelle sind grundsätzlich nicht endgültig.

6. *Technik:* in verkleinertem, natürlichem oder vergrößertem Maßstab ausgeführte räumliche Abbilder eines technischen Entwurfs oder Erzeugnisses zur anschaulichen Darstellung, zu Lehrzwecken, als Spielzeug oder als wissenschaftliche Versuchsobjekte in Modell-Versuchen.

7. *Wirtschaftswissenschaften:* konstruiertes, vereinfachtes Abbild des tatsächlichen Wirtschaftsablaufes, z. T. in mathematischer Formulierung

8. *Soziologie, Psychologie und Politikwissenschaften:* die heuristischen Möglichkeiten der Modelle werden genutzt, um idealisierte, auf wesentliche Strukturbeziehungen reduzierte Verhaltens-, Interaktions-, Kommunikations-, Lern- oder Konflikt-Modelle als Gedankenexperimente zu konstruieren, an denen — entweder zur Weiterentwicklung von Theorien oder zu wissenschaftlichen (d. h. auf Gesetzen beruhenden) Prognosen — die Kausalfaktoren, die einen politischen oder sozialen Sachverhalt hauptsächlich bestimmen, studiert werden.

In jedem der genannten Bereiche bietet ein Modell eine abstrakte Sicht auf einen Gegenstand oder Ablauf in der Realität. Durch die Abstraktion ermöglichen Modelle, die für wichtig erachteten Bestandteile eines Gegenstandes oder Ablaufs hervorzuheben. Ein Modell ist also keineswegs immer ein Abbild eines tatsächlichen Gegenstandes, obwohl der Begriff im allgemeinen Sprachgebrauch meist so verwendet wird. Stattdessen fordern manche Definitionen eine *Ähnlichkeit* zwischen Modell und tatsächlichem Gegenstand. Dies verlagert das Problem der Begriffsdefinition von der Definition des Modellbegriffs auf die Definition des Ähnlichkeitsbegriffs.

Weiterhin ist unklar, in welcher zeitlichen Reihenfolge Modell und Original zueinander stehen. In manchen Anwendungsfällen existiert das Original vor dem Modell, vielfach z. B. in den Naturwissenschaften. In anderen Fällen ist die zeitliche Reihenfolge aber genau andersherum. Architekturmodelle werden beispielsweise in der Regel vor den realen Bauwerken erstellt. Ähnliches gilt für Prototypen, die im Ingenieurwesen gebaut werden.

Zusammenfassend lässt sich sagen, dass keines der angegebenen Anwendungsgebiete genau der Informatik entspricht, sondern diese eine Mischform der Anwendungsgebiete Logik und

Mathematik, Naturwissenschaften sowie Technik ist. Um eine Definition für die Bedeutung des Modellbegriffs innerhalb dieser Arbeit angeben zu können, müssen daher die Eigenschaften der Modelle aus unterschiedlichen Anwendungsgebieten betrachtet werden.

2.1.2 Klassifikation nach Eigenschaften

Modelle besitzen eine Vielzahl von Eigenschaften, die zur Klassifizierung genutzt werden können. Die Eigenschaften sind weitgehend unabhängig vom Anwendungsbereich des Modells. Sie kennzeichnen daher auch die Modelle, die keinem der oben angegebenen Anwendungsbereiche eindeutig zugeordnet werden können, also insbesondere auch die Modelle in der Softwaretechnik. In Tab. 2.1 sind Eigenschaften von Modellen und ihre Bedeutung angegeben. Eine Klasse von Modellen kann mehrere der aufgeführten Eigenschaften besitzen. Modelle einer Software können z. B. sowohl dynamisch als auch statisch sein, bzw. dynamische und statische Anteile enthalten.

Modelle verschiedener Anwendungsgebiete besitzen eine Teilmenge der angebenen Eigenschaften. Die Eigenschaften können verwendet werden, um die Güte eines Modells bezogen auf einen bestimmten Anwendungsfall zu beurteilen. Je besser die maßgeblichen Eigenschaften des Modells den Eigenschaften des abgebildeten Systems oder Prozesses entsprechen, desto höher ist die Qualität des Modells. Problematisch ist allerdings, dass eine objektive Beurteilung eine klare Definition des Ähnlichkeitsbegriffs erfordert, die für viele der angegeben Kriterien nicht vorliegt.

Wie oben festgestellt, müssen Modelle in der Informatik eine Vereinigungsmenge der Eigenschaften aus Logik und Mathematik, Naturwissenschaften und Technik enthalten, da die Informatik Anteile dieser drei Anwendungsgebiete von Modellen umfasst. Aufgrund dieser Mischcharakteristik wird der Modellbegriff in der Informatik in unterschiedlicher Art und Weise verwendet. Entsprechend schwierig ist es, den Modellbegriff für die Informatik allgemein festzulegen. Thomas [Tho01] zeigt, dass nahezu alle Modelltypen der *Allgemeinen Modelltheorie* von Stachowiak [Sta73] in der Informatik von Bedeutung sind oder zumindest waren.

Um den Modellbegriff trotzdem festlegen zu können, muss das betrachtete Anwendungsgebiet weiter eingeschränkt werden. Der in dieser Arbeit vorgestellte Ansatz zur Implementierung von Codegeneratoren ist dem Gebiet der Softwaretechnik zuzuordnen, so dass die Definition des Modellbegriffs nur die relevanten Eigenschaften von Modellen dieses Gebietes umfassen muss. Selbst für diesen eingeschränkten Bereich ist die Festlegung des Modellbegriffs aber offensichtlich nicht trivial. Beispielsweise enthält das *Lehrbuch der Objektmodellierung* [Bal99] keine Definition des Modellbegriffs.

2.1.3 Begriffsdefinition

In einer terminologischen Untersuchung zur Softwaretechnik definieren Hesse u. a. [HBvB+94] ein Modell als *„idealisierte, vereinfachte, in gewisser Hinsicht ähnliche Darstellung eines Gegenstands, Systems oder Weltausschnitts mit dem Ziel, daran bestimmte Eigenschaften des Vor-*

Attribut	**Beschreibung**
geometrisch	wenn es geometrische Ähnlichkeit zum Original besitzt
physikalisch	wenn es physikalische Effekte nutzt, die auch im Original auftreten
biologisch	wenn es mit dem Original biologisch verwandt oder ähnlich ist
stofflich	wenn in ihm Stoffe benutzt werden, die auch im Original vorkommen
strukturell	wenn es eine strukturelle Ähnlichkeit zum Original gibt, d. h. gleichbenannte Komponenten und gleiche Relationen zwischen ihnen besitzt
funktionell	wenn seine Funktion, d. h. sein Input-Output-Verhalten, dem Original ähnlich ist
stochastisch	wenn in ihm zufällige Einflüsse und Komponenten auftreten, wenn Zufallsgeneratoren genutzt werden, um die im Original vorkommenden Zufallseinflüsse nachzubilden
deterministisch	wenn es keine zufallsabhängigen Einflüsse gibt
statisch	wenn in ihm keine zeitabhängigen Änderungen auftreten
dynamisch	wenn es zeitabhängigen Änderungen unterliegt
kontinuierlich oder stetig	wenn alle in ihm auftretenden Größen stetige Funktionen der Zeit sind und keine sprunghaften Wert- oder Zustandsänderungen auftreten
diskret	wenn darin sprunghafte Wert- oder Zustandsänderungen auftreten
kombiniert/hybrid	wenn es darin sprunghafte Wert- oder Zustandsänderungen und darüber hinaus nichtlineare, durch Differentialgleichungen abbildbare zeitabhängige Prozesse gibt
physisch	wenn es in stofflicher, körperlicher Form existiert
abstrakt oder mathematisch	wenn es nicht physisch ist, sondern als abstraktes Abbild des Originals zur Lösung von Aufgaben geeignet ist (als Aufgabe kommen Identifikationen, Deduktionen und Berechnungen in Frage)
Computermodell	wenn der Computer mit einem geeigneten Programm als Modell des Originalobjektes dient

Tabelle 2.1: Klassifizierung von Modellen nach Lorenz [Lor04]

bilds besser deuten zu können". Diese Begriffsdefinition unterscheidet sich kaum von der oben angegebenen allgemeinen Definition des Modellbegriffs und kann daher auch nicht wesentlich zur Klärung der Bedeutung im Kontext der Softwaretechnik beitragen.

Nach [Sch99] sollte der Modellbegriff aufgrund seiner unklaren Semantik generell überhaupt nicht mehr benutzt werden. Wenn er dennoch benutzt wird, „müsse sich seine Bedeutung aus dem Kontext ergeben". Aufgrund der Vielzahl der möglichen Eigenschaften von Modellen erscheint dies eine sinnvolle Einschränkung. Zudem macht Schefe Vorschläge für alternative Begriffe, die die gängigen Bedeutungen des Modellbegriffs abdecken. Im Kontext dieser Arbeit sind Schefes Definitionen der Begriffe *Spezifikation* und *verkleinerte Systemversion* relevant, da die hier betrachteten Modelle Struktur und Verhalten von Softwaresystemen beschreiben:

Definition 1 *Eine **Spezifikation** ist eine Übersetzung allgemein-sprachlicher Beschreibungen des Systems und enthält Formalisierungen einschlägiger Begriffe zur Validierung. Als Systemvorgabe bildet eine Spezifikation die Theorie zur Verifikation realisierender Strukturen.*

Definition 2 *Eine **verkleinerte Systemversion** oder dessen formalsprachliche Beschreibung bildet einen **Prototyp**, der zur Durchführung von Experimenten geeignet ist, die Zweckmäßigkeit und Benutzbarkeit in einem aktuellen System überprüfen.*

Auf der Basis dieser Definitionen kann der Modellbegriff für die weitere Arbeit festgelegt werden, da die hier betrachteten Modelle entweder eine Spezifikation oder ein Prototyp sind. Durch Herausziehen der Gemeinsamkeit der Definitionen 1 und 2 lässt sich eine Definition finden, die zwar immer noch recht allgemein, für die weitere Arbeit aber ausreichend genau ist:

Definition 3 *Ein **Modell** ist eine formale Beschreibung eines (Software-) Systems.*

Neben dem Modellbegriff wird in dieser Arbeit auch der Begriff *Metamodell* häufig verwendet. Die betrachteten Modellierungssprachen und -konzepte basieren auf dieser besonderen Klasse von Modellen. Im folgenden Abschnitt wird daher der Metamodellbegriff gesondert definiert.

2.2 Metamodelle

Die Menge der Modelle, die sich mit Hilfe einer Modellierungssprache beschreiben lassen, wird durch das Vokabular der Sprache und die Regeln, die die zulässige Verwendung des Vokabulars festlegen, beschränkt. Es gibt verschiedene Möglichkeiten, das Vokabular und die Regeln festzulegen. Die meisten (formalen) Sprachen werden durch eine Menge von Wörtern und eine Grammatik, die festlegt, wie die Wörter verwendet werden dürfen, definiert. Eine alternative Möglichkeit zur Beschreibung einer Sprache ist, die Sprache durch ein Modell zu beschreiben. Entsprechende Modelle, die Elemente und Struktur einer Klasse von Modellen definieren, werden als *Metamodelle* bezeichnet. Metamodelle sind also Modelle gemäß Definition 3, die im Wesentlichen die Eigenschaften aus Definition 1 besitzen.

Das Wort *Meta* stammt aus dem Griechischen und bedeutet „über, nach, neben, zwischen" [Wis01]. In den meisten Anwendungsgebieten werden Meta-Ebenen verwendet, um Aussagen

über die Struktur der Informationen untergeordneter Ebenen zu treffen. Ein Beispiel dafür ist die Kommunikationstheorie, die *Meta-Kommunikation* als „Kommunikation über Art und Inhalt des Gesagten“ definiert [Sch81]. Im Allgemeinen wird auf einer Meta-Ebene die Semantik der darunterliegenden Ebene beschrieben, wobei zur Beschreibung wiederum die Konstrukte der darunterliegenden Ebene verwendet werden. Ein Beispiel dafür sind Bücher, die Syntax und Semantik der deutschen Sprache erklären und in deutscher Sprache verfasst sind.

Metamodelle werden, wie alle Modelle, immer mit einer bestimmten Absicht erstellt. Es ist daher schwierig, ein Metamodell zu verstehen, wenn die vom Modellierer verfolgte Absicht unklar ist. Typische Anwendungsgebiete von Metamodellen im Bereich der Informatik sind:

- Bereitstellung eines Schemas für Daten, die ausgetauscht werden sollen. Beispiele für diesen Anwendungsfall sind der Austausch von Modelldaten zwischen Software-Entwicklungswerkzeugen und der Austausch von Produktdaten. Neben dem Schema wird das Metamodell häufig auch als Grundlage der Definition eines „Speichers“ für Daten verwendet.

- Definition einer Sprache, die eine bestimmte Methode oder einen bestimmten Prozess unterstützt. Prominenteste Beispiele solcher Sprachen sind die OMG-Modellierungssprachen *Common Warehouse Metamodel* [Obj03a], *Meta Object Facility* [ACC+03] und UML.

Metamodelle werden in beiden Anwendungsfällen zur Definition der abstrakten Syntax von Daten verwendet. Die abstrakte Syntax definiert den strukturellen Aufbau der Elemente, die zum Aufbau eines Modells verwendet werden dürfen. Neben der abstrakten Syntax benötigt man allerdings die konkrete Syntax und die Semantik der verwendeten Modellierungselemente, um ein Modell erstellen bzw. verstehen zu können. In metamodellbasierten Sprachdefinitionen werden konkrete Syntax und Semantik einer Sprache meist durch eine Kombination von Konsistenzbedingungen, umgangssprachlichen Angaben und Beispielen festgelegt. Metamodelle lassen sich aber durchaus zur Definition von konkreter Syntax und Semantik einsetzen. Dazu werden Modelle für abstrakte Syntax, konkrete Syntax und Semantik einer Sprache definiert und zusätzlich die Beziehungen zwischen den Elementen dieser Modelle angegeben. Clark u. a. [CEK+00] sowie Reggio und Astesiano [RA01] beschreiben Ansätze, die die Semantik der UML durch Metamodelle und Abbildungen zwischen Metamodellen spezifizieren.

In einem einführenden Artikel zur Metamodellierung [Met02] wird ein Zeichenprogamm verwendet, um die Verwendung eines Metamodells zur Festlegung der Bedeutung einer Klasse von Modellen zu verdeutlichen. Wenn der Anwender eines Zeichenprogramms ein Rechteck zeichnen möchte, markiert er zwei Ecken des Rechtecks. Das Zeichenprogramm errechnet aus den festgelegten Koordinaten die Abmaße des Rechtecks und stellt dieses dar. Dieser Vorgang funktioniert nur dann, wenn das Zeichenprogramm im Vorhinein „weiß“, dass die nachfolgenden Aktionen zum Erstellen eines Rechtecks ausgeführt werden. Andernfalls könnte nach dem Markieren des zweiten Punktes genauso gut ein Kreis oder eine Raute dargestellt werden.

In der Regel werden Zeichenprogramme verwendet, um Informationen übersichtlich darzustellen. Der Anwender weist jedem grafischen Symbol eine besondere Bedeutung zu. Beispiels-

weise werden Organisationseinheiten häufig durch Rechtecke ausgedrückt. In diesem Fall unterscheidet sich die Semantik des Zeichenprogrammes deutlich von der Semantik, die der Anwender im Sinn hat. Die unterschiedliche Deutung desselben Modells durch Werkzeug und Anwender liegt hier an der zweckentfremdeten Verwendung des Werkzeugs durch den Anwender. Daraus wird deutlich, dass die Semantik eines Modells wesentlich von der Anwendungsdomäne abhängt. Darauf aufbauend wird die Semantik eines Metamodells als *„eine Sammlung von Konzepten, die ein Vokabular definieren, um Aussagen über eine Anwendungsdomäne zu machen“* [HBvB+94] definiert.

Diese Definition des Metamodellbegriffs erscheint naheliegend und ist leicht verständlich. Metamodellarchitekturen, beispielsweise das Metadatenmodell der *Object Management Group* (OMG), erscheinen dagegen in der Regel komplex und und schwer verständlich. Ein Grund dafür ist, dass Metamodelle in der Informatik nach demselben Prinzip aufgebaut sind, das im oben beschriebenen Beispiel aus den Sprachwissenschaften dargestellt wurde. Häufig werden Metamodelle aus exakt den gleichen Elementen aufgebaut wie die Modelle, deren Struktur sie definieren. Daraus resultieren sowohl die Vor- als auch die Nachteile der Metamodellierung. Einerseits ist das Metamodell aus Elementen aufgebaut, deren Syntax und Semantik dem Anwender bekannt ist, so dass das Metamodell leichter zu verstehen ist. Dies wiederum verbessert das Verständnis der Syntax und Semantik, die das Metamodell festlegt, so dass ein Kreislauf entsteht, der in jedem Durchlauf das Modellverstehen auf beiden Modellierungsebenen verbessert. Andererseits wirkt eben dieser Kreislauf häufig verwirrend, weil die eindeutige Zuordnung eines bestimmten Modellierungselementes zu einer bestimmten Modellierungsebene schwer fällt.

Die praktische Relevanz von Metamodellen wurde unter anderem aus diesem Grund zunächst angezweifelt. Dies hat sich spätestens seit Erscheinen der *Unified Modeling Language* (UML), die durch ein Metamodell spezifiziert ist, geändert. Zwischenzeitlich wurde gar das Gegenteil behauptet (z. B. „Meta-meta is better-better“ [CDI+97]), d. h. Metamodellierung als eine Art Allheilmittel für die oben genannten Anwendungsbereiche Datendefinition und Sprachdefinition betrachtet. Mittlerweile lässt sich feststellen, dass die Metamodellierung zwar eine akzeptierte Vorgehensweise insbesondere zur Definition der Elemente einer Modellierungssprache ist, aber durchaus nicht in allen Fällen die beste Lösung darstellt [Jec00].

Wie beschrieben verursachen die unterschiedlichen Ebenen der Metamodellierung einen Großteil der Verständnisprobleme. Neben den Schwierigkeiten bei der Zuordnung von Modellelementen zu den verschiedenen Modellierungsebenen entstehen weitere Verständnisprobleme, weil sowohl Beziehungen zwischen den Modellen und Sprachen derselben Ebene bestehen als auch zwischen Modellen und Sprachen verschiedener Ebenen. Abb. 2.1 zeigt die Beziehungen zwischen Modellen und Sprachen verschiedener Ebenen der sprachbasierten Metamodellierung und fokussiert auf die Ebenen der Metamodellierung, die allgemein als relevant angesehen werden [PSV98].

Die Modelle der Ebene 0 werden durch Elemente einer Sprache ausgedrückt, die auf derselben Ebene angeordnet ist. Modelle der Ebene 0 bilden einen Ausschnitt aus der „realen Welt“ ab. Im Kontext dieser Arbeit ist dieser Ausschnitt ein Softwaresystem. Die Sprache der Ebene 0 wird durch ein Modell der Ebene 1 definiert. Dieses Modell ist somit auch ein Modell des Modells der Ebene 0 und wird daher als Metamodell bezeichnet. Da die Sprache der Ebene 1 also zur

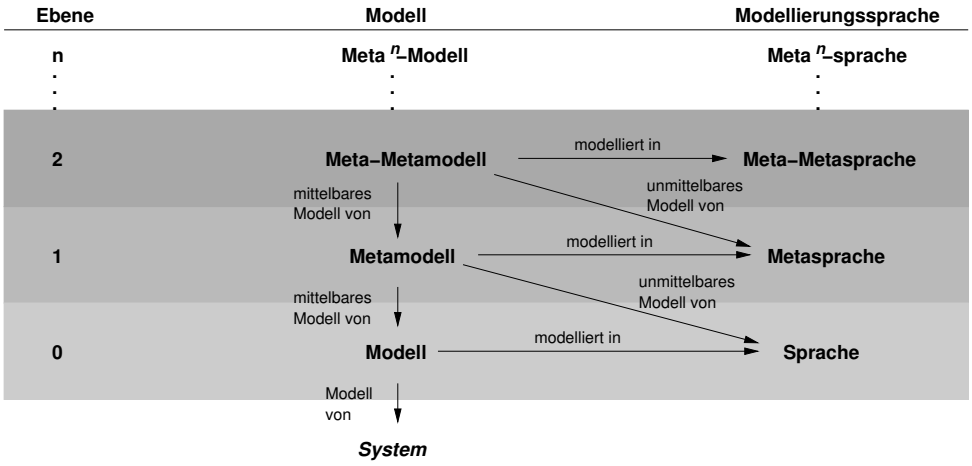


Abbildung 2.1: Sprachbasierter Metamodellbegriff nach Strahringer [Str98]

Definition eines Metamodells verwendet wird, wird sie als Metasprache bezeichnet. Zwischen den Modellen und Sprachen der Ebenen 1 und 2 existieren gleichartige Beziehungen, so dass sich dort Meta-Metamodell und Meta-Metamodellierungssprache befinden. Dies lässt sich auf n-Ebenen erweitern, so dass auf der n-ten Ebene Meta^n-modelle und -Modellierungssprachen anzuordnen sind.

Die Sprache einer Ebene i wird benutzt, um die Modelle derselben Ebene zu definieren. Gleichzeitig ist die Sprache aber selbst durch ein Modell der nächsthöheren Ebene festgelegt. So befindet sich das Modell einer Sprache der Ebene i auf der Ebene $i + 1$ und wird folglich durch eine Sprache der Ebene $i + 1$ beschrieben. Das Modell auf der Ebene $i + 1$ wird deshalb als *unmittelbares Modell* der Sprache auf der Ebene i bezeichnet. Außerdem ist das Modell der Ebene $i + 1$ *mittelbares Modell* aller Modelle auf der Ebene i, da es die Elemente festlegt, aus denen die Modelle der Ebene i bestehen dürfen.

Aus diesen Zusammenhängen resultiert neben der begrifflichen Verwirrung ein weiteres Problem, da sich diese Beziehungen zwischen Modellen und Sprachen endlos fortführen lassen. Um dieses Problem zu lösen, muss an einer Stelle mit einer Sprache oder einem Modell begonnen werden. Dann fehlt aber die Beziehung zu einem Modell der nächsthöheren Ebene bzw. einer Sprache derselben Ebene. Die Lösung dieses Problems ist, eine Sprache durch ein Modell derselben Ebene festzulegen.

Aus der beschriebenen Problematik folgt, dass der Metamodellbegriff nicht auf Basis der Ebenen der Metamodellierung festgelegt werden kann. Stattdessen wird eine etwas allgemeinere Formulierung gewählt, die lediglich die wesentliche Eigenschaft festlegt, die ein Metamodell von einem Modell unterscheidet:

Definition 4 *Ein **Metamodell** ist ein Modell, das die Syntax der Modellelemente einer Klasse von Modellen festlegt.*

Der beschriebene, sprachbasierte Metamodellbegriff steht in der weiteren Arbeit im Mittelpunkt. Ähnlich relevant ist allerdings der prozessbasierte Metamodellbegriff, der aus dem Ge-

biet der Prozessmodellierung stammt. Ein prozessbasiertes Metamodell definiert anstelle einer Modellierungssprache einen Prozess, der zur Modellbildung durchgeführt wird. Dieselben Beziehungen die in Abb. 2.1 zwischen Modellen und Sprachen angegeben sind, sind in einer prozessbasierten Metamodellarchitektur zwischen Modellen und Prozessen vorhanden.

2.3 Modellierung

Nachdem in den vorangegangenen Abschnitten die Begriffe *Modell* und *Metamodell* für diese Arbeit festgelegt wurden, stellt sich die Frage nach der Entstehung eines Modells. Die Tätigkeiten, die ausgeführt werden, um ein Modell zu erzeugen, werden allgemein unter dem Begriff *Modellierung* zusammengefasst. Die Qualität der entstehenden Modelle hängt unmittelbar von der Vorgehensweise ab, die zur Modellierung gewählt wird.

Eine entscheidende Komponente der modellbasierten Software-Entwicklung ist daher der Prozess, der während der Entwicklung eines Softwaresystems durchgeführt wird. Ein Softwareentwicklungsprozess legt Art und Reihenfolge der auszuführenden Tätigkeiten während der Entwicklung eines Softwaresystems fest. Es ist eine Vielzahl von Prozessen entwickelt worden, die die modellbasierte Software-Entwicklung unterstützen. Neben dem unten beschriebenen *Rational Unified Process* sind dies beispielsweise dessen Vorgänger *Objectory* [Jac94], *Booch OOAD* [Boo94] und *OMT* [RBEL91], *Fusion* [CAB94], *Catalysis* [DW99]. Zudem verwendet nahezu jede Firma, die modellbasiert Software entwickelt, eine eigene Entwicklungsmethode, die sich meist aus Bestandteilen der genannten Methoden zusammensetzt.

Unterschiedliche Prozesse verwenden häufig auch verschiedene Notationen. Viele Prozesse wurden bereits vor der Standardisierung der UML entworfen, so dass ihre Entwickler gezwungenermaßen eine von der UML abweichende Notation verwendeten. Außerdem sind Prozesse häufig domänenspezifisch, d. h. an die Anforderungen bestimmter Anwendungsdomänen angepasst. Unterschiedliche Anwendungsgebiete haben in der Regel aber auch unterschiedliche Anforderungen an die Notation. Wenn ein Lösungsvorschlag für spezielle Anforderungen auch allgemein von Nutzen ist, wird er zuweilen in die UML übernommen. Ein Beispiel dafür sind die Kapselklassen aus *ROOM* [SGW94], die eine komponentenorientierte Modellierung auf Ebene der UML-Klassendiagramme ermöglichen und in die Version 2.0 der UML integriert wurden.

Die meisten der in jüngerer Zeit entwickelten Prozesse verwenden allerdings die UML bzw. eine Anpassung der UML, um Modelle zu beschreiben. Zudem ist zu beobachten, dass ältere Prozesse an die UML angepasst werden. Eine Hauptmotivation für die Anpassung ist die Verfügbarkeit von Werkzeugen, die die Software-Entwicklung unterstützen. Ein Beispiel für einen Prozess, der nachträglich auf UML-Notation umgestellt wurde, ist die OCTOPUS-Methode zur Entwicklung eingebetteter Systeme [AKZ96], die zunächst Elemente der Methoden OMT und Fusion verwendete.

Im Folgenden werden zwei modellbasierte Prozesse näher betrachtet. Stellvertretend für die Gruppe der iterativen Prozesse wird der *Rational Unified Process* (RUP) beschrieben. Anschließend wird auf die *Model Driven Architecture* (MDA) eingegangen, die einen stark werkzeug-

unterstützten Prozess definiert, der die Software-Entwicklung weitgehend auf Modellierungstätigkeiten reduziert, indem alle weiteren Schritte automatisiert werden. Modelle rücken dadurch stärker in das Zentrum der Softwareentwicklung.

2.3.1 Rational Unified Process

Der *Rational Unified Process* (RUP) ist ein Software-Entwicklungsprozess, der an die besonderen Anforderungen einzelner Organisationen angepasst werden kann. Die Entstehung des RUP ist eng mit der Entstehung der UML verbunden. Zunächst verfolgten die Gründer der Firma Rational das Ziel, den gesamten Softwareentwicklungsprozess zu vereinheitlichen. Zu diesem Zweck wollten sie die Elemente ihrer jeweiligen Methoden vereinigen. Es zeigte sich, dass dieses Ziel zu ambitioniert war, so dass zunächst nur eine einheitliche Modellierungssprache entwickelt wurde. Diese Modellierungssprache, die UML, wurde explizit ohne Bezug auf ein Vorgehensmodell definiert. In einem nächsten Schritt wurde der *Unified Software Development Process* [JBR99] bzw. RUP [Kru98] vorgestellt.

Grundsätzlich basiert der RUP auf folgenden sechs Prinzipien zur Erstellung von Software:

1. Software soll iterativ entwickelt werden
2. Anforderungen an die Software müssen kontinuierlich verwaltet werden
3. Es sollen komponentenbasierte Architekturen verwendet werden
4. Software sollte visuell modelliert werden
5. Die Softwarequalität muss verifiziert werden
6. Änderungen an der Software müssen verwaltet werden

Der RUP unterscheidet einen Software-Entwicklungsprozess in dynamische und statische Anteile. In Abb. 2.2 ist der Ablauf eines RUP-Prozesses dargestellt. Die horizontale Achse repräsentiert die dynamischen Aspekte des Entwicklungsprozesses. Es sind die verschiedenen Iterationsschritte aufgetragen, die während der Entwicklung durchlaufen werden. Es ist dargestellt, dass vier Phasen durchlaufen werden, die jeweils einige Iterationen enthalten. Die vertikale Achse repräsentiert die statischen Anteile des RUP. Diese bestehen aus den Arbeitsabläufen, die zur Entwicklung der Software durchgeführt werden müssen. Die statischen Anteile entsprechen im Wesentlichen den aus früheren Entwicklungsprozessen bekannten Prozessphasen.

Ein grundlegender Unterschied des RUP gegenüber früheren, linearen Entwicklungsprozessen wie beispielsweise dem V-Modell [Bun97] ist, dass die Software grundsätzlich in Iterationen entwickelt wird. Dieses Vorgehen wurde aufgrund der Erfahrung mit linearen Entwicklungsprozessen gewählt, um einige der immer wieder auftretenden Probleme bei der Softwareentwicklung lösen zu können. Allerdings sieht auch der RUP aufeinanderfolgende Entwicklungsphasen vor und nutzt die Möglichkeiten iterativer Softwareentwicklung daher nicht vollständig aus [Hes00].

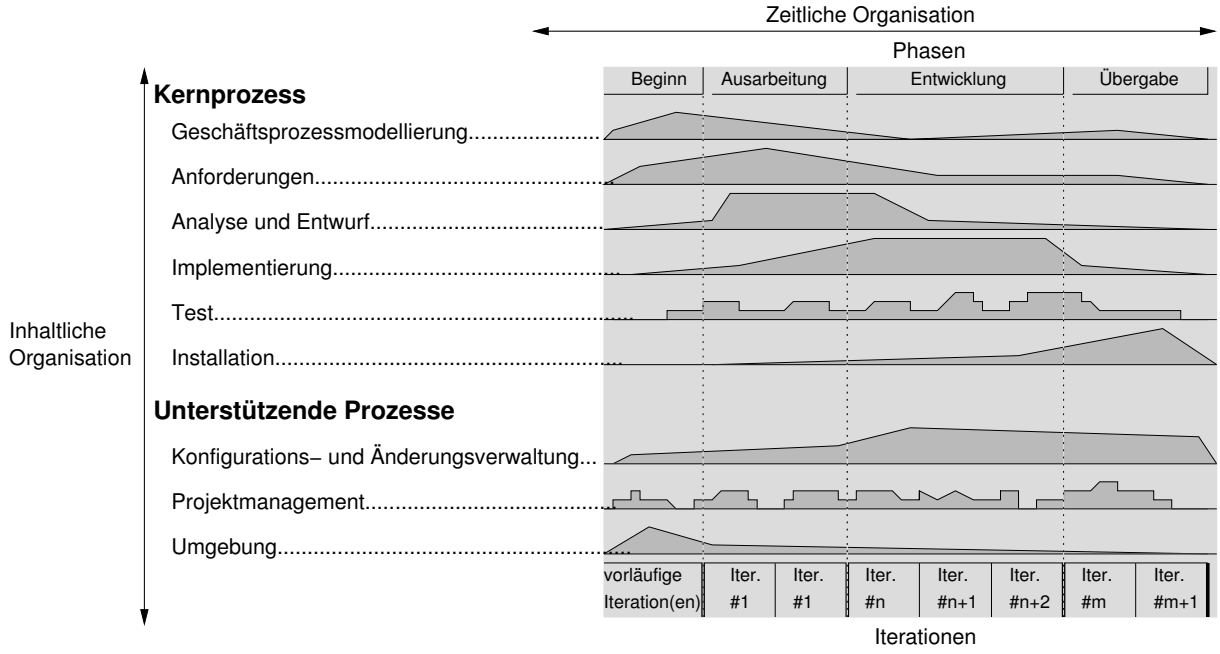


Abbildung 2.2: Dynamische und statische Anteile des Rational Unified Process

Es hat sich gezeigt, dass selten alle Anforderungen an ein Softwaresystem bereits mit Ende der Anforderungsanalyse feststehen. Stattdessen kommen neue Anforderungen hinzu und alte fallen weg oder ändern sich. In einem iterativen Prozess können Veränderungen der Anforderungen einfach bei der nächsten Iteration berücksichtigt werden. Dies allein ist noch kein wirklicher Vorteil gegenüber linearen Prozessen, sondern lediglich eine bessere Integration des Änderungsmanagements in den Prozess. Der tatsächliche Vorteil eines iterativen Prozesses ist, dass frühzeitig Prototypen des zu entwickelnden Systems entstehen. Diese leisten einen wertvollen Beitrag zur Festlegung der Anforderungen, weil der Auftraggeber früher einen Eindruck des Systems bekommt.

Die Integration der verschiedenen Komponenten eines Softwaresystems erfolgt in einem iterativen Prozess am Ende der einzelnen Iterationsschritte. Dies führt zu früherem Erkennen von Problemen gegenüber einem linearen Prozess, der die Integration der Systemkomponenten erst in einer späten Phase der Entwicklung vorsieht. Ähnlich der leichteren Berücksichtigung von sich ändernden Anforderungen resultiert der Vorteil des iterativen Prozesses aus dem frühzeitigen Durchlaufen aller Entwicklungsphasen. Es wird deutlich, dass das frühzeitige Erkennen von Problemen und Risiken der Hauptvorteil eines iterativen Prozesses gegenüber einem linearen Prozess ist.

Nahezu jeder Entwicklungsprozess beginnt mit der Analyse der Anforderungen an das zu entwickelnde System. Diese Phase ist sehr wichtig, weil alle nachfolgenden Entwicklungsphasen auf den Ergebnissen der Anforderungsanalyse aufbauen. Der RUP definiert einen Arbeitsablauf, der die systematische Erhebung und Verwaltung der Anforderungen an ein Softwaresystem ermöglicht.

In der Designphase des RUP wird die Architektur der Software festgelegt. In den ersten Iterationen wird eine Architektur entwickelt, die in späteren Iterationen in einen lauffähigen Prototypen umgesetzt wird. Dieser Prototyp wird in nachfolgenden Iterationen zum fertigen System verfeinert. Der RUP enthält Vorlagen zur Beschreibung einer Systemarchitektur, die verschiedene Sichten auf ein System bereitstellen. Zudem definiert der Prozess Aktivitäten zur Identifikation signifikanter Elemente und enthält Richtlinien zur Unterstützung von Entwurfsentscheidungen.

Der RUP sieht vor, dass ein Softwaresystem aus Komponenten besteht. Eine Komponente wird als „nicht-trivialer Bestandteil einer Software, ein Modul, ein Paket, oder ein Subsystem“ definiert, „das eine bestimmte Funktion erfüllt, klar eingegrenzt ist und in eine bestimmte Architektur integriert werden kann“ [Kru98, Seite 27]. Komponenten im Sinne des RUP können unabhängig voneinander entwickelt und getestet werden, so dass die Entwicklungsgeschwindigkeit erhöht wird. Außerdem können gut entworfene Komponenten wiederverwendet werden. Ein weiteres Argument für die Verwendung von Komponenten ist die gute Unterstützung durch Infrastrukturen wie OMG's *Common Object Request Broker Architecture* (CORBA), Microsoft's *.NET* und Sun's *Enterprise Java Beans* (EJB).

Die Ergebnisse aller Aktivitäten werden während der Ausführung des RUP durch Modelle dokumentiert. Ein großer Teil des RUP beschäftigt sich daher mit der Erstellung und Wartung von Modellen eines zu entwickelnden Systems. Es wird unterschieden zwischen Anwendungsfallmodellen, Analyse- und Designmodellen sowie Testmodellen[1]. Die verwendete Modellierungssprache ist in allen Prozessphasen die Unified Modeling Language (UML) [UP03b].

Der RUP unterscheidet zwischen Produktqualität und Prozessqualität. Die Produktqualität ist die Qualität des Softwaresystems, das während der Durchführung des RUP entwickelt wird. Dies umfasst alle Aspekte des Systems, also sowohl die einzelnen Komponenten als auch die Architektur. Die Prozessqualität wird einerseits durch die Qualität des Prozesses, der zur Erstellung der Software durchgeführt wird, und andererseits durch die Artefakte, die zur Unterstützung des Produktes erstellt werden, bestimmt. Letztere umfassen insbesondere die Modelle, die in den verschiedenen Prozessphasen erstellt werden. Sowohl die Produkt- als auch die Prozessqualität wird vom RUP durch einen Test-Arbeitsablauf unterstützt, währenddessen die Qualität der erstellten Artefakte überprüft wird.

Ein weiterer Bestandteil des RUP ist die Verwaltung von Konfigurationen und Änderungen der Software. Aufgrund der iterativen Entwicklung verändern sich viele der erstellten Artefakte von Iteration zu Iteration. Diese Änderungen müssen verwaltet werden, um die Synchronisation zwischen den verschiedenen Systemkomponenten zu gewährleisten. Ähnliches gilt für Konfigurationen, die zur Anpassung eines Systems an verschiedene Anwendungsfälle durchgeführt werden.

Eine Grundvoraussetzung für die effiziente Durchführung des RUP ist die Unterstützung der verschiedenen Phasen durch geeignete Werkzeuge zur Erzeugung der Artefakte. Neben Werkzeugen zur Erfassung und Verwaltung der Anforderungen sind dies Werkzeuge zur Erstellung und Wartung der Modelle, insbesondere grafische Modellierungs- und Programmierwerkzeuge.

[1] Außerdem wird noch ein weiterer Modelltyp definiert, der Strukturen und Abläufe in einem Unternehmen darstellt. Dieser Modelltyp ist hauptsächlich für den Bereich geschäftlicher Anwendungen interessant.

Durch den Einsatz dieser Werkzeuge kann der Entwicklungsprozess zumindest teilautomatisiert werden. Die Implementierung entsprechender Werkzeuge wird durch den MOmo-Baukasten erleichtert. Eine weitergehende Automatisierung sieht die sog. *Model Driven Architecture* vor, die im folgenden Abschnitt beschrieben wird.

2.3.2 Model Driven Architecture

Die *Model Driven Architecture* (MDA) definiert einen Modell-zentrischen Ansatz zur Entwicklung von Softwaresystemen. Der Aufbau der MDA wird durch die *MDA-Spezifikation* [MM01] festgelegt und durch den *MDA-Guide* [MM03] erläutert. Die *Object Management Group* (OMG) versteht die MDA als Bindeglied zwischen den verschiedenen OMG-Standards. Abb. 2.3 zeigt die Rollen verschiedener OMG- und Nicht-OMG-Technologien innerhalb der MDA.

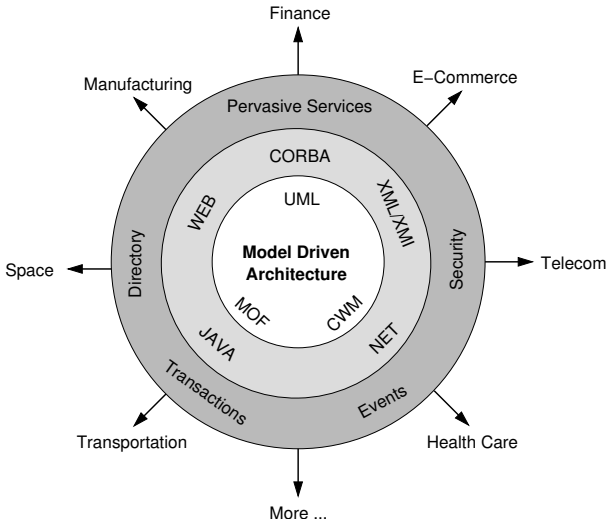


Abbildung 2.3: Komponenten der MDA (nach [MM01])

Die Rolle des Kerns der MDA spielen die OMG-Modellierungssprachen *Common Warehouse Metamodel* (CWM), *Meta Object Facility* (MOF) und UML. Diese werden verwendet, um Anwendungen zu modellieren. Je nach Anwendungstyp wird eine geeignete Modellierungssprache ausgewählt. Zur Implementierung der Modelle werden jeweils aktuelle Implementierungstechnologien verwendet. Einige derzeit aktuelle Implementierungstechnologien sind im hellgrauen Ring in Abb. 2.3 enthalten. Es ist eines der Hauptanliegen der MDA, die Software-Entwicklung von den verwendeten Implementierungstechnologien zu entkoppeln, so dass die angegebenen Technologien nur als Beispiele zu verstehen sind. Im dunkelgrau dargestellten Ring sind externe Dienste angegeben, die eine Applikation nutzen kann. Die angegebenen Dienste sind wiederum nur als Beispiel zu verstehen; sie entsprechen den derzeit spezifizierten CORBA-Diensten. Die außenliegenden Pfeile verweisen auf die Anwendungsgebiete der MDA und zeigen, dass die OMG die Anwendung der MDA in nahezu allen denkbaren Anwendungsgebieten vorsieht.

Die MDA definiert einen Software-Entwicklungsprozess, der einen Weg zur automatischen Erzeugung einer Anwendung aus Modellen vorgibt. Die Idee hinter der MDA ist, dass die Automatisierung

- die Nutzung neuer Implementierungstechnologien erleichtert,
- die Integration von Daten und Anwendungen vereinfacht,
- die Wartbarkeit verbessert, und
- Testen und Simulation an laufenden Anwendungen ermöglicht.

Es ist leicht zu erkennen, dass sich die Ziele der MDA nicht wesentlich von den Zielen anderer Ansätze zur modellbasierten Software-Entwicklung unterscheiden. Der Unterschied ist lediglich die Fokussierung auf die automatische Erzeugung des Codes für die Anwendungen. Diese Fokussierung beseitigt das Synchronisationsproblem zwischen den Modellen, die in den frühen Phasen traditioneller Entwicklungsprozesse entworfen werden, und den Implementierungen der Modelle, die in späteren Phasen entstehen. Häufig werden in der Implementierungsphase Änderungen am Entwurf einer Software vorgenommen. Dadurch entsteht zumindest zeitweise ein inkonsistenter Zustand, der, insbesondere bei überlappenden Entwurfs- und Implementierungsphasen, zu erheblichem zusätzlichem Aufwand führt. Außerdem werden Änderungen, die in der Implementierungsphase durchgeführt werden, oft durch die verwendete Implementierungsplattform hervorgerufen. Solche unerwünschten Szenarios sollen durch die MDA vermieden werden, indem die Implementierungen vollautomatisch aus den Modellen erzeugt werden sollen. Alle Änderungen werden an den Modellen durchgeführt.

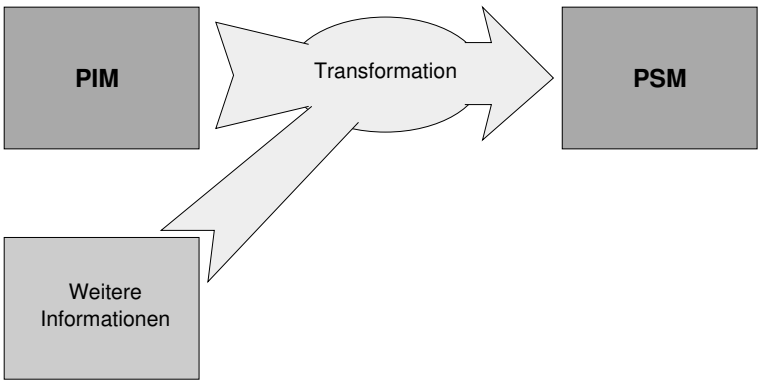


Abbildung 2.4: Grundprinzip der MDA (nach [MM01])

Abb. 2.4 zeigt das Grundprinzip der MDA. Ein plattformunabhängiges Modell (PIM[2]) wird mit weiteren Informationen kombiniert, um ein plattformspezifisches Modell (PSM) zu erzeugen. Ein PIM beschreibt die Funktionalität eines Systems und abstrahiert vollständig von der

[2]engl. **P**latform **I**ndependent **M**odel

Implementierung. Ein PSM ist eine Abbildung eines PIMs auf eine bestimmte Implementierungstechnologie, aber noch keine konkrete Implementierung. Es kann sowohl beliebig viele PIMs als auch beliebig viele PSMs desselben Systems geben. Verschiedene PIMs, die dasselbe System modellieren, zeigen meist verschiedene Sichten auf das System.

Durch die Abstraktion von technischen Details sollen Validierung, Portierung und Integration bzw. Interoperabilität eines Systems verbessert werden. Die Validierung wird erleichtert, weil man plattformabhängige Eigenschaften bestimmter Mechanismen nicht berücksichtigen muss. Beispiele für entsprechende Mechanismen sind Ausnahmebehandlung, verfügbare Parametertypen oder Komponentenmodelle. Stattdessen kann ein einfaches, einheitliches Modell zur Validierung der Funktionalität verwendet werden.

Die Portierung eines Systems auf eine neue Plattform wird deutlich einfacher. Weil die Funktionalität eines Systems durch ein PIM ohne Berücksichtigung der Eigenschaften einer Implementierungstechnologie beschrieben wird, liegen präzise Definitionen der Struktur und des Verhaltens eines zu portierenden Systems vor. Diese Definitionen lassen sich leichter auf eine neue Implementierungstechnologie abbilden als eine mit Eigenschaften einer anderen Implementierungstechnologie durchsetzte Systemspezifikation. Aus demselben Grund werden Integration und Interoperabilität eines Systems durch eine plattformunabhängige Systemspezifikation verbessert.

Besonders geeignet zur Spezifikation eines PIMs ist gemäß der MDA-Spezifikation die UML. Dies überrascht nicht, werden MDA und UML doch beide durch die OMG spezifiziert. Es ist allerdings ebenso möglich, eine andere deklarative Sprache einzusetzen, um ein PIM zu beschreiben. Die in der Spezifikation genannten Beispiele anderer geeigneter Sprachen umfassen insbesondere Sprachen, die explizit zur Beschreibung von Schnittstellen entwickelt wurden (CORBA-IDL, Microsoft-IDL). Allerdings haben diese Sprachen bereits einen klaren Bezug zu der jeweiligen Implementierungstechnologie, so dass die klare Trennung zwischen PIM und PSM bei ihrer Anwendung ein Stück weit aufgegeben wird. Es ist daher vorzuziehen, die UML oder eine vergleichbare Modellierungssprache zu verwenden. Aus diesem Grund sind die Modellierungssprachen in Abb. 2.3 auch als Kern der MDA dargestellt.

Die Hauptaktivitäten bei der Software-Entwicklung mit der MDA sind Transformationen zwischen plattformunabhängigen und plattformspezifischen Modellen. Die Tranformationsarten umfassen neben der oben beschriebenen Transformation eines PIMs in ein PSM auch alle anderen möglichen Kombinationen von plattformunabhängigen und plattformspezifischen Modellen. Der Zweck einer Transformation eines PIMs in ein anderes PIM kann beispielsweise die Erweiterung eines Analysemodells zu einem Entwurfsmodell sein. Transformationen zwischen PSMs werden durchgeführt, um plattformspezifische Komponenten mit weiteren Informationen zu versorgen, die zum Einsatz der Komponenten benötigt werden. Diese Informationen umfassen Initialisierungsdaten und Konfigurationen. Die Umwandlung eines PSMs in ein PIM entspricht einem *Reverse Engineering*. Das Modell wird von allen plattformabhängigen Informationen „befreit“ und kann leichter durch weitere Transformationen auf andere Plattformen portiert werden.

Das Hauptproblem der MDA ist ihre sehr abstrakte Beschreibung, die lediglich einen groben Ablauf skizziert, aber keinerlei Hinweise auf geeignete Techniken und Werkzeuge enthält, um

die notwendigen Transformationen durchzuführen. Die MDA löst daher keine konkreten Probleme, die bei der Softwareentwicklung auftreten. Allerdings hat die MDA ein deutlich erhöhtes Interesse an Ansätzen und Werkzeugen zur Transformation von bzw. Codeerzeugung aus (UML-)Modellen bewirkt, da das Erreichen der Ziele der MDA in noch höherem Maße von der Verfügbarkeit geeigneter Sprachen und Werkzeuge zur Modelltransformation und Codeerzeugung abhängig ist als die aktuellen, modellbasierten Software-Entwicklungsprozesse. Aufgrund des gesteigerten Interesses wurde in letzter Zeit eine Vielzahl von Lösungsansätzen vorgestellt, die den MDA-Rahmen mit konkreten Techniken und Werkzeugen auszufüllen versuchen, um das Hauptziel der MDA, die Steigerung des Abstraktionsniveaus bei der Softwareentwicklung, zu erreichen. Im folgenden Kapitel wird ein Überblick über die zur Zeit aktuellen Ansätze und Werkzeuge im MDA-Umfeld gegeben.

2.4 Zusammenfassung

In diesem Kapitel wurden die wesentlichen Bestandteile der modellbasierten Softwareentwicklung beschrieben. Zunächst wurden die Begriffe *Modell* und *Metamodell* festgelegt. Ein Modell wird im Weiteren als *formale Beschreibung eines Softwaresystems* angesehen und ein Metamodell als *Modell, das die Syntax der Modellelemente einer Klasse von Modellen festlegt*, betrachtet. Obwohl beide Definitionen recht ungenau sind, reicht ihre Genauigkeit für den in dieser Arbeit beschriebenen Anwendungsfall aus.

Modelle und auch Metamodelle müssen systematisch erstellt werden, um eine gute Qualität zu erreichen. Die Qualität eines Modells ist das Maß an Ähnlichkeit zwischen dem Modell und dem abgebildeten System oder Prozess in Bezug auf die anwendungsrelevanten Eigenschaften. Der Prozess, der bei der modellbasierten Software-Entwicklung ausgeführt wird, hat maßgeblichen Einfluss auf die Qualität sowohl der Modelle als auch der resultierenden Systeme. Der derzeitige Stand der Technik auf dem Gebiet der Software-Entwicklungsprozesse sieht eine werkzeugunterstützte, iterative Vorgehensweise vor. Als Beispiel für eine solche Vorgehensweise wurde der *Rational Unified Process* beschrieben.

Zukünftig werden Modelle in noch stärkerem Maße in den Mittelpunkt der Software-Entwicklung gelangen. Die Vision ist, die Umsetzung eines Modells in ein lauffähiges System zu automatisieren. Dazu definiert die *Model Driven Architecture* eine schrittweise Umsetzung eines plattformunabhängigen Modells in ein plattformspezifisches Modell, das abschließend zur automatischen Erzeugung lauffähiger Komponenten und/oder Systeme in einer bestimmten Implementierungstechnologie verwendet wird. Die Umsetzung soll durch Modelltransformatoren und Codegeneratoren vollständig automatisiert werden. Die Entwicklung von Modelltransformatoren und Codegeneratoren für Modellierungssprachen ist daher eine Kernaufgabe, die bewältigt werden muss, um die MDA zu realisieren.

Kapitel 3

Literaturüberblick

Modellbasierte Entwicklungsmethoden und -werkzeuge versuchen in immer stärkerem Maße, Anwendungssysteme aus Analyse- und Designmodellen automatisch zu erstellen. Die *Model Driven Architecture* (MDA), die in Abschnitt 2.3.2 vorgestellt wurde, beschreibt ein Vorgehensmodell, um Softwaresysteme aus Modellen schrittweise in Implementierungen zu übertragen. Zunächst soll ein plattformunabhängiges Modell erstellt werden, das Struktur und Verhalten einer Anwendung unabhängig von implementierungsspezifischen Aspekten beschreibt. Das plattformunabhängige Modell kann mit den implementierungsspezifischen Informationen angereichert werden. Das entstandene plattformspezifische Modell dient schließlich als Basis der Codegenerierung, die Implementierungen der Modelle erzeugt.

Sowohl die Anreicherung eines plattformunabhängigen Modells mit implementierungsspezifischen Informationen als auch die abschließende Codeerzeugung sind geeignet, um durch (automatische) Transformationen durchgeführt zu werden. Solchen automatischen Transformationen wird allgemein eine große Bedeutung zugemessen, weil der Abstraktionsgrad bei der Softwareentwicklung durch die Verfügbarkeit von Transformatoren und Generatoren für Modellierungssprachen erhöht werden kann. Aufgrund der großen Bedeutung sind in letzter Zeit viele Ansätze zur Modelltransformation und Codegenerierung entworfen und (prototypisch) implementiert worden.

In diesem Kapitel werden die Entwicklungen in den Bereichen Modelltransformation und Codegenerierung zusammengefasst. Zunächst erfolgt eine Festlegung der Begriffe *Modelltransformation* und *Codegenerierung* für die weitere Arbeit (Abschnitt 3.1). Anschließend werden die Ansätze zur Modelltransformation (Abschnitt 3.2) und Codegenerierung (Abschnitt 3.3) vorgestellt und bewertet. Die Ergebnisse der Bewertungen bilden die Grundlage der Anforderungen für den MOmo-Baukasten, der in Kapitel 4 vorgestellt wird.

3.1 Klassifikation

Im Allgemeinen kann eine Modelltransformation als Abbildung eines Quellmodells auf ein Zielmodell verstanden werden. Diese Auffassung schließt sowohl die Transformation eines plattformunabhängigen in ein plattformspezifisches Modell als auch die Erzeugung des Codes einer Programmiersprache aus einem plattformspezifischen Modell ein. Um zwischen Modelltransformation und Codegenerierung unterscheiden zu können, müssen die Beziehungen zwischen Quell- und Zielmodell einer Transformation näher betrachtet werden.

Quell- und Zielmodell lassen sich in derselben Sprache oder in verschiedenen Sprachen beschreiben. Wenn die Modelle in derselben Sprache beschrieben werden, werden Konstrukte einer Sprache auf Konstrukte derselben Sprache abgebildet. Durch solche Transformationen könnte beispielsweise der Modellierungsstil verschiedener Modelle einander angepasst oder ein Modell mit plattformspezifischen Details angereichert werden. Wenn Quell- und Zielmodell dagegen in unterschiedlichen Sprachen beschrieben sind, müssen die Elemente der beiden Sprachen aufeinander abgebildet werden.

Ein typisches Szenario für letztgenannte Transformationen ist die metamodellbasierte Werkzeugintegration. Wenn zwei Werkzeuge verschiedene Sprachen zur Definition von Modellen verwenden, muss zum Datenaustausch zwischen den Werkzeugen zumindest die Repräsentation des Modells verändert werden. Die Semantik des Modells sollte dagegen erhalten bleiben bzw. durch die Transformation ein Modell mit „korrekter“ Semantik erzeugt werden. Die Modelle, die mit den Werkzeugen erstellt werden, werden jeweils durch die Elemente des Metamodells beschrieben, so dass sich durch Anwendung der Transformation ein Modell aus der Repräsentation für ein Werkzeug in die Repräsentation eines weiteren Werkzeugs überführen lässt.

Ein weiteres mögliches Szenario für eine Verwendung einer Modelltransformation ist die Transformation eines Modells, das in einer domänenspezifischen Sprache beschrieben ist, in ein Modell, das die Elemente einer Programmiersprache verwendet. Eine solche Transformation kann metamodellbasiert durchgeführt werden, wenn sowohl die domänenspezifische Sprache als auch die Programmiersprache durch ein Metamodell definiert wird, so dass sich die Validierungs- und Verifikationsmechanismen der Modelltransformationsansätze nutzen lassen. Das Ergebnis ist ein Modell, dessen Elemente direkt mit den Elementen einer Implementierungstechnologie korrespondieren.

Um die Implementierung automatisch erzeugen zu können, wird ein weiterer Transformationsschritt benötigt, der das Modell in Artefakte der Implementierungstechnologie übersetzt. Diese Transformation unterscheidet sich von den zuvor beschriebenen, weil die Struktur der Artefakte nicht durch ein Metamodell beschrieben wird. Um existierende (Compiler-) Technologien nutzen zu können, wird Quellcode der entsprechenden Programmiersprache erzeugt und in einem abschließenden Schritt in die eigentliche Implementierung übersetzt. Der Sprachumfang einer Programmiersprache wird fast immer durch eine Grammatik festgelegt, so dass es zunächst nahe liegt, Modelltransformation und Codegenerierung durch die Art der Festlegung der Sprache, die zur Beschreibung des Zielmodells verwendet wird, zu unterscheiden.

Ähnlich argumentieren Czarnecki und Helsen in ihrem Überblick über Ansätze zur Codegenerierung und Modelltransformation [CH03]. Sie betrachten die Codegenerierung aus einem Modell als Sonderfall einer Modelltransformation, weil der einzige Unterschied zwischen den beiden Transformationen in der Definition der Sprache besteht, die zur Beschreibung des Zielmodelles verwendet wird. Czarnecki und Helsen bezeichnen Codegenerierung deshalb als „Modell-zu-Text"-Transformation und argumentieren, dass die Sprache, die innerhalb des Textes verwendet wird, nur durch ein Metamodell definiert zu werden müsse, um aus der „Modell-zu-Text"- eine „Modell-zu-Modell"-Transformation zu machen. Auf diese Weise lassen sich auch grammatikbasierte Ansätze zur Definition der abstrakten Syntax von Programmiersprachen als Metamodellierung betrachten. Genauso gut lässt sich allerdings auch in der entgegengesetzten Richtung argumentieren, da sich Metamodelle durch textuelle Beschreibungen repräsentieren lassen. Die Unterscheidung zwischen Text und Modell ist daher wenig geeignet, um eine Definition der Begriffe Modelltransformation und Codegenerierung festzulegen.

Aus diesem Grund werden die oben genannten Eigenschaften herangezogen, um die Begriffe „Modelltransformation" und „Codegenerierung" für die weitere Arbeit zu definieren:

Definition 5 *Eine **Modelltransformation** verwendet die Abbildung eines Metamodells A auf ein Metamodell B, um Instanzen von A in Instanzen von B auf der Ebene der abstrakten Syntax ineinander zu überführen. Die konkreten Repräsentationen der Instanzen werden erst in einem nachfolgenden Schritt erzeugt.*

Definition 6 ***Codegenerierung** bezeichnet die Abbildung eines Metamodells auf die Artefakte einer Programmiersprache oder Implementierungstechnologie, wobei konkrete textuelle Repräsentationen erzeugt werden.*

Wie beschrieben lässt sich die Erzeugung von Code einer Programmiersprache als eine spezielle Modelltransformation verstehen, die sich durch die Konzentration auf die konkrete Syntax der Zielsprache von anderen Modelltransformationen unterscheidet. Aufgrund der engen Verwandtschaft von Modelltransformation und Codegenerierung werden im Folgenden sowohl Ansätze zur Modelltransformation als auch Ansätze zur Codegenerierung vorgestellt.

3.2 Modelltransformation

Seit der Publikation der MDA sind viele Ansätze zur Durchführung von Modelltransformationen vorgeschlagen worden. Einen Überblick über die derzeit aktuellen Entwicklungen geben Sendall und Kozaczynski [SK03] sowie Czarnecki und Helsen [CH03]. Erstgenannte konzentrieren sich auf Werkzeuge zur Modelltransformation und klassifizieren diese, indem sie folgende Eigenschaften untersuchen:

- Ermöglicht ein Werkzeug die direkte Manipulation eines Modells? Dazu müssen anwenderseitige Zugriffe auf die interne Repräsentation des Modells unterstützt werden.

- Enthält das Werkzeug eine standardisierte Schnittstelle zur Repräsentation von Modellen, so dass es die Durchführung einer Modelltransformation durch ein externes Werkzeug ermöglicht?
- Stellt das Werkzeug eine Sprache bereit, die zur Definition und Durchführung von Modelltransformationen verwendet werden kann?

Jedes Werkzeug, das mindestens einen der genannten Mechanismen unterstützt, betrachten Sendall und Kozaczynski als Modelltransformationswerkzeug. Wird dieser Argumentation gefolgt, ist nahezu jedes CASE-Werkzeug auch ein Modelltransformationswerkzeug. Dies gilt insbesondere für die UML-basierten Werkzeuge, weil diese in der Regel XMI-Dokumente lesen und schreiben können. Genau genommen stellen die Werkzeuge, die lediglich eine standardisierte Schnittstelle anbieten, aber keinerlei Unterstützung für Modelltransformationen bereit. Daher werden sie im Weiteren nicht berücksichtigt.

Werkzeuge, die ausschließlich die direkte Manipulation der internen Repräsentation eines Modells ermöglichen, können ähnlich beurteilt werden. Da einige dieser Werkzeuge allerdings neben dem Zugriff auf die interne Repräsentation noch weitere Mechanismen zur Durchführung von Modelltransformationen enthalten, werden Werkzeuge dieser Kategorie im folgenden Überblick über die verfügbaren Ansätze zur Modelltransformation berücksichtigt.

Eine entsprechende Berücksichtigung erfolgt auch bei Czarnecki und Helsen [CH03], die die verschiedenen Ansätze zur Durchführung von Modelltransformationen anhand der Konzepte klassifizieren, die zur Beschreibung der Transformationen verwendet werden. Sie unterscheiden folgende Techniken:

- Direkte Programmierung
- Deklarative Ansätze
- Graphtransformationen
- Strukturtransformation

Zusätzlich werden hybride Ansätze, die mehrere der genannten Techniken vereinen, und weitere Ansätze, die eine bestimmte Implementierungstechnologie anwenden, betrachtet. In der folgenden Betrachtung wird diese Aufteilung übernommen. Allerdings werden die strukturellen, hybriden und technologiespezifischen Ansätze zusammengefasst, da es nur wenige Ansätze gibt, die in eine dieser Kategorien eingeordnet werden müssen.

3.2.1 Direkte Programmierung

Die Ansätze, die eine direkte Programmierung von Modelltransformationen ermöglichen, bestehen im Wesentlichen aus Rahmenwerken, die die Repräsentation von Metamodellen ermöglichen und standardisierte Schnittstellen zum Zugriff auf die Metamodelle bereitstellen. So gesehen sind alle Implementierungen des MOF-Standards (siehe Kapitel 5), unter anderem *nsmdr*

[Plo02], *NetBeans MDR* [Mat03] und *dMOF* [DST02], bereits Ansätze zur Modelltransformation.

Einen geringfügig komfortableren Ansatz implementiert *Jamda* [Boo03], das zusätzlich eine Infrastruktur zur Organisation der Transformationsregeln bereitstellt. Auch bei der Verwendung von Jamda müssen aber alle Transformationsregeln manuell implementiert werden. Zudem ist Jamda im Unterschied zu den Implementierungen des MOF-Standards auf Transformationen zwischen UML-Modellen fokussiert. Erweiterungen des UML-Metamodells, die nicht durch Stereotypen ausgedrückt werden können, müssen manuell implementiert werden.

Direkte Programmierung ist der flexibelste aber auch der aufwändigste Ansatz, um Modelltransformationen zu implementieren. Aufgrund des hohen Aufwandes bietet sich dieser Ansatz nur dann an, wenn für eine häufig durchzuführende oder besonders komplexe Transformation eine besonders effiziente Implementierung benötigt wird.

3.2.2 Deklarative Ansätze

Die Gruppe der deklarativen Ansätze definiert zunächst die Typen der Quell- und Zielelemente einer Relation. Anschließend wird die Relation durch die Angabe von Bedingungen spezifiziert. Rein deklarative Ansätze sind nicht ausführbar und somit nur zur Spezifikation von Modelltransformationen geeignet. Dieser Nachteil wird in den meisten Ansätzen ausgeglichen, indem eine geeignete Technologie zur Implementierung zusätzlich angegeben wird. Die meisten Autoren deklarativer Ansätze schlagen die Verwendung logischer Programmiersprachen zur Implementierung der Transformationen vor.

Beispiele für deklarative Ansätze sind [AKP03], [GLR+02], [AST+03], [DMC03] und [QP03]. Auffällig ist, dass die letzten drei Vorschläge jeweils Vorschläge für den *„MOF 2.0 Query/-Views/Transformations“*-Standard (QVT) sind. Alle deklarativen Ansätze sind frei von Seiteneffekten. Dies ist ihr wesentlicher Vorteil gegenüber den anderen Ansätzen zur Implementierung von Modelltransformationen.

Akehurst u. a. verwenden binäre Relationen bzw. Mengen binärer Relationen und beschreiben diese mit Objektmodellen, um Abbildungen zwischen einem Modell der abstrakten Syntax und einem Modell der Semantik zu definieren [AKP03]. Die Motivation ist, sowohl Syntax als auch Semantik einer Modellierungssprache durch Metamodelle beschreiben zu können. Der vorgestellte Ansatz wird auch als Basis für die Implementierung von Werkzeugen verwendet. Allerdings ist noch unklar, ob sich die beschriebene Vorgehensweise auch für die Definition komplexer Sprachen eignet, da das Verfahren bisher lediglich für einen kleinen UML-Ausschnitt angewendet wurde.

Gerber u. a. untersuchen die Eignung verschiedener Ansätze zur Implementierung von MDA-Transformationen und kommen zu dem Ergebnis, dass ein deklarativer Ansatz zu bevorzugen ist, weil die Transformationsregeln vergleichsweise einfach zu verstehen sind [GLR+02]. Dies wird durch den Wegfall von Anwendungsreihenfolge und Terminierungssemantik begründet. Daher wird vorgeschlagen, eine logische Programmiersprache zur Implementierung von Transformationsregeln zu verwenden. In einem ersten Ansatz verwendeten Gerber u. a. die rein de-

klarative, logische Programmiersprache *Mercury* [SHC95]. Aufgrund von Problemen bei der Abbildung der MOF-Sematik auf das Mercury-Typsystem wurde später die Sprache *F-Logic* [KLW95] verwendet.

F-Logic wird auch als Basis für eine prototypische Implementierung des in [DMC03] beschriebenen Ansatzes verwendet. Der Ansatz der QVT-Partner [QP03] unterscheidet dagegen zwischen Relationen, die als bidirektionale, nicht-ausführbare Spezifikationen betrachtet werden, und Abbildungen, die ausführbare, gerichtete Transformationen repräsentieren und die Relationen implementieren. Der dritte QVT-Ansatz [AST+03] enthält neben den deklarativen auch imperative Anteile und ist deshalb den hybriden Ansätzen zuzuordnen, die in Abschnitt 3.2.4 vorgestellt werden.

3.2.3 Graphtransformationsbasierte Ansätze

Graphtransformationen sind regelbasierte Manipulationen beliebiger markierter Graphen und Hypergraphen. Sie wurden entworfen, um die Vorzüge von Graphen und Regeln in einem Modell zu vereinen. Graphen werden in der Informatik und anderen Disziplinen verwendet, um Beziehungen zwischen Objekten darzustellen. Die Regeln erlauben eine deklarative Programmierung (siehe Abschnitt 3.2.2) mit vergleichsweise einfacher Syntax und Semantik.

Als potentielle Anwendungsgebiete werden hauptsächlich visuelle Programmierung und formale Spezifikation von Softwaresystemen angegeben. Die Verwendung von Graphtransformationen zur Implementierung von Modelltransformationen liegt daher nahe, wenn die Modelle durch visuelle Sprachen beschrieben werden. Dementsprechend wurden Graphtransformationen bereits lange vor Erscheinen der MOF angewendet, um Modelltransformationen durchzuführen. Im Rahmen des IPSEN-Projektes [Nag96] werden u. a. sog. *Triple Graph Grammars* (TGG) zur Integration von Werkzeugen vorgeschlagen. Eine TGG besteht aus zwei Graphgrammatiken, die jeweils Mengen von Graphen beschreiben, und einer dritten Graphgrammatik, die Beziehungen zwischen den Knoten dieser Graphen beschreibt [Sch94]. Das Werkzeug PROGRES [SWZ95] wird innerhalb des IPSEN-Projektes zur Implementierung von Graphtransformationen verwendet.

Auch MOF und UML werden in der Regel grafisch notiert, so dass entsprechend viele der in letzter Zeit vorgeschlagenen Ansätze zur Durchführung von Modelltransformation zwischen UML- bzw. MOF-Modellen auf Graphtransformationen basieren. Im Folgenden werden einige dieser Ansätze vorgestellt, um die grundlegenden Eigenschaften dieser Kategorie von Modelltransformationsansätzen zu zeigen.

Die *Bidirectional Object Oriented Transformation Language*[1] (BOTL) [BM03a, MB03] formalisiert die UML-Klassendiagramme, um auf dieser Basis Modelltransformationen zu beschreiben. BOTL wurde entwickelt, um metamodellbasierte Werkzeugintegrationen durchführen zu können. Die Transformationen werden durch Paare von UML-Objektdiagrammen definiert, die jeweils Fragmente des Quellmodells auf Fragmente des Zielmodells abbilden. Eine prototypi-

[1] auch als *Basic* Object Oriented Transformation Language bezeichnet [BM03b]

sche Implementierung der Sprache wurde auf Basis des UML-Werkzeuges ArgoUML [Arg] realisiert.

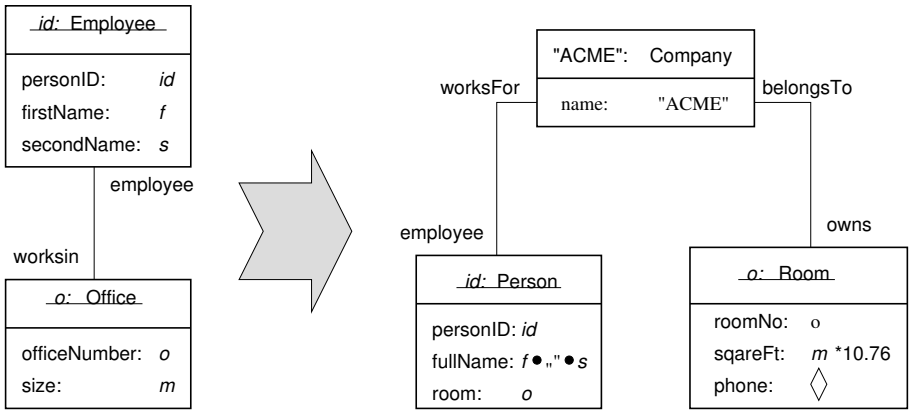


Abbildung 3.1: Beispiel einer BOTL-Transformation [BM03b]

Abb. 3.1 zeigt eine BOTL-Regel, die eine Transformation zwischen zwei unterschiedlichen Abbildungen der Organisationsstruktur eines Unternehmens festlegt. Die Attribute der neu erzeugten Objekte des Zielmodells werden aus den Attributen der Objekte des Quellmodells ermittelt. Leider bietet BOTL keine Möglichkeiten zur Strukturierung von Regelsätzen. Zudem bleibt unklar, wie sich die parallele Anwendung mehrerer Regeln mit sich überlappenden Quellmodellen auswirkt.

Die *Graph Rewriting and Transformation Language* (GReAT) [KASS03] enthält drei Subsprachen zur Definition von Mustern, Graphtransformationen und Kontrollflüssen. Muster werden durch eine grafische Spezifikationssprache definiert, deren wesentliche Mechanismen den UML-Klassendiagrammen ähneln. Die Muster geben die Fragmente der Quell- und Zielmodelle an, die aufeinander abgebildet werden sollen. Die GReAT-Regeln binden Aktionen an Muster. Die an Muster gebundenen Aktionen werden immer dann ausgeführt, wenn ein Muster im Quellmodell gefunden wird. GReAT definiert drei Aktionen zum Erhalten, Löschen und Neuerzeugen von Objekten in einem Graph. Um Auswahl und Anwendung der Regeln zu steuern, wird die Kontrollflusssprache eingesetzt.

Die in Abb. 3.2 dargestellte Regel erzeugt für jedes Auftreten der links von der gestrichelten Linie dargestellten Konstellation von (Oder-) Zuständen einen neuen Zustand. Der neue Zustand wird als Kind des oberen Oder-Zustandes auf der linken Regelseite angelegt. Im Unterschied zu BOTL wird die Transformation der Attribute einer Klasse explizit dargestellt. Einerseits wird die Transformation der Attribute dadurch leichter erkennbar, andererseits wird das Diagramm aber auch umfangreicher, so dass die Probleme beim Darstellen komplexer Transformationen noch einmal wachsen.

UMLX [Wil03] ist eine grafische Modellierungssprache zur Beschreibung von Modelltransformationen auf der Basis von UML-Klassendiagrammen. Die Klassendiagramme werden um die in Abb. 3.3 dargestellten Konstrukte erweitert, um Transformationen beschreiben zu können. Modelltransformationen, die durch UMLX beschrieben werden, bestehen aus sechs Elementen.

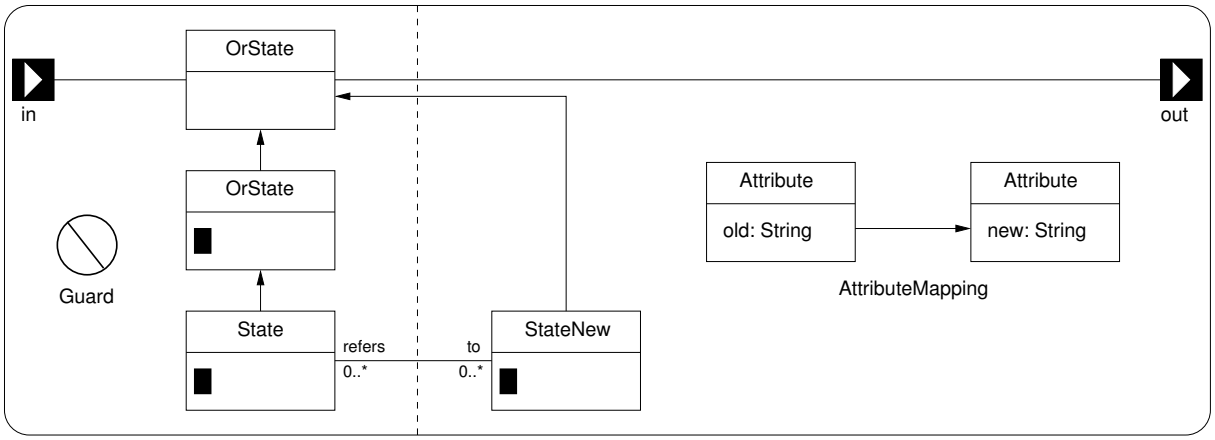


Abbildung 3.2: Eine Regel in GReAT [KASS03]

Die Elemente des Quellmodells werden durch Kanten von sog. *Input-Ports* gekennzeichnet, die in der zweiten Zeile von Abb. 3.3 dargestellt sind. Analog dazu werden die Elemente des Zielmodells durch Kanten zu sog. *Output-Ports* festgelegt. Die eigentliche Transformation enthält Konstrukte zum Kopieren einer Klasse vom Quell- zum Zielmodell (*Preservation*), zum Ersetzen einer Klasse des Quellmodells im Zielmodell (*Evolution*) und zum Löschen eines Elementes aus dem Quellmodell (*Removal*).

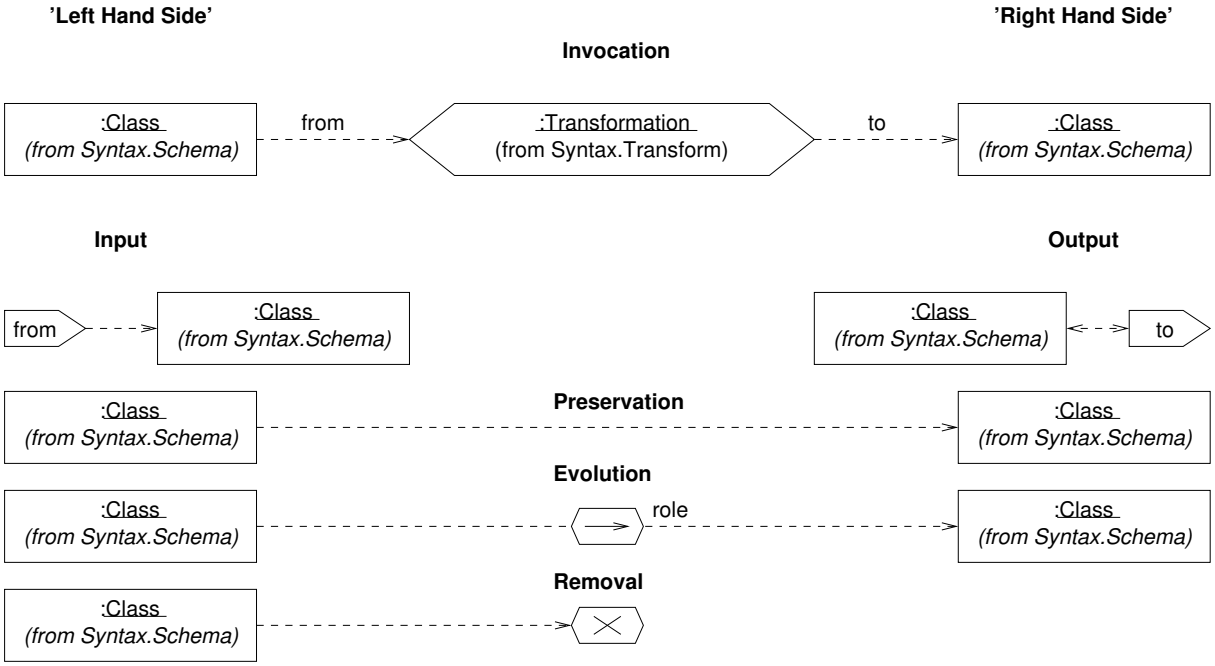


Abbildung 3.3: Elemente der Transformationssprache UMLX [Wil03]

Durch die geringe Anzahl von Konstrukten ist UMLX leicht verständlich. Außerdem wird die Wirkung einer Transformation deutlicher dargestellt als in BOTL oder GReAT. Allerdings er-

fordern selbst einfache Transformationen den Einsatz vieler verschiedener Konstrukte, so dass die Diagramme zur Darstellung einer Transformation schnell sehr groß und unübersichtlich werden. Dieser Nachteil wiegt schwer, weil UMLX im Unterschied zu anderen Transformationssprachen keine textuelle Notation enthält. Die Durchführung einer Transformation erfolgt zur Zeit durch Anwendung von XSLT-Stylesheets auf XMI-Dokumente, die ein Modell repräsentieren. Für spätere Versionen sind Compiler geplant, die UMLX-Transformationen durch Java-Programme durchführen können, um Effizienz und Skalierbarkeit zu verbessern.

Visual Automated Model Transformations (VIATRA) [CHM+02] ist ein Ansatz, der Modelltransformationen einsetzt, um Verifikation und Validierung von UML-Modellen zu ermöglichen. In der Entwurfsphase werden die MOF-Metamodelle der Quell- und Zielsprache entworfen sowie die Transformationsregeln und Kontrollstrukturen durch ein spezielles UML-Profil beschrieben. In der anschließenden Phase wird eine Implementierung der beschriebenen Modelltransformation erzeugt, die zur Durchführung von Modelltransformationen auf der Basis von XMI-Dokumenten verwendet werden kann. Auf dieser Grundlage soll VIATRA in weiteren Schritten ausgebaut werden, um die gewünschten Verifikations- und Validierungsmöglichkeiten zu realisieren. Die grafische Notation von VIATRA ähnelt der oben gezeigten BOTL-Notation.

Ein weiterer graphbasierter Ansatz zur Modelltransformation ist *Visual Model Transformation* (VMT) [SPGB03]. Die VMT-Sprache basiert ebenfalls auf Graphtransformationen und ist eine grafische deklarative Sprache, die Spezifikation und Komposition von Transformationsregeln unterstützt. Die grafischen Elemente der Sprache erlauben Auswahl, Erzeugung, Veränderung und Löschen von Modellelementen. Zusätzliche Bedingungen auf den Quell- und Zielmodellen werden durch die *Object Constraint Language* (OCL) [OCL03] ausgedrückt. Die Notation ähnelt wiederum stark der BOTL-Notation. VMT wurde prototypisch als Erweiterung der Entwicklungsumgebung IBM/Rational XDE implementiert, deren Modelltransformationseigenschaften im folgenden Abschnitt beschrieben werden.

3.2.4 Weitere Ansätze

Neben den Ansätzen, die in den vorangegangenen Abschnitten vorgestellt wurden, werden auch Ansätze vorgeschlagen, die sich keiner der oben angegebenen Kategorien eindeutig zuordnen lassen. Diese Ansätze sind entweder hybrid, d. h. kombinieren verschiedene Mechanismen, oder verwenden eine bestimmte Technologie zur Realisierung der Transformationen. Hybride Ansätze werden beispielsweise von Alcatel u. a. [AST+03], Bezivin u. a. [BDJ+03] und IBM [IBM] vorgeschlagen bzw. angewendet.

Alcatel u. a. [AST+03] definieren die *Transformation Rule Language* (TRL), die deklarative und imperative Sprachelemente miteinander verbindet. Die deklarativen Sprachlemente werden verwendet, um die Verbindungen zwischen Quell- und Zielelementen einer Transformation zu definieren. Die konkreten Transformationsregeln werden dagegen durch die imperativen Sprachelemente beschrieben. Die Regeln enthalten explizite Informationen über die Modellelemente, die während der Ausführung einer Transformation erzeugt, verändert oder gelöscht werden. TRL unterstützt die Organisation der Transformationsregeln durch Module, die zum Gruppieren zusammengehöriger Regeln verwendet werden können. Sowohl auf Regel- als auch

auf Modulebene können Vererbungsbeziehungen definiert werden, so dass eine flexible Organisation der Regeln erlaubt wird.

Die *Atlas Transformation Language* (ATL) [BDJ+03] enthält ebenfalls sowohl deklarative als auch imperative Sprachelemente. Eine ATL-Regel darf beliebige Kombinationen der Sprachelemente enthalten, so dass nicht nur die Sprache hybrid ist, sondern auch die einzelnen Regeln hybrid sein können. Während rein deklarative Regeln ein Muster im Quellmodell auf ein Muster im Zielmodell abbilden, entsprechen rein imperative Regeln den Prozeduren einer imperativen Programmiersprache. Hybride Regeln enthalten zusätzlich zu den Mustern einer deklarativen Regel einen imperativen Anteil, der nach der Anwendung des deklarativen Anteils ausgeführt wird. ATL-Regeln sind unidirektional und können ebenso wie TLR-Regeln durch Vererbungbeziehungen hierarchisch organsiert werden. Die Anwendung der ATL-Regeln erfolgt nach dem Aufruf einer festgelegten Startregel durch Finden passender Muster im Quellmodell und Anwendung der entsprechenden Regeln. Aufgrund der imperativen Anteile in hybriden Regeln lassen sich außerdem auch explizite Regelausführungen definieren.

Die Entwicklungsumgebung [IBM] ist eine Erweiterung des UML-Werkzeuges Rational Rose. Aufgrund der schrittweisen Erweiterung des Werkzeuges werden verschiedene Ansätze zur Durchführung von Modelltransformationen in dem Werkzeug vereint. Ursprünglich enthielt das Werkzeug einen einfachen, musterbasierten Mechanismus, um die Anwendung der Entwurfsmuster aus [GHJV94] zu erleichtern. Dieser Mechanismus wurde anschließend zu einem hybriden Ansatz ausgebaut, der allgemeine Modelltransformationen unterstützt. Die Auswahl der zu transformierenden Elemente aus einem Quellmodell kann entweder durch OCL-Abfragen oder Anweisungen in Java vorgenommen werden.

Weitere Ansätze zur Realisierung von Modelltransformationen sind das *Common Warehouse Metamodel* (CWM) [Obj03a] und MTRANS [PBG01]. CWM definiert ein Rahmenwerk, das einen Mechanismus zur Verbindung von Elementen aus einem Quell- und einem Zielmodell festlegt. Das Rahmenwerk legt allerdings nicht fest, wie vorzugehen ist, um die Elemente des Zielmodells aus den Elementen des Quellmodells abzuleiten. Es könnte prinzipiell jeder der beschriebenen Ansätze verwendet werden, um die in CWM beschriebenen Transformationen zu realisieren.

MTRANS ist eine textuelle Transformationssprache, die zur Transformation beliebiger MOF-konformer Modelle eingesetzt werden kann. Das MTRANS-Rahmenwerk besteht aus drei Elementen, einem MTRANS-Editor sowie zwei *Browsern* für das Quell- und das Ziel-Metamodell. Ein MTRANS Programm wird in ein XSLT-Stylesheet übersetzt, das zur Durchführung einer Transformation auf XMI-Dokumente angewendet wird. Ein Schwachpunkt von XSLT ist die Syntax, die einen hohen Schreibaufwand zur Folge hat. Dieser Schwachpunkt kann durch MTRANS beseitigt werden. Ein weiteres XSLT-Problem, die mangelnde Ausführungsgeschwindigkeit bei komplexen Transformationen, bleibt allerdings erhalten, weil MTRANS keine Modularisierungsmechanismen enthält.

3.2.5 Bewertung

Seit dem Erscheinen der MDA ist eine Vielzahl von Ansätzen vorgestellt worden, die geeignet erscheinen, um die notwendigen Umwandlungen plattformunabhängiger Modelle in plattformspezifische Modelle vorzunehmen. Inwiefern die verschiedenen Vorschläge tatsächlich zur Lösung des Problems beitragen können, ist allerdings noch weitgehend unklar, da die Ansätze nur prototypisch implementiert sind und wenig praxisnahe Tests durchgeführt wurden.

Die Ansätze, die auf direkter Programmierung einer Modelltransformation durch den Anwender basieren, skalieren schlecht bez. des Aufwandes, der zur Realisierung eines Modelltransformators erbracht werden muss. Aus diesem Grund ist die Praxistauglichkeit dieser Kategorie von Ansätzen gering und ihre Anwendbarkeit auf einige wenige Spezialfälle begrenzt.

Deklarative Ansätze sind flexibel und ermöglichen kompakte Beschreibungen von Modelltransformationen. Ein Problem der deklarativen Ansätze ist der deutliche Unterschied zwischen den grafischen Notationen, die im UML-Umfeld in der Regel verwendet werden, und den mathematischen Notationen, die in diesen Ansätzen benutzt werden. Aufgrund der hohen Flexibilität und Präzision sowie der vergleichsweise guten Skalierung, die bei der Beschreibung und Ausführung komplexer Modelltransformationen zu erwarten ist, enthalten die meisten der Vorschläge zur Standardisierung von Modelltransformationen auf der Basis der MOF deklarative Anteile.

Graphtransformationsbasierte Ansätze bieten den besten Übergang von der Beschreibung von Modellen zu der Beschreibung von Modelltransformationen. Viele der vorgestellten Ansätze verwenden die Klassen- bzw. Objektdiagramme der UML und erweitern diese um die Konstrukte, die zur Beschreibung von Regeln benötigt werden. Auf der notationellen Ebene besteht daher eine enge Verwandtschaft zur Modellierung mit der UML, die den Übergang erleichtert. Allerdings führt die grafische Notation häufig zu Schwierigkeiten bei der Definition komplexer Transformationen, da häufig sehr komplexe Regeln oder sehr viele Regeln definiert werden müssen, und die Übersichtlichkeit entsprechend eingeschränkt wird. Ansätze, die neben der grafischen auch eine textuelle Notation der Transformationsregeln erlauben, sind daher vorzuziehen, da bei komplexen/vielen Regeln eine kompaktere und damit übersichtlichere Darstellung erreicht werden kann. Weitere Schwierigkeiten können durch die nicht-deterministische Anwendung der Regeln entstehen, die eine sorgsame Definition einer Menge von Regeln voraussetzt, um die gewünschten Resultate zu erhalten [CH03].

Die vorgestellten Ansätze zur Durchführung von Modelltransformationen nutzen die Möglichkeiten der vorhandenen Technologien gut aus. Die Umsetzung der Ansätze in praxistaugliche Werkzeuge ist der nächste Schritt, der ausgeführt werden muss, um eine rein modellbasierte Softwareentwicklung zu ermöglichen. Aufgrund der Vielzahl der verfügbaren Ansätze und der verschiedenen Stärken und Schwächen der einzelnen Ansätze vermuten [CH03], dass eine Kombination der Konzepte verschiedener Ansätze benötigt wird, um reale Problemstellungen lösen zu können. Die noch zu lösende Aufgabe besteht demnach weniger in der Entwicklung eines weiteren Ansatzes zur Modelltransformation, sondern eher in der Durchführung praxisrelevanter Fallstudien.

3.3 Codegenerierung

Nahezu jedes Werkzeug, das die Modellierung von Softwaresystemen unterstützt, bietet mittlerweile auch die Möglichkeit, aus den Modellen Code einer Programmiersprache zu erzeugen. Dies gilt insbesondere für weitverbreitete UML-Werkzeuge wie Together und IBM/Rational Rose (XDE), die mehr und mehr von reinen UML-Werkzeugen zu vollständigen Entwicklungsumgebungen ausgebaut werden. Allerdings ist die Codegenerierung dieser Werkzeuge nicht sehr flexibel, weil die Eingabesprache und auch große Teile des erzeugten Codes fest im Werkzeug implementiert sind. Um deutliche Veränderungen an der Codegenerierung vornehmen zu können, müssen Erweiterungen programmiert werden. Daraus resultiert hoher Aufwand bei der Anpassung solcher Werkzeuge an einen bestimmten Anwendungsfall.

Neben dem Ausbau von UML-Werkzeugen sind auch verschiedene Ansätze entwickelt worden, deren Fokus auf der automatischen Erzeugung von (Teil-)systemen aus Modellen liegt. [Voe03] klassifiziert diese Ansätze anhand von Mustern und bewertet die Eigenschaften der verschiedenen Kategorien. Er unterscheidet im Wesentlichen schablonenbasierte Ansätze und Ansätze, die ein Rahmenwerk zur Programmierung zur Verfügung stellen. Sehr ähnliche Unterscheidungen verwenden auch [CH03] und [SK03].

3.3.1 Direkte Programmierung

Direkte Programmierung auf Basis einer Metamodellimplementierung wurde bereits in Abschnitt 3.2.1 als Möglichkeit zum Implementieren von Modelltransformationen vorgestellt. Aufgrund der hohen Flexibilität dieses Ansatzes kann derselbe Ansatz auch zur Realisierung von Codegeneratoren angewandt werden. Allerdings bleibt auch der wesentliche Nachteil erhalten: Codegeneratoren arbeiten in der Regel auf einer niedrigeren Abstraktionsebene als Modelltransformatoren, so dass der wesentliche Nachteil hier sogar noch schwerer wiegt als bei der Entwicklung von Modelltransformatoren. Die direkte Programmierung von Codegeneratoren ist somit nur dann sinnvoll, wenn der erzeugte Code besondere Anforderungen zu erfüllen hat, die den hohen Entwicklungsaufwand rechtfertigen.

3.3.2 Schablonenbasierte Ansätze

In Abb. 3.4 sind die Elemente der schablonenbasierten Codegenerierung schematisch dargestellt. Die Schablonen enthalten sowohl Fragmente des zu erzeugenden Codes als auch Metaattribute, die Informationen aus dem zu übersetzenden Modell repräsentieren. Durch Anwendung der Schablonen auf ein Modell werden die Codeanteile mit den ausgelesenen Informationen kombiniert, so dass Programme oder Teile von Programmen erzeugt werden, die das Modell repräsentieren. Die Programme können anschließend in ausführbaren Code übersetzt werden. In diesem Abschnitt werden die Eigenschaften einiger aktueller Codegeneratoren bzw. Codegenerator-Rahmenwerke vorgestellt.

AndroMDA [And] ist ein Rahmenwerk zur Implementierung von Codegeneratoren. Ein mit Hilfe von AndroMDA realisierter Codegenerator verarbeitet eine XMI-Repräsentation eines UML-

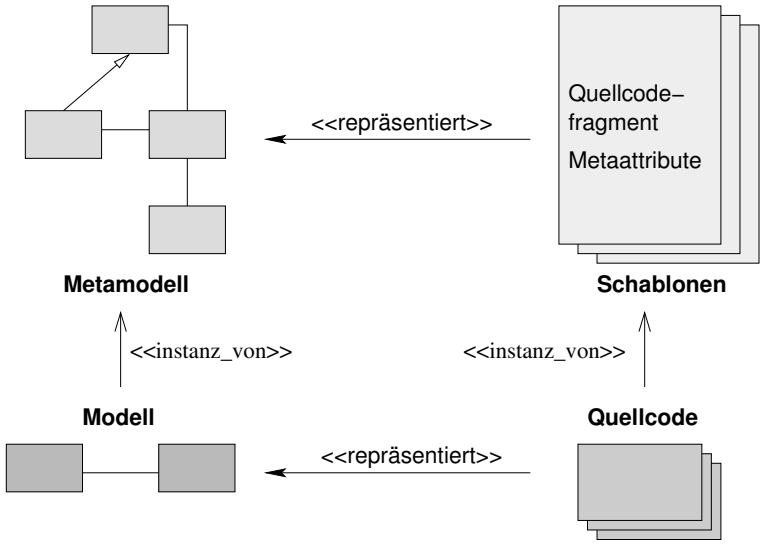


Abbildung 3.4: Ablauf einer schablonenbasierten Codegenerierung

Modells und erzeugt Komponenten eines Anwendungssystems. In der Regel werden die generierten Komponenten durch *Enterprise Java Beans* (EJB) [DeM03] implementiert. Die vordefinierten Schablonen erzeugen EJB´s, die in einem Applikationsserver lauffähig sind. Ähnlich stark auf die Erzeugung von EJB´s ausgerichtet sind FUUT-je [vEB] und OptimaJ [OpJ].

Technisch basiert AndroMDA auf *XDoclet* [XDT, WR03] und *Velocity* [Vel]. XDoclet ist ein Werkzeug, das attributorientierte Programmierung in Java ermöglicht und daher auch zur Codegenerierung verwendet werden kann. Zusätzliche Attribute, die den Übersetzungsprozess beeinflussen können, werden durch javadoc-Kommentare [JDC] ausgedrückt. Velocity stellt eine Schablonenablaufumgebung zur Verfügung, die eine einfache imperative Sprache implementiert. Velocity ist eng mit Java integriert, so dass innerhalb einer Schablone Java-Code verwendet werden kann, um auf Elemente einer Java-Datenstruktur zuzugreifen.

Prinzipiell ermöglicht AndroMDA die Anpassung des generierten Codes an einen bestimmten Anwendungsfall. Allerdings muss das Eingabemodell in UML beschrieben werden, und die Codeerzeugung ist deutlich auf Java, insbesondere EJB´s, ausgerichtet. Anpassung und Erweiterung sind vergleichsweise aufwändig, da nur wenige Informationen direkt an die Schablonen übergeben werden und auf alle weiteren Informationen in den Schablonen durch Java-Codesequenzen zugegriffen werden muss.

ArchitectureWare [Voe] ist aus dem *b+m Generator Framework* hervorgegangen und stellt ein Rahmenwerk für die Implementierung für Codegeneratoren zur Verfügung. Wie AndroMDA ist auch ArchitectureWare in erster Linie für die Codegenerierung aus UML-Modellen entwickelt worden. In der ausgelieferten Konfiguration unterstützt ArchitectureWare nur die Codegenerierung aus UML-Klassendiagrammen. Anpassungen und Erweiterungen des erzeugten Codes lassen sich durch Schablonen spezifizieren, die in einer ArchitectureWare-eigenen Sprache ausgedrückt werden.

Das Rahmenwerk basiert auf *nsuml* [NSU], einer Implementierung des UML-Metamodells. Erweiterungen der Eingabesprache erfordern eine Erweiterung der Implementierung des Metamodells. Die Erweiterung der Metamodellimplementierung muss manuell vorgenommen werden und ist entsprechend aufwändig. Möglicherweise lässt sich auch der nsuml-Generator zu diesem Zweck verwenden oder eine Umstellung des Rahmenwerkes auf eine andere Metamodellimplementierung vornehmen. Allerdings sieht der Ansatz dies nicht vor, so dass auch diese Möglichkeiten einen hohen Aufwand erfordern.

Die Codeerzeugung wird von ArchitectureWare in zwei Schritten vorgenommen. Zunächst wird das UML-Modell in eine Instanz eines Java-Metamodells übertragen, das anschließend durch Anwendung der Schablonen in Code übertragen wird. Durch diese Vorgehensweise ist auch die Erzeugung von Code einer anderen Programmiersprache aufwändig, weil zunächst ein Metamodell für die Zielsprache erstellt und implementiert werden muss.

ArcStyler [Obj] unterstützt die Erzeugung von Software-Komponenten, die entweder EJB´s oder .NET-Komponenten [NET] sein können. Noch stärker ausgeprägt ist die Flexibilität des erzeugbaren Codes bei *Codagen Architect* [CTC], einem Codegenerator, der neben EJB´s und .NET auch fertige Schablonen zur Erzeugung von Code in den Programmiersprachen Java, C#, C++ und Basic unterstützt. Anders als AndroMDA und ArchitectureWare verarbeiten ArcStyler und Codagen nicht nur die statischen sondern auch die dynamischen Anteile der UML. Der Schwerpunkt bei ArcStyler und Codagen Architect liegt weniger auf der Anpassbarkeit des erzeugten Codes durch den Anwender als auf der Unterstützung mehrerer Zielsprachen und -technologien durch vordefinierte Schablonen.

3.3.3 Bewertung

In Tab. 3.1 werden die Eigenschaften der Ansätze zur Codegenerierung aus Modellen zusammengefasst. Dabei werden die Ansätze, die auf direkter Programmierung basieren, nicht aufgeführt, da deren Eigenschaften im Wesentlichen vom direkt programmierten Anteil abhängig sind und daher nicht bewertet werden können. Im Einzelnen enthält die Tabelle Angaben über den verwendeten Mechanismus zur Realisierung der Codeerzeugung, die Eingabesprache, die Zielsprache oder -technologie, und die Anpassbarkeit. Die Anpassbarkeit wird in Anpassbarkeit der Eingabesprache und Anpassbarkeit des zu generierenden Codes unterschieden.

	Mechanismus	Quelle	Ziel	Anpassbarkeit[2]
AndroMDA	Velocity/XDoclet	UML (statisch)	EJB	nein/ja
ArchitectureWare	Schablone	UML (statisch)	EJB	nein/eingeschränkt
ArcStyler	Schablone	UML	EJB, .NET	nein/ja
Codagen Architect	Schablone	UML	diverse	nein/ja

Tabelle 3.1: Eigenschaften von Ansätzen zur Codegenerierung aus Modellen

[2] der Eingabesprache / des generierten Codes

Tab. 3.1 zeigt, dass die bestehenden Ansätze zur Codegenerierung eine Eingabesprache auf eine oder mehrere Implementierungstechnologien abbilden können. Der Fokus liegt dabei auf der Bereitstellung einer fertigen Lösung, die der Anwender zur Erzeugung von Implementierungen für seine Modelle verwenden kann. Einige Generatoren, insbesondere Codagen, enthalten Schablonen für verschiedene Zielsprachen bzw. -technologien, so dass ein weites Spektrum möglicher Anwendungen abgedeckt wird.

Anpassungen der Schablonen sind in allen Werkzeugen möglich. Allerdings verwenden die meisten Werkzeuge eigene Sprachen und -mechanismen zur Beschreibung von Schablonen, so dass der Anwender zunächst diese Sprachen und -mechanismen erlernen muss. Mitunter werden in den Schablonen direkte Bezüge auf eine Metamodellimplementierung genommen, so dass der Anwender zusätzlich Kenntnisse des Aufbaus der entsprechenden Implementierung benötigt. Zudem enthalten Schablonen einiger Ansätze sowohl Fragmente des zu generierenden Codes als auch Informationen über das zu generierende Artefakt selbst, z. B. den Dateinamen. Diese Mischung erschwert das Verständnis einer Schablone.

Die Eingabesprache ist bei allen Ansätzen die UML bzw. eine Teilmenge der UML, die nur die statischen Anteile umfasst. Veränderungen der Eingabesprache sind nur möglich, indem Stereotypen verwendet werden, um eine bestimmte Verwendung einzelner UML-Elemente zu kennzeichnen. Die Stereotypen können in den Schablonen ausgewertet werden, so dass domänenspezifische Elemente modelliert und zur Codeerzeugung verwendet werden können. Dies gilt allerdings nur, solange die domänenspezifischen Elemente durch Modellelemente der UML „simuliert“ werden können.

Keiner der vorgestellten Ansätze kann ohne manuelle Programmierung an eine Modellierungssprache angepasst werden, die deutlich von der UML abweicht. Die Implementierung eines Codegenerators, der einem Modelltransformator nachgeschaltet wird, erfordert aber genau diese Anpassung. Es werden daher Werkzeuge und Komponenten benötigt, die die Entwicklung von Codegeneratoren für beliebige Modellierungssprachen erleichtern. Da viele der in Abschnitt 3.2 vorgestellten Modelltransformatoren auf der Basis von MOF-Modellen arbeiten, sollten Codegeneratoren unterstützt werden, deren Syntax durch ein MOF-Modell definiert ist. Ein entsprechender Ansatz wird in den folgenden Kapiteln vorgestellt.

3.4 Zusammenfassung

In diesem Kapitel wurden die Begriffe *Modelltransformation* und *Codegenerierung* definiert. Modelltransformationen verwenden die Abbildung eines Metamodells A auf ein Metamodell B, um Instanzen von A in Instanzen von B zu überführen, während Codegenerierungen Metamodelle auf die Artefakte einer Programmiersprache oder Implementierungstechnologie abbilden. Modelltransformation und Codegenerierung sind wichtige Voraussetzungen, um die Entwicklung von Softwaresystemen (teil-) automatisieren zu können.

Im Anschluss an die Begriffsdefinitionen wurden aktuelle Ansätze zur Realisierung von Modelltransformationen bzw. Codegenerierungen vorgestellt. Die meisten Ansätze zur Modelltransformation sind sehr flexibel bezüglich der zu verarbeitenden Modelle und können für ein weites

Spektrum von Aufgaben eingesetzt werden. Die existierenden Ansätze zur Codegenerierung verarbeiten dagegen nur Modelle, die durch die UML beschrieben werden. Einfache Anpassungen durch die in UML enthaltenen Mechanismen (Stereotypen) sind zwar möglich, weitergehende Anpassungen der Eingabesprache sind jedoch nur durch direktes Programmieren der Bestandteile eines Generators durch den Anwender möglich.

Veränderungen an dem zu erzeugenden Code für ein UML-Modell können dagegen bei den schablonenbasierten Ansätzen relativ leicht vorgenommen werden. Allerdings basieren die meisten existierenden Werkzeuge auf proprietären Techniken zur Beschreibung und Ausführung der Schablonen. Dies führt zu erhöhtem Einarbeitungsaufwand, weil neben den Elementen der zu übersetzenden Modellierungssprache auch die Elemente der Beschreibungssprache erlernt werden müssen. Durch den Einsatz standardisierter Technologien lässt sich dies vermeiden.

Die bisherigen Erfahrungen mit der Anwendung der UML in den verschiedenen Anwendungsdomänen haben bereits gezeigt, dass angepasste Sprachen benötigt werden. Zum einen sind in der UML zu viele Elemente enthalten, die in einzelnen Anwendungsdomänen nicht benötigt werden. Zum anderen fehlen aber auch Elemente, die die Modellierung bestimmter Systeme vereinfachen können. Auch die Kopplung der Modelltransformationsansätze wird stark eingeschränkt, wenn nur auf Basis des UML-Metamodells Code generiert werden kann.

Um dieses Szenario zu realisieren, wird eine Infrastruktur benötigt, die die Implementierung von Codegeneratoren für alle Sprachen unterstützt, die durch ein Metamodell spezifiziert werden können. In der weiteren Arbeit wird ein Baukasten vorgestellt, der die Realisierung von Codegeneratoren für alle Modellierungssprachen erleichert, die durch ein MOF-Metamodell definiert sind. Da sich durch MOF auch die Elemente von Programmiersprachen wie Java beschreiben lassen, wie Dedic und Matula zeigen [DM02], erweitert eine solche Infrastruktur das Einsatzgebiet von Codegeneratoren gegenüber den existierenden Ansätzen deutlich.

Kapitel 4

Der MOmo-Baukasten

Die modellbasierte Softwareentwicklung benötigt in allen Entwicklungsphasen die Unterstützung durch geeignete Werkzeuge, die den Umgang mit den Modellen erleichtern. Die derzeit am weitesten verbreitete Modellierungssprache ist die UML, deren Syntax durch ein Metamodell festgelegt ist. Aus diesem Grund basieren viele Werkzeuge auf einer Implementierung des UML-Metamodells. Erweiterungen und Anpassungen der UML können entweder durch Auszeichnen von Modellierungselementen mit Stereotypen oder durch eine Erweiterung bzw. Anpassung des UML-Metamodells erfolgen. Während Stereotypen hauptsächlich für kleinere Anpassungen vorgesehen sind, können durch Veränderungen des Metamodells beliebig tiefgreifende Änderungen vorgenommen werden. Angepasste bzw. neu erstellte Metamodelle können daher als Grundlage der Entwicklung domänenspezifischer Modellierungssprachen und -werkzeuge auf Basis der UML angesehen werden.

Der Literaturüberblick in Kapitel 3 zeigt, dass die meisten Ansätze zur Modelltransformation auf Metamodellebene arbeiten, d. h. Abbildungen zwischen den Elementen eines Metamodells und Elementen desselben oder eines anderen Metamodells beschreiben. Mit Hilfe dieser Ansätze lassen sich auch Modelle auf Implementierungstechnologien abbilden; vorausgesetzt, es gibt ein Metamodell, das die Implementierungstechnologie beschreibt. Ein Beispiel für ein solches Metamodell ist das Java-Metamodell von Dedic und Matula [DM02].

Die bestehenden Ansätze zur Erzeugung von Quelltext einer Programmiersprache für die Elemente eines Metamodells sind allerdings nicht flexibel genug, um Abbildungen beliebiger Metamodelle auf Quelltexte zu beschreiben. Stattdessen definieren sie in der Regel Abbildungen der Elemente des UML-Metamodells auf die Artefakte einer Implementierungstechnologie. Eine Abbildung eines Metamodells, das eine Programmiersprache beschreibt, auf Quelltext derselben oder einer anderen Programmiersprache erfordert dagegen eine manuell erstellte Implementierung eines entsprechenden Codegenerators. Der Entwicklungsaufwand, der dafür notwendig ist, verhindert den Einsatz von Codegeneratoren und damit auch die Erhöhung des Abstraktionsgrades bei der Softwareentwicklung. Das Hauptziel des MOmo-Baukastens ist daher,

den Aufwand zu reduzieren, der zur Implementierung von Codegeneratoren erbracht werden muss.

In Abschnitt 4.1 werden die grundlegenden Anwendungsfälle beschrieben, die der Entwicklung der Komponenten des MOmo-Baukastens zugrunde liegen. Daraus werden die Anforderungen abgeleitet, die die MOmo-Komponenten erfüllen sollen. In Abschnitt 4.2 wird die Umsetzung der Anforderungen in ein Konzept zur Implementierung von Codegeneratoren beschrieben. Abschließend wird in Abschnitt 4.3 ein Beispiel für die Anwendung eines Codegenerators angegeben, um das Prinzip der Codeerzeugung zu verdeutlichen.

4.1 Anforderungsdefinition

Software-Baukästen werden entwickelt, um die Entwicklung von Anwendungen zu vereinfachen. Die Anforderungen an einen Software-Baukasten hängen daher maßgeblich von der Anwendungsdomäne ab, die unterstützt werden soll. Der MOmo-Baukasten soll die Entwicklung von Werkzeugen, insbesondere Codegeneratoren, für MOF-basierte Modellierungssprachen vereinfachen. Im Folgenden werden zunächst die zwei Anwendungsfallkategorien des MOmo-Baukastens beschrieben und anschließend die Anforderungen definiert, die aus den Anwendungsfällen abgeleitet wurden.

4.1.1 Anwendungsfallkategorien

Die Anwendungsfälle für den MOmo-Baukasten können in zwei Kategorien unterschieden werden. In beiden Kategorien wird der im Baukasten enthaltene Codegenerator eingesetzt, um eine Implementierung eines MOF-Modells zu erzeugen. Die generierte Implementierung wird anschließend mit anderen Komponenten zu einer Anwendung verbunden. Im allgemeinen Fall kann es sich um eine beliebige Anwendung handeln, so dass keine weiteren Komponenten des MOmo-Baukastens benutzt werden. Eine spezielle Anwendungsfallkategorie ist die Entwicklung eines Codegenerators für eine Modellierungssprache, deren Sprachumfang durch ein MOF-Modell festgelegt ist. In diesem Anwendungsfall können zusätzliche Komponenten des Baukastens verwendet werden, um den Implementierungsaufwand weiter zu reduzieren. Im Weiteren werden die beiden genannten Kategorien im Detail beschrieben.

4.1.1.1 Implementierung eines Metamodells

Die UML ist eine weit verbreitete grafische Modellierungssprache, die in einer Vielzahl von Anwendungsdomänen verwendet werden kann. Zusätzlich enthält die UML Mechanismen, um die Sprache an bestimmte Anwendungsdomänen anzupassen. Eine Anpassung kann sowohl die Auswahl einer Teilmenge der Elemente der „Standard“-UML als auch die Definition zusätzlicher Elemente bedeuten. Das UML 2.0 Metamodell enthält verschiedene Ebenen, die einzelne Aspekte der Modellierungselemente festlegen. Aufgrund dieser Architektur lassen sich vergleichsweise leicht Anpassungen der UML durch Metamodelle definieren. Entspre-

chende Metamodelle erben die für die Anwendungsdomäne geeigneten Eigenschaften aus dem UML-Metamodell und enthalten weitere Metaobjekte, die zusätzliche benötigte Mechanismen und Eigenschaften festlegen. Aufgrund dieser Architektur ist die UML-Version 2.0 besser geeignet als ihre Vorgänger, um domänenspezifische Sprachen auf ihrer Basis zu definieren.

Wenn eine Anwendungsdomäne durch eine Anpassung der UML unterstützt wird, sollten auch die verwendeten Werkzeuge diese Anpassung implementieren. Eine Aufgabe des MOmo-Baukastens ist daher, die Implementierung von Werkzeugen für Anpassungen der UML zu unterstützen. Ein Werkzeug, das die modellbasierte Entwicklung unterstützen soll, muss Modelle laden, repräsentieren und speichern können. Wird die Modellierungssprache durch ein Metamodell definiert, bestehen die Modelle aus Instanzen der Elemente des Metamodells. Eine Implementierung einer solchen Modellierungssprache erfordert demzufolge eine Implementierung der Klassen des Metamodells.

Die Implementierung eines umfangreichen Metamodells wie etwa des UML-Metamodells erfordert beträchtlichen Aufwand, um alle im Metamodell enthaltenen Klassen manuell zu implementieren. Aus diesem Grund verwenden einige UML-Werkzeuge vereinfachte Darstellungen des UML-Metamodells und/oder implementieren nur Teile der Sprache. Dies führt nahezu zwangsläufig zu einem von der Spezifikation abweichenden Verhalten des Werkzeugs bzw. zu unvollständigen Implementierungen. Eine Möglichkeit, den Aufwand für die Implementierung des Metamodells zu reduzieren, ist die automatische Erzeugung der Implementierung. Um eine Implementierung für die UML oder Anpassungen der UML erzeugen zu können, wird ein Werkzeug benötigt, das Implementierungen für MOF-Modelle generieren kann. Ein solches Werkzeug bildet den Kern des MOmo-Baukastens. Der erste Anwendungsfall für den

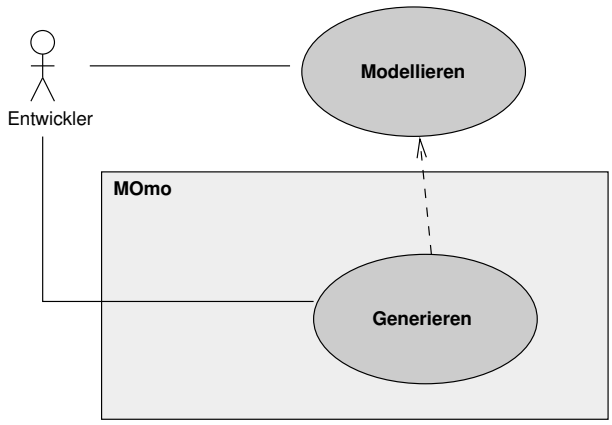


Abbildung 4.1: Implementierung eines Metamodells

MOmo-Baukasten ist also die Erzeugung von Metamodellimplementierungen als Grundlage für den Bau von Werkzeugen für Modellierungssprachen, die durch ein MOF-Modell definiert werden. In Abb. 4.1 ist dargestellt, wie die Entwicklung eines Werkzeuges unter Verwendung eines MOmo-Generators abläuft. Die Grundlage der Entwicklung ist ein Metamodell, das den Sprachumfang der zu unterstützenden Sprache festlegt. Dies ist durch den Anwendungsfall **Modellieren** dargestellt. Die Modellierung soll nicht durch den MOmo-Baukasten unterstützt wer-

den, sondern durch ein beliebiges MOF- oder UML-fähiges Werkzeug erfolgen. Das entstandene Metamodell wird durch den MOmo-MOF-Codegenerator in eine Implementierung übersetzt, die mit dem Rest der Anwendung verbunden wird. Der Anwendungsfall **Generieren** repräsentiert diesen Arbeitsschritt. Da die Codegenerierung das Ergebnis der Modellierung als Eingabe verwendet, ist der Anwendungsfall **Generieren** abhängig vom Anwendungsfall **Modellieren**, was durch den gestrichelten Pfeil dargestellt ist. Ein Beispiel für ein Werkzeug, das auf diese Art entwickelt wird, ist das UML-Modellierungswerkzeug ArgoUML [Arg] (siehe Abb. 4.2)

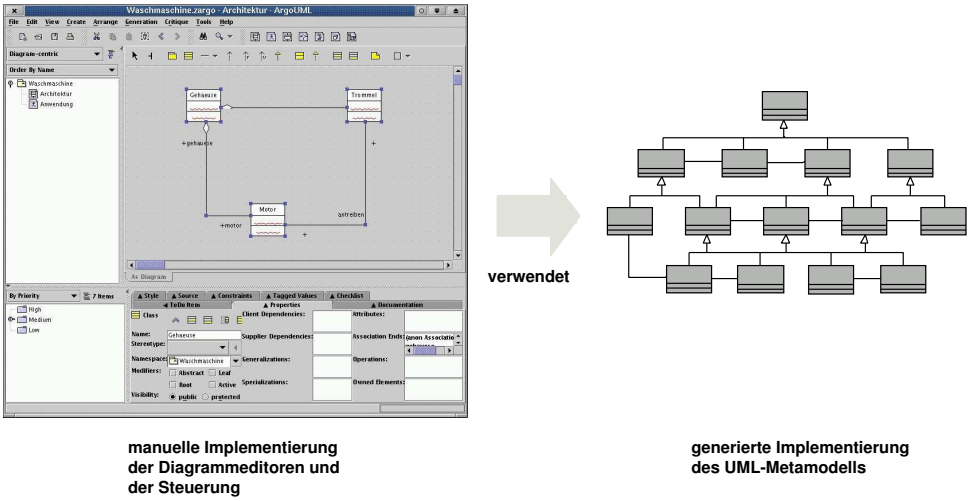


Abbildung 4.2: Aufbau des UML-Werkzeuges ArgoUML

ArgoUML besitzt eine *Model-View-Controller*-Architektur (siehe z. B. [GHJV94]). Sowohl die Steuerung als auch die Sichten auf das Modell werden manuell implementiert und mit einer generierten Implementierung des UML-Metamodells kombiniert. Diese Form der Implementierung hat eine enge Verflechtung der manuell implementierten Anteile mit den generierten Anteilen zur Folge, so dass bei Veränderungen an den generierten Schnittstellen auch Veränderungen an manuell erstellten Anteilen vorgenommen werden müssen. Beispielsweise wurden die Schnittstellen der Bibliothek *nsuml* [NSU], die eine Implementierung des UML-Metamodells bereitstellt, bei der Umstellung von der UML-Version 1.3 zur UML-Version 1.4 deutlich verändert. Der daraus resultierende hohe Anpassungsaufwand hatte zur Folge, dass ArgoUML nach wie vor nur die UML-Version 1.3 implementiert. Um solche Kompatibilitätsverluste zu vermeiden, muss die generierte Schnittstelle vom Anwender anpassbar sein. Nach Möglichkeit sollte der generierte Code auch standardisierte Schnittstellen enthalten, um das Verwendungsrisiko weiter zu verringern.

4.1.1.2 Implementierung von Codegeneratoren

Die Entwicklung von Codegeneratoren für MOF-basierte Modellierungssprachen ist ein Anwendungsfall, der die in Abschnitt 4.1.1.1 beschriebene Implementierung eines Metamodells als Teilaufgabe enthält. Neben der Metamodellimplementierung werden weitere Bestandteile

benötigt, um eine Abbildung der Elemente des Metamodells auf Elemente einer Implementierungstechnologie festzulegen.

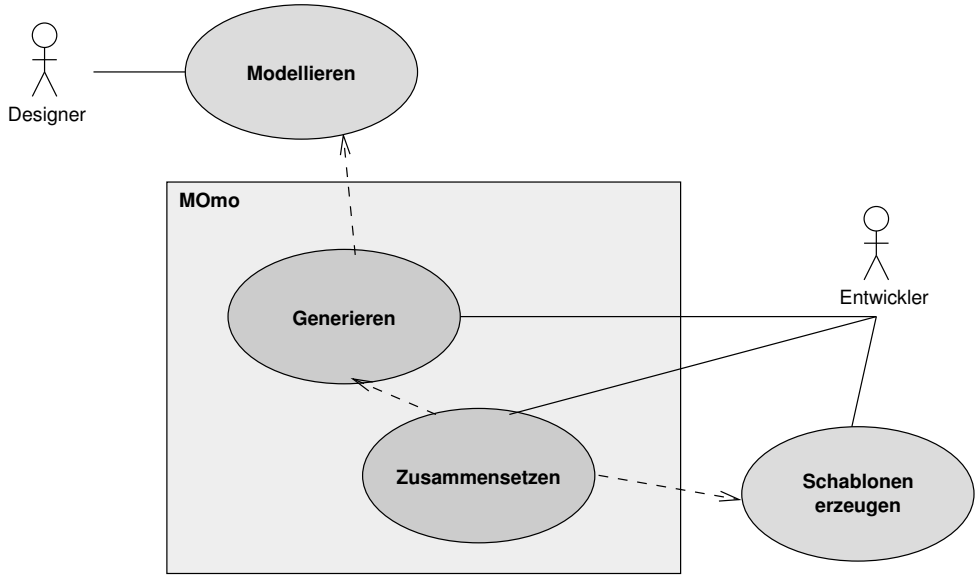


Abbildung 4.3: Implementierung eines Codegenerators

Abb. 4.3 zeigt, welche Teile des Entwicklungsprozesses eines Codegenerators durch den MOmo-Baukasten unterstützt werden. Da die Entwicklung eines Codegenerators einen Spezialfall des Werkzeugbaus darstellt, enthält der Anwendungsfall den oben beschriebenen Anwendungsfall „Implementierung eines Metamodells“ aus Abb. 4.1.

Allerdings werden Modellierungssprachen in der Regel von Spezialisten entworfen, so dass hier zwischen *Designer* und *Entwickler* unterschieden wird. Der Designer entwirft die Sprache und hat dabei keinerlei Berührungspunkte mit dem MOmo-Baukasten. Das Ergebnis ist das Metamodell der Sprache, die zur Beschreibung der Modelle verwendet werden muss, die durch den zu entwickelnden Codegenerator übersetzt werden können. Der Entwickler entwickelt den Codegenerator für die Modellierungssprache und verwendet dazu Komponenten und Werkzeuge, die der MOmo-Baukasten zur Verfügung stellt. Mit Hilfe des MOmo-MOF-Codegenerators wird eine Implementierung des Metamodells erzeugt. Dies ist durch den Anwendungsfall **Generieren** dargestellt. Die Abhängigkeit des Anwendungsfalles **Generieren** vom Anwendungsfall **Modellieren** wird durch den gestrichelten Pfeil zwischen den beiden Anwendungsfällen repräsentiert. Weiterhin definiert der Entwickler die Schablonen, die die Abbildungen der Elemente der Modellierungssprache auf Elemente der Zielsprache festlegen. Abschließend kombiniert der Entwickler die generierten Komponenten mit den generischen Komponenten, die der MOmo-Baukasten bereitstellt, sowie den Schablonen, um den Codegenerator zusammenzusetzen. Dies wird durch den Anwendungsfall **Zusammensetzen** dargestellt, dessen Abhängigkeiten von den Anwendungsfällen **Generieren** und **Schablonen erzeugen** wiederum durch gestrichelte Pfeile dargestellt sind.

4.1.2 Anforderungen

Der MOmo-Baukasten soll die Entwicklung von Werkzeugen für MOF-basierte Modellierungssprachen, insbesondere Codegeneratoren, vereinfachen. Um dieser Aufgabe gerecht zu werden, müssen verschiedene Anforderungen erfüllt werden. Im Folgenden werden die Anforderungen an den MOmo-Baukasten aus den in Abschnitt 4.1.1 beschriebenen Anwendungsfällen abgeleitet.

Eingabeformat Die Entwickung eines Werkzeuges für eine MOF-basierte Modellierungssprache beginnt mit der Definition der zu unterstützenden Modellierungssprache. Aufgrund der Angleichung der Metamodelle von MOF und UML in den Versionen 2.0 beider Standards können UML-Werkzeuge auch zur Definition von MOF-Modellen verwendet werden. Dadurch steht eine Vielzahl von Modellierungswerkzeugen zur Verfügung, um die Eingabe für die Werkzeuge des MOmo-Baukastens zu erzeugen. Dies kann entweder durch Unterstützung der werkzeugspezifischen Speicherformate oder durch Verwendung eines allgemein anerkannten Standards ermöglicht werden.

Für die UML und alle anderen Modellierungssprachen, die durch ein MOF-Modell definiert werden, definiert der XMI-Standard (siehe Kapitel 7) eine Repräsentation von Modellen durch XML-Dokumente. Die meisten am Markt befindlichen Werkzeuge enthalten Schnittstellen, die das Laden und Speichern von XMI-Dokumenten ermöglichen. Aus diesem Grund soll der MOmo-Baukasten XMI-Dokumente verarbeiten können. Da die meisten erhältlichen Werkzeuge zur Zeit noch die 1.x Versionen der MOF/UML/XMI-Standards implementieren, müssen MOF/UML 1.x Modelle importiert werden können.

Änderbarkeit/Erweiterbarkeit des generierten Codes Wenn generierter Code als Basis einer Anwendung eingesetzt wird, entsteht eine hohe Abhängigkeit von den generierten Schnittstellen. Werden diese in einer späteren Version des Generators verändert, sind umfangreiche Anpassungen der Anwendung erforderlich. Dies verhindert mitunter die Anpassung eines Werkzeuges an neue Versionen eines Standards.

Ein Beispiel dafür ist das frei verfügbare Werkzeug ArgoUML [Arg], das nach wie vor die Version 1.3 des UML-Standards implementiert, weil die späteren Versionen vom Generator der verwendeten Metamodellimplementierung nicht mehr unterstützt werden. Der Grund dafür ist die zwischenzeitliche Einführung des *Java Metadata Interface*-Standards (JMI) [Dir02] für die Abbildung von MOF-Metamodellen auf Java und die Umstellung neuerer Versionen der von ArgoUML verwendeten Metamodellimplementierung auf diesen Standard (vgl. Abschnitt 4.1.1.1).

In vielen Fällen erleichtern kleine Änderungen am generierten Code die Entwicklung einer Anwendung. Es ist daher wichtig, dass solche Änderungen leicht durchgeführt werden können. Ein MOmo-Generator muss Änderungen des generierten Codes ohne tiefgreifende Eingriffe in den Generator selbst erlauben. Die Änderung der Generierungsvorschrift für ein Artefakt soll ohne Kenntnisse über den MOmo-Baukasten durchgeführt werden können. Außerdem sollen die Generierungsvorschriften parameterisierbar sein können, um ihren Anwendungsbereich zu

vergrößern. Parameterisierungen können z. B. die Namen der generierten Artefakte beeinflussen.

Übersetzungsgeschwindigkeit Obwohl der Durchsatz bei Codegeneratoren eine gegenüber anderen Anwendungssystemen untergeordnete Rolle spielt, muss der Codegenerator zumindest leistungsfähig genug sein, um im Entwicklungsprozess keinen maßgeblichen Einfluss auf die Entwicklungszeit zu haben. Dies gilt insbesondere, weil die generierten Artefakte meist anschließend durch vorhandene Technologien wie Übersetzer für Programmiersprachen weiter transformiert werden müssen.

Die Anforderung einer vernachlässigbaren Zeitdauer für die Durchführung einer Modelltransformation soll für alle mit dem MOmo-Baukasten erstellten Codegeneratoren gelten. Als vernachlässigbar wird eine Zeitdauer angenommen, die nicht wesentlich über der Zeitdauer liegt, die zum Übersetzen der generierten Quelltexte in Maschinencode liegt. Die Generatoren können sich zum einen bez. der unterstützten Eingabesprache und zum anderen bez. der Komplexität der zu generierenden Artefakte unterscheiden. Der verwendete Ansatz für den MOmo-Baukasten muss demzufolge sowohl mit dem Umfang der Modellierungssprache als auch der Komplexität der zu erzeugenden Artefakte möglichst gut skalieren.

Flexibilität MOmo-Generatoren sollen flexibel verwendbar sein. Dies bedeutet insbesondere, dass die zu generierenden Artefakte auf einer beliebigen Implementierungssprache oder -technologie basieren dürfen. Daher darf der MOmo-Baukasten keinerlei Restriktionen bez. des Aufbaus der zu generierenden Artefakte voraussetzen. Zudem müssen im Zusammenhang mit der oben genannten Forderung nach leichter Veränderbarkeit des Codes alle Informationen über das Modell in jedem Stadium des Generierungsprozesses verfügbar sein.

Konfigurierbarkeit Verschiedene Implementierungssprachen und -technologien stellen verschiedene Anforderungen an Codegeneratoren. Beispielsweise sollen die Namen von Paketen der Programmiersprache Java [GJSB00] häufig mit den Pfaden der erzeugten Dateien korrespondieren. Abhängig von der Struktur des zu erzeugenden Codes muss eine beliebige Anzahl von Artefakten für ein Modellelement oder auch ein Artefakt für mehrere Modellelemente erzeugt werden können. Aus diesen Gründen müssen MOmo-Generatoren durch eine Konfiguration an verschiedene Anwendungsszenarien angepasst werden können. Die Konfiguration muss unter anderem folgende Eigenschaften festlegen:

- Welche Artefakte werden für ein Modellelement erzeugt?
- Welche Artefakte sollen im aktuellen Durchlauf erzeugt werden?
- Welches Basisverzeichnis soll für generierte Artefakte verwendet werden?
- Welchen Namen erhalten die zu erzeugenden Artefakte?

Minimierung des Entwicklungsaufwandes Die Hauptaufgabe eines Baukastens ist die Reduzierung des Aufwandes zur Implementierung einer Anwendung. Dies gilt auch für den MOmo-Baukasten, der den Aufwand zur Implementierung von Codegeneratoren für MOF-basierte Modellierungssprachen minimieren soll. Aus diesem Grund soll bei der Implementierung eines Codegenerators für eine neue Kombination von Modellierungssprache und Implementierungstechnologie nur der implementierungsabhängige Teil neu zu entwickeln sein. Alle anderen Bestandteile des Generators muss der Baukasten entweder bereitstellen oder durch enthaltene Werkzeuge erzeugen können.

Weiterhin soll die verbesserte Integration von MOF und UML in den Versionen 2.0 genutzt werden können, um eine Familie von Codegeneratoren für spezifische Modellierungssprachen zu entwickeln. Um dies zu ermöglichen, darf eine Erweiterung oder Veränderung einer Sprache nur Entwicklungsaufwand für die neuen bzw. veränderten Modellierungselemente erfordern.

Verwendung standardisierter Technologien und Werkzeuge Die Implementierung eines Codegenerators für eine Modellierungssprache erfordert eine Abbildung der Elemente der Modellierungssprache auf Elemente der Zielsprache bzw. der Zieltechnologie. Solche Abbildungen können von erheblicher Komplexität sein und zu einem entsprechend hohen Zeitaufwand bei ihrer Implementierung führen. Der benötigte Zeitaufwand steigt zusätzlich, wenn der Entwickler sich in ihm unbekannte Technologien einarbeiten muss. Aus diesem Grund sollen standardisierte und weit verbreitetete Technologien zur Implementierung der Codegeneratoren eingesetzt werden. Dies vermindert zusätzlich die Abhängigkeit von einzelnen Werkzeugen und verringert die Einarbeitungszeit für die Implementierung neuer Anwendungen.

4.2 Entwurf

Die Aufgabe des MOmo-Baukastens ist es, die Entwicklung von Codegeneratoren für MOF-basierte Modellierungssprachen zu erleichtern. Dazu werden verschiedene Komponenten benötigt, die einzelne Arbeitsschritte bei der Entwicklung eines Codegenerators unterstützen und die in Abschnitt 4.1 beschriebenen Anforderungen erfüllen. Im Folgenden wird der Aufbau eines MOmo-Codegenerators schrittweise vorgestellt. Ausgewählte Komponenten, die jeder MOmo-Generator enthält, werden in den nachfolgenden Kapiteln 8 und 9 noch einmal im Detail beschrieben.

Der wichtigste Anwendungsfall, für den der MOmo-Baukasten entwickelt wurde, ist die Entwicklung von Codegeneratoren für Modellierungssprachen, die durch ein MOF-Modell definiert sind. Einige der Komponenten, die für diese Anwendung benötigt werden, lassen sich auch zur Entwicklung anderer Werkzeuge verwenden. Der Aufbau des Baukastens ist wesentlich durch den erstgenannten Einsatzzweck motiviert. Im Folgenden wird aus diesem Grund beschrieben, wie MOmo-Generatoren aufgebaut sind. Andere Anwendungen, die mit Hilfe des MOmo-Baukasten implementiert werden, enthalten eine Teilmenge der beschriebenen Komponenten.

Abb. 4.4 zeigt das allgemeine Szenario des Einsatzes eines Codegenerators für eine MOF-basierte Modellierungssprache. Der Codegenerator erhält ein XMI-Dokument, das das zu transformierende Modell repräsentiert, als Eingabe und erzeugt die durch eine Konfiguration vorgegebenen Artefakte. In der Regel besteht die Menge der zu erzeugenden Artefakte aus Dateien, die Quelltexte der gewählten Zielsprache enthalten. Die Quelltexte können durch Anwendung entsprechender Compiler in eine lauffähige Repräsentation des Modells übersetzt werden.

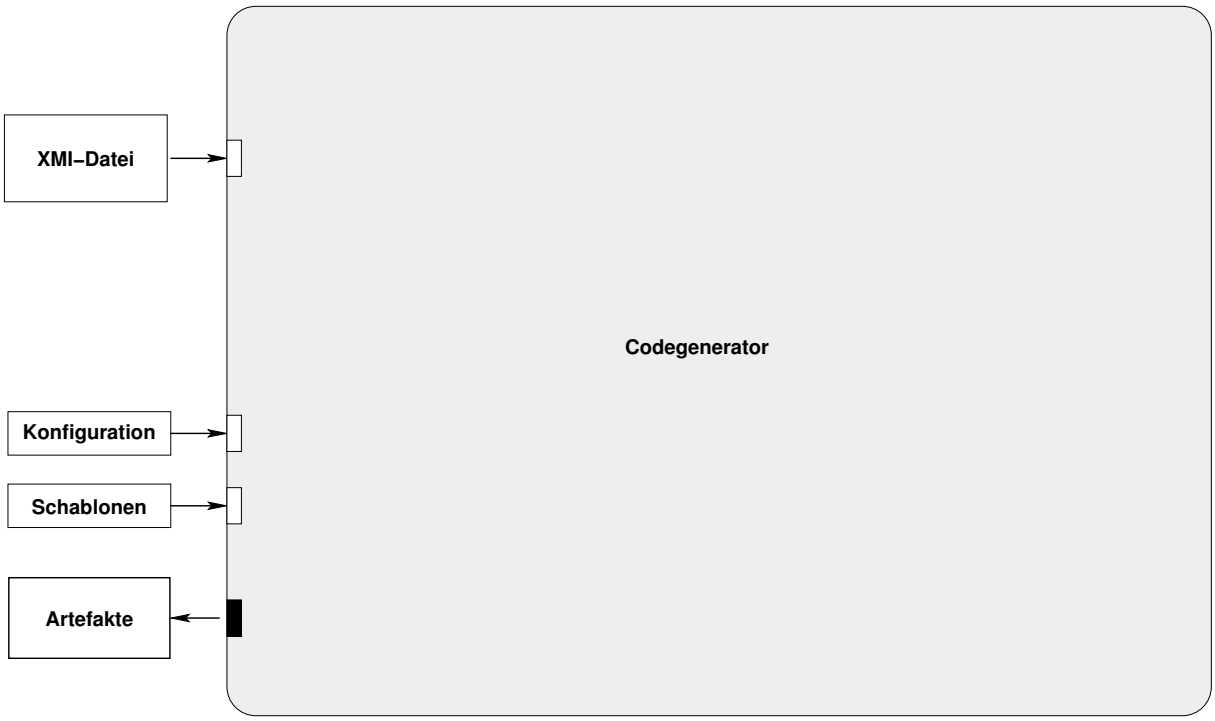


Abbildung 4.4: Aufgabe eines MOmo-Generators

In Abb. 4.4 wird der MOmo-Codegenerator durch eine Komponente dargestellt, die durch ein Rechteck mit abgerundeten Ecken notiert ist. Weitere Rechtecke repräsentieren die Daten, die in den Generator ein- bzw. aus dem Generator ausfließen. Datenflüsse werden durch Pfeile zwischen Daten und Ein- bzw. Ausgängen dargestellt. Eingänge werden durch kleine weiße und Ausgänge durch kleine schwarze Rechtecke markiert. Dieselbe Notation wird im Folgenden zur schrittweisen Beschreibung des Aufbaus von MOmo-Generatoren verwendet. Die folgenden Abbildungen enthalten neben den bereits beschriebenen Notationselementen Pfeile zwischen zwei Ein- bzw. zwei Ausgängen. Solche Verbindungen kennzeichnen die Weiterleitung von Daten zwischen hierarchisch angeordneten Komponenten. Eine Komponente kann Daten aus einem Eingang einer direkt übergeordneten Komponente erhalten und an einen Ausgang einer direkt übergeordneten Komponente weiterreichen.

4.2.1 Aufbau von MOmo-Generatoren

Um eine Repräsentation eines Modells aufbauen zu können, wird ein *XMI-Reader* benötigt, der XMI-Dokumente einlesen kann. Während des Einlesens werden die im XMI-Dokument enthaltenen Beschreibungen von Modellelementen verwendet, um eine Generator-interne Repräsentation des Modells aufzubauen. Zusätzlich werden daher Implementierungen von Klassen benötigt, die instanziiert werden können, um die einzelnen Elemente des Modells abzubilden. Sowohl der Reader als auch die Klassen werden durch den MOmo-Baukasten indirekt zur Verfügung gestellt, wenn die Syntax der Modellierungssprache, die durch den Generator verarbeitet werden soll, durch ein MOF-Modell beschrieben werden kann. Der MOmo-Baukasten enthält einen MOF-Generator, der Implementierungen von MOF-Modellen aus XMI-Dokumenten erzeugen kann. Die erzeugten Implementierungen enthalten sowohl den Reader als auch Instanzen aller Klassen, die durch das MOF-Modell festgelegt werden. Diese Instanzen werden im Folgenden als *Metaobjekte* bezeichnet. Die im Rahmen dieser Arbeit entstandene Realisierung des MOmo-Baukastens verwendet die Programmiersprache Java zur Implementierung von Reader und MOF-Modellen. Die Schnittstellen, die entsprechende Implementierungen bereitstellen, werden in Kapitel 8 beschrieben.

Für jedes MOF-Modell kann eine XMI-Schnittstelle erzeugt werden, die aus je einer Klasse zum Lesen und Schreiben von XMI-Dokumenten besteht. Da der MOF-Codegenerator des MOmo-Baukastens die MOF-Version 2.0 implementiert, wird eine XMI-Schnittstelle erzeugt, die konform zur XMI-Version 2.1 ist. Zusätzlich enthält der MOmo-Baukasten zwei weitere XMI-Reader, die frühere Versionen des XMI-Standards lesen und MOF 1.x und UML 1.x in MOF 2.0 Modelle umwandeln können. Diese Reader wurden benötigt, um die Komponenten des MOmo-Baukastens entwickeln zu können, weil die meisten Modellierungswerkzeuge noch nicht auf die neuen Versionen des UML- bzw. MOF-Standards umgestellt worden sind. Die vorhandenen Reader implementieren die Regeln zur Transformation von MOF 1.4 Modellen nach MOF 2.0, die in [ACC+03, Kapitel 11] definiert sind. Zum Einlesen von UML-Modellen werden ähnliche Regeln angewendet. Da UML-Modelle Elemente enthalten können, die sich in MOF nicht darstellen lassen, kann allerdings nicht jedes UML-Modell eingelesen werden.

Neben den Komponenten, die zum Einlesen eines Modells benötigt werden, enthalten MOmo-Generatoren weitere Komponenten, die für die eigentliche Durchführung der Codegenerierung zuständig sind. Auch diese Komponenten stellt der MOmo-Baukasten teilweise zur Verfügung. In Abb. 4.5 ist der minimale Aufbau eines MOmo-Generators dargestellt.

Ein Codegenerator erhält ein XMI-Dokument, das das zu übersetzende Modell darstellt, eine Konfiguration, die den Ablauf des Generierungsprozesses steuert und eine Menge von Schablonen, die die Abbildung von Metaobjekten auf Artefakte beschreiben. Der Reader erzeugt eine Menge von Metaobjekten, die das zu übersetzende Modell repräsentieren, und reicht diese an die Schablonenablaufumgebung weiter. Die Schablonenablaufumgebung wendet die Schablonen auf die Metaobjekte an. Welche Schablonen auf einen Metaobjekttyp anzuwenden sind, wird durch die Konfiguration bestimmt. Durch die Ausführung der Schablonen wird eine Menge von Artefakten erzeugt.

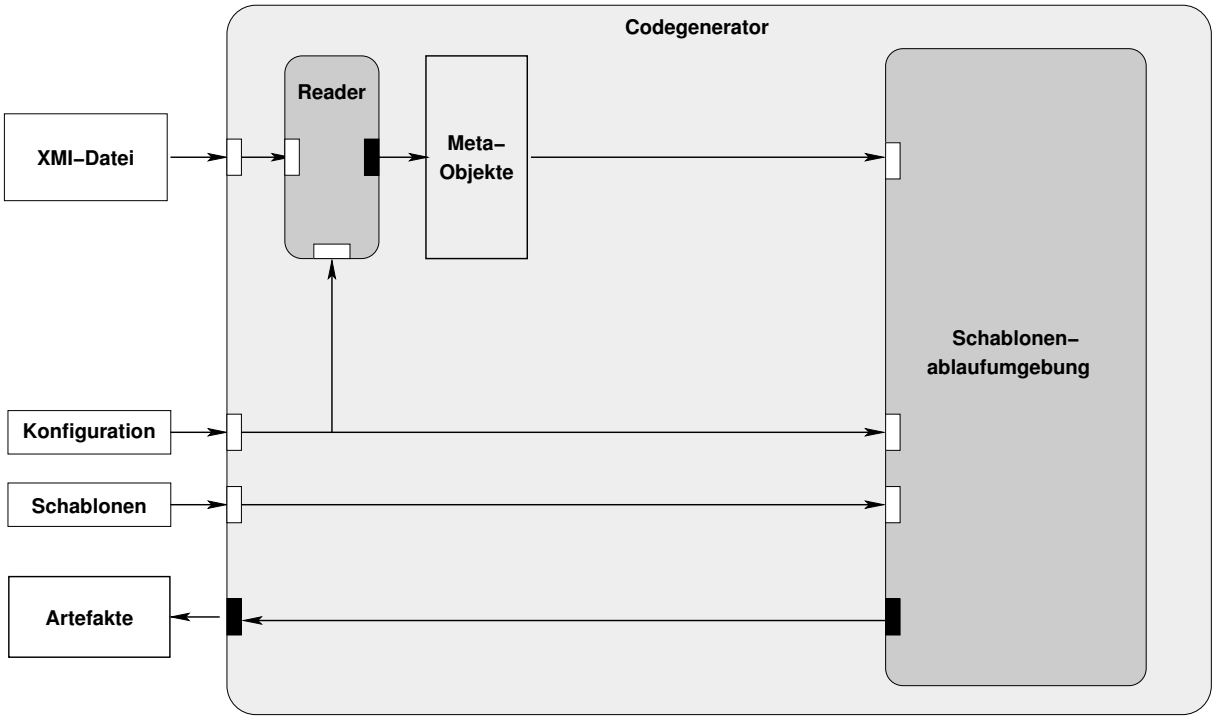


Abbildung 4.5: Prinzipieller Aufbau eines MOmo-Codegenerators

4.2.2 Schablonenablaufumgebung

Die Schablonenablaufumgebung setzt sich aus zwei Komponenten zusammen. Im Unterschied zu Reader und Metaobjekten werden diese Komponenten nicht durch einen Generator erzeugt. Stattdessen enthält der MOmo-Baukasten generische Komponenten, die jede beliebige MOF-basierte Modellierungssprache verarbeiten können. Eine Komponente verwendet die Repräsentation, die durch den Reader aufbaut wird, um sog. *Kontexte* zu erzeugen. Unter einem Kontext wird eine Aufbereitung eines Modellelementes oder einer Gruppe von Modellelementen verstanden, die zur Verarbeitung durch einen Schablonenmechanismus geeignet ist. Als Schablonenmechanismus wird im Folgenden alles angesehen, was zur Formulierung und Ausführung von Schablonen geeignet ist. Ein Kontext zur Erzeugung von Quelltext für eine Klasse muss beispielsweise den Zugriff auf den Namen, die Attribute und Operationen ermöglichen.

Abb. 4.6 zeigt die Aufgliederung der Schablonenablaufumgebung in die Komponenten **Kontextgenerator** und **Schablonenausführer**. Der Kontextgenerator erzeugt eine Menge von Kontexten, deren Aufbau durch die Konfiguration beeinflusst werden kann. Die Menge der Kontexte wird an den Schablonenausführer weitergegeben, der in Abhängigkeit von der Konfiguration die Artefakte erzeugt.

Der konkrete Aufbau der Kontexte und der Schablonenablaufumgebung hängt vom gewählten Schablonenmechanismus ab. In Kapitel 9 werden verschiedene Möglichkeiten diskutiert, die sich zur Realisierung der Ablaufumgebung eignen. Für die Implementierung des

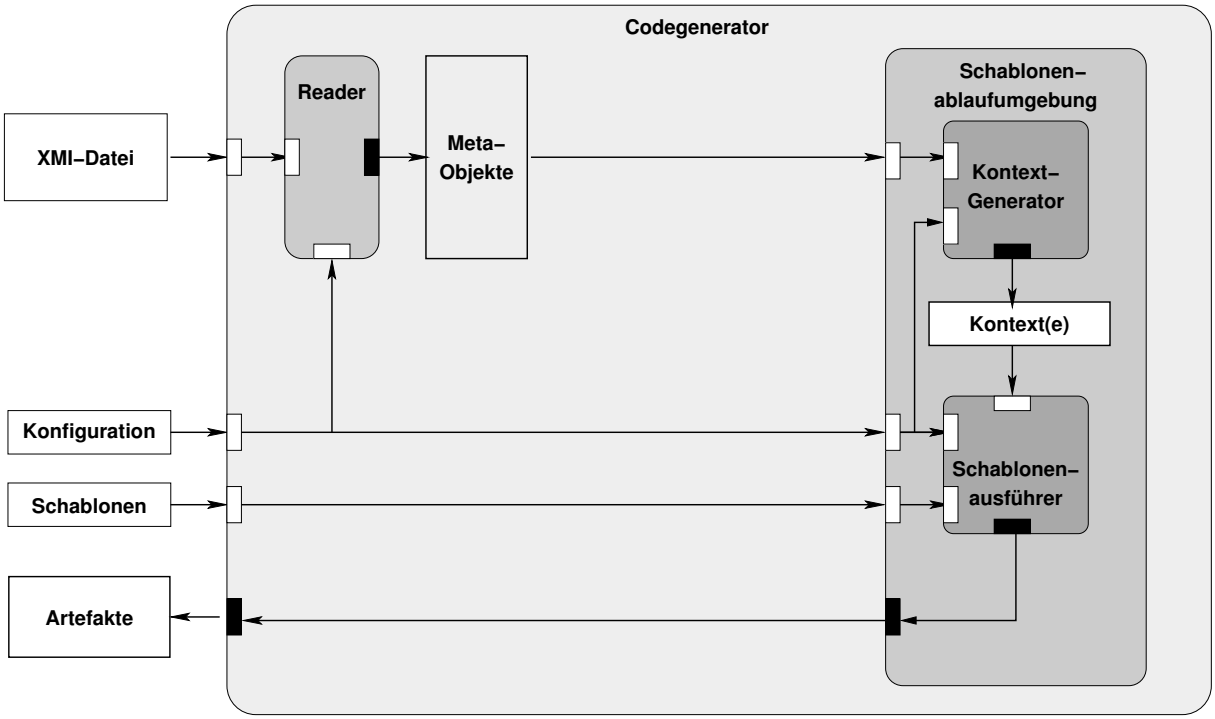


Abbildung 4.6: Vollständiger Aufbau eines MOmo-Codegenerators

MOmo-Baukastens im Rahmen dieser Arbeit wurde ein XSLT-basierter [Cla99b] Ansatz ausgewählt, so dass die Kontexte durch XML-Dokumente implementiert werden.

4.2.3 Hinzufügen optionaler Komponenten

Neben den bisher beschriebenen, zwingend notwendigen Komponenten kann ein MOmo-Generator weitere optionale Komponenten enthalten. Diese Komponenten werden im Weiteren als *MOmo-Module* bezeichnet und können an zwei Stellen in den Generierungsprozess eingebunden werden. Die erste Möglichkeit ist die Definition eines MOmo-Moduls, das die vom Reader aufgebaute Objektrepräsentation verändert, bevor die Kontexte erzeugt werden. Die zweite Möglichkeit ist, ein MOmo-Modul auf die generierten Artefakte anzuwenden.

Die maximale Konfiguration eines MOmo-Generators ist in Abb. 4.7 dargestellt. Gegenüber der in Abb. 4.6 dagestellten Basisstruktur enthält der Generator MOmo-Module, die die Metaobjekte verändern können, und Formatierer, die auf die generierten Artefakte angewendet werden können.

Die Metaobjekte, die durch den Reader erzeugt werden, können durch benutzerdefinierte Module verändert werden, die eine durch den MOmo-Baukasten vorgegebene Schnittstelle implementieren. Jedes Modul, das in der Konfiguration angegeben ist, wird vom Steuerprogramm initialisiert und ausgeführt. Zur Initialisierung erhält es die Menge aller vom Reader erzeugten Metaobjekte.

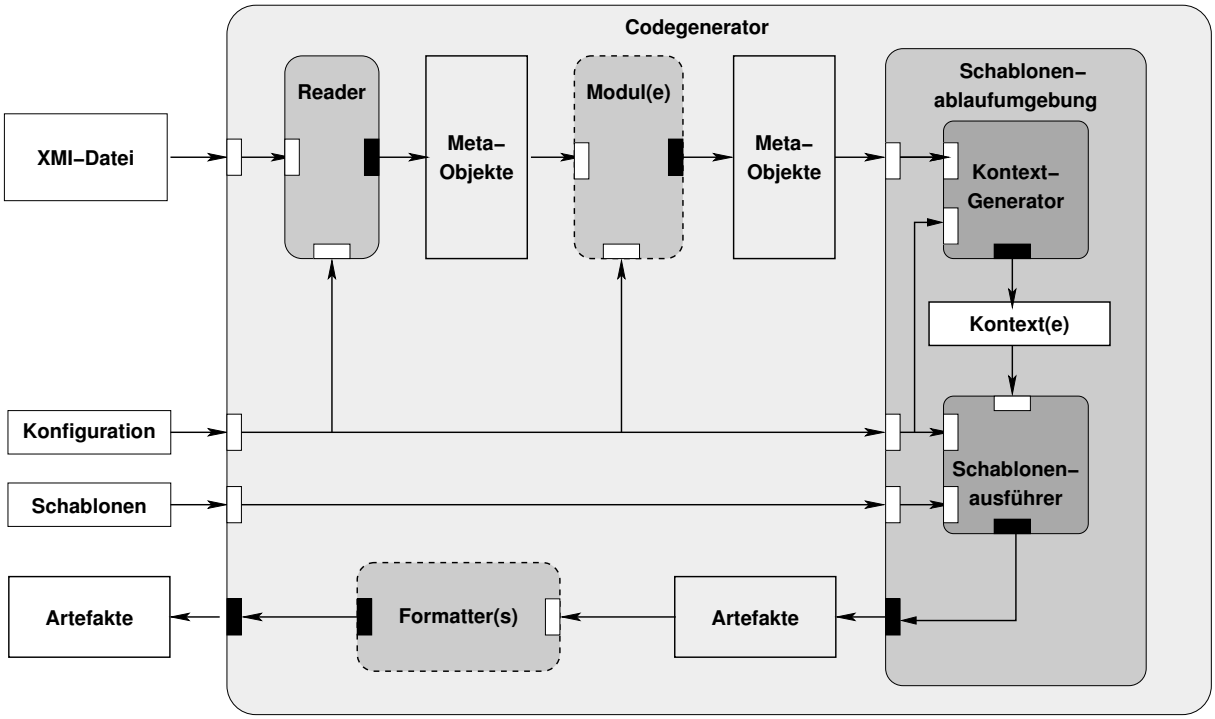


Abbildung 4.7: Aufbau eines MOmo-Codegenerators mit optionalen Komponenten

Der Grund für die Einführung der Modulschnittstelle zwischen Reader und XML-Generator ist, dass komplexe Operationen durch Manipulation der Metaobjekte leichter realisiert werden können als in späteren Phasen der Codegenerierung. Dies gilt insbesondere für Sonderfälle wie zum Beispiel die Verarbeitung eines UML-Modells durch einen MOF-Generator. In diesem Fall müssen die in UML definierten Datentypen auf die MOF-Datentypen abgebildet werden, um die Schablonen unverändert verwenden zu können.

Der MOmo-Baukasten enthält bereits einige Komponenten, die zur Anpassung eines gegebenen Modells verwendet werden können. Ein MOmo-Modul führt die gerade beschriebene Abbildung von Datentypen durch, um auch UML-Modelle durch den im Baukasten enthaltenen MOF-Generator verarbeiten zu können. Diese Funktionalität wurde für die Erzeugung der Komponenten des Baukastens benötigt, weil noch keine Werkzeuge zur Erstellung von UML/MOF 2.0 Modellen verfügbar waren, und daher die von der OMG bereitgestellten UML-Modelle zur initialen Erzeugung der MOF-Metamodellimplementierung verwendet werden mussten.

MOF bietet zur Strukturierung eines Modells das Modellierungselement *Paket* an, das als Container für weitere Modellierungselemente verwendet wird. Zwischen Paketen können Beziehungen bestehen, die die Verwendung der Elemente eines Paketes in einem anderen Paket festlegen. Die MOF-Version 2.0 führt eine neue Beziehung zwischen Paketen, sog. *Paketvereinigungen* ein. Ein zweites MOmo-Modul führt die Expansion von Paketvereinigungen durch (siehe Abschnitt 5.3.4). In der Spezifikation der UML-Infrastruktur wird angegeben, dass in einem XMI-Dokument entweder alle Paketvereinigungen eines Modells erhalten bleiben oder

alle expandiert werden müssen. Expandieren einer Paketvereinigung bedeutet, dass die Vereinigung durch die entsprechenden Import-, Redefinitions- und Vererbungsbeziehungen ersetzt wird. Dieser Ersetzungsprozess kann die Erzeugung neuer Metaobjekte beinhalten und sollte daher vor der Anwendung der Schablonen durchgeführt werden. Die Schablonen können dann unverändert verwendet werden, um die Artefakte zu erzeugen.

Die Möglichkeit, auf die bereits vollständig generierten Artefakte weitere Bearbeitungsschritte anzuwenden, kann prinzipiell für beliebige Veränderungen der Artefakte angewendet werden, wird aber im Wesentlichen zur Formatierung der Artefakte benutzt. In diesem Fall beschließt ein Formatierungsschritt den Transformationsvorgang. Dieser Schritt ist hauptsächlich zur Unterstützung der Entwicklung von Generatoren vorgesehen. Er wird notwendig, weil beispielsweise die Ausgabe einer durch XSLT [Cla99b] durchgeführten Transformation häufig unlesbar ist, und dadurch die Fehlersuche im generierten Code erschwert wird.

4.3 Beispiel

Im Folgenden wird ein Beispiel für den Ablauf einer Codeerzeugung durch einen MOmo-Codegenerator dargestellt. Zunächst wird eine sehr einfache komponentenorientierte Modellierungssprache vorgestellt. Diese Sprache ist nicht zur Modellierung realer Anwendungen, sondern lediglich zur Demonstration der wesentlichen Eigenschaften einer MOF-basierten Modellierungssprache entworfen worden und enthält deshalb nur wenige Modellierungselemente. Als bespielhafte Anwendung der Sprache wird der Aufbau einer Waschmaschine modelliert. Anschließend werden die Artefakte beschrieben, die während der Verarbeitung durch einen MOmo-Codegenerator erzeugt werden.

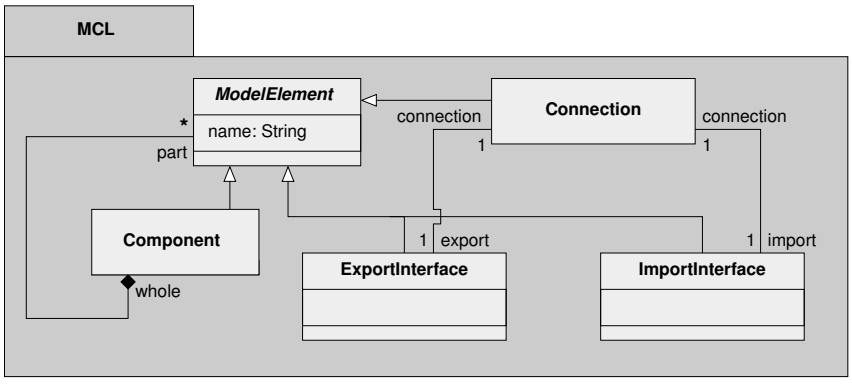


Abbildung 4.8: Definition der *Minimal Component Language* als MOF-Modell

Abb. 4.8 zeigt das MOF-Modell der *Minimal Component Language* (MCL). Die MCL besteht aus einem Paket **MCL**, das die Klassen **ModelElement**, **Component**, **ExportInterface** , **ImportInterface** und **Connection** enthält. Die abstrakte Klasse **ModelElement** stellt die Möglichkeit bereit, ein Modellelement zu benennen. Diese Eigenschaften wird von allen anderen Klassen des Modells geerbt. Instanzen der Klasse **Component** repräsentieren Komponenten,

die der wichtigste Bestandteil von MCL-Modellen sind. Eine Komponente kann andere Modellelemente enthalten. Die enthaltenen Modellelemente sind Instanzen der Klassen **Component**, **ExportInterface**, **ImportInterface** und **Connection**.

Dadurch wird abgebildet, dass eine Komponente aus anderen Komponenten bestehen, Schnittstellen bereitstellen bzw. anfordern, und Verbindungen zwischen Schnittstellen enthaltener Komponenten einschliessen kann.

Eine bereitgestellte Schnittstelle, die durch eine Instanz der Klasse **ExportInterface** repräsentiert wird, kann mit einer geforderten Schnittstelle, die durch eine Instanz der Klasse **ImportInterface** dargestellt wird, verbunden werden. Solche Verbindungen werden durch Instanzen der Klasse **Connection** repräsentiert.

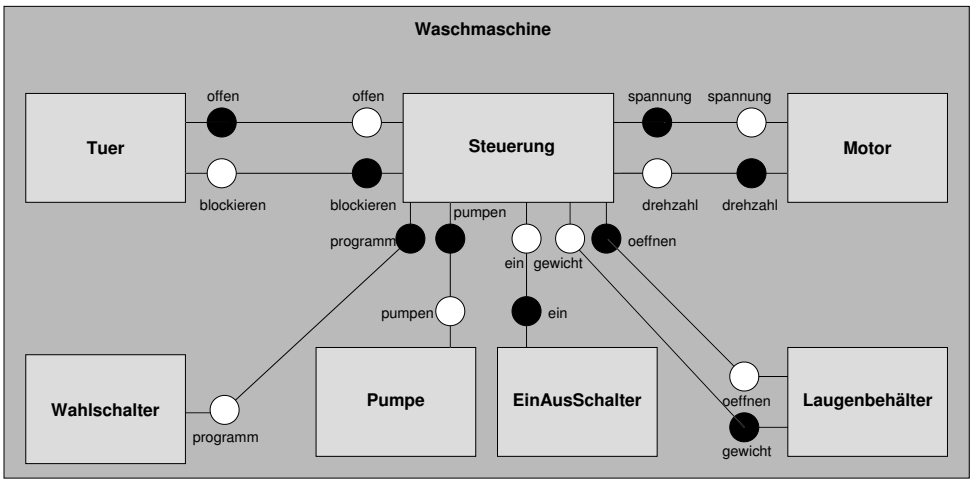


Abbildung 4.9: Beispiel in MCL

Abb. 4.9 zeigt die grafische Notation eines MCL-Modells, das den vereinfachten Aufbau einer Software zur Steuerung einer Waschmaschine darstellt. MCL-Komponenten werden durch Rechtecke dargestellt. Die Schnittstellen einer Komponente werden durch „Lollipops“ repräsentiert. Ein weißer Lollipop steht für eine Schnittstelle, die von der Komponente exportiert wird, und ein schwarzer Lollipop kennzeichnet eine Schnittstelle, die von einer Schnittstelle importiert werden soll. Eine Komponente ist genau dann funktionsfähig, wenn alle ihre Importschnittstellen mit einer Exportschnittstelle verbunden sind.

Der dargestellte Ausbau zeigt die Hauptkomponente **Waschmaschine**, die sieben Komponenten enthält. Die Komponenten **Tuer**, **Wahlschalter**, **Pumpe**, **EinAusSchalter**, **Laugenbehaelter** und **Motor** repräsentieren typische Bestandteile einer Waschmaschine. Das Zusammenspiel dieser Bestandteile während eines Waschvorgangs wird durch eine weitere Komponente, die **Steuerung**, bestimmt.

Die Steuerung muss jederzeit über den Zustand der Tür (offen oder geschlossen) informiert sein und kann die Tür blockieren, um ein Öffnen der Tür während des Waschvorgangs zu verhindern. Um über den aktuellen Zustand der Tür informiert werden zu können, exportiert die Komponente **Steuerung** die Schnittstelle **offen**. Diese Schnittstelle wird von der Komponente **Tuer**

importiert, so dass diese Zustandsänderungen an die Komponente **Steuerung** melden kann. Im Gegenzug exportiert die Komponente **Tuer** die Schnittstelle **blockieren**, die von der Komponente **Steuerung** importiert wird. Über diese Verbindung kann die Steuerung ein Blockieren der Tür auslösen.

Die Komponente **Wahlschalter** ermöglicht die Auswahl eines Waschprogramms, das von der Steuerung ausgeführt werden soll. Dazu wird die Schnittstelle **programm** von der Komponente **Wahlschalter** exportiert und von der Komponente **Steuerung** importiert. Wird ein Waschvorgang gestartet, fragt die Steuerung über diese Verbindung das ausgewählte Waschprogramm ab. Das Starten des Waschvorgangs erfolgt durch die Komponente **EinAusSchalter**, die die Schnittstelle **ein** von der Komponente **Steuerung** importiert. Wird der Ein-Aus-Schalter betätigt, meldet die Komponente **EinAusSchalter** dies an die Steuerung.

Während eines Waschvorgangs wird mehrfach Wasser in den Laugenbehälter eingelassen und wieder abgepumpt. Das Einlassen von Wasser kann die Komponente **Steuerung** über die Schnittstelle **oeffnen** auslösen, die sie von der Komponente **Laugenbehaelter** importiert. Um Flüssigkeit aus dem Laugenbehälter abpumpen zu können, importiert die Komponente Steuerung die Schnittstelle **pumpen** von der Komponente **Pumpe**. Außerdem soll die Komponente **Laugenbehaelter** ihr aktuelles Gewicht an die Komponente **Steuerung** melden können, um das Starten des Waschvorgangs bei überladener Maschine zu verhindern. Dazu importiert die Komponente **Laugenbehaelter** die Schnittstelle **gewicht** von der Komponente **Steuerung**.

Durch die beiden Verbindungen zwischen den Komponenten **Steuerung** und **Motor** wird die Regelung der Drehzahl der Trommel ermöglicht, die durch den Motor angetrieben wird. Die Komponente **Steuerung** importiert die Schnittstelle **spannung** von der Komponente **Motor**, um durch Verändern der Spannung die Drehzahl beeinflussen zu können. Außerdem importiert die Komponente **Motor** die Schnittstelle **drehzahl** von der Komponente **Steuerung**, um die aktuelle Drehzahl an die Steuerung melden zu können.

Um aus dem beschriebenen Modell Code generieren zu können, muss eine Repräsentation erzeugt werden, die von einem Codegenerator verarbeitet werden kann. MOmo-Codegeneratoren erwarten XMI-Dokumente als Eingabe. Abb. 4.10 zeigt ein XMI-Dokument, das einen Ausschnitt des in Abb. 4.9 dargestellten Modells repräsentiert. Im Folgenden soll der Ausschnitt kurz erläutert werden; eine genauere Beschreibung des Aufbaus von XML- bzw. XMI-Dokumenten enthalten die Kapitel 6 und 7. Jede Komponente wird durch ein XML-Element des Knotentyps **mcl:Component** repräsentiert, das einen eindeutigen Bezeichner, den Namen der Komponente und den Bezeichner der übergeordneten Komponente enthält. In Abb. 4.10 sind nur die drei Komponenten **Waschmaschine**, **Steuerung** und **Tuer** repräsentiert. Alle weiteren Komponenten der Waschmaschine werden in gleicher Weise dargestellt.

Jede Komponente kann mehrere andere Modellelemente enthalten. Enthaltene Modellelemente werden durch **part**-Knoten ausgedrückt. Im dargestellten Ausschnitt wird auf diese Weise zum einen ausgedrückt, dass die Komponenten **Steuerung** und **Tuer** in der Komponente **Waschmaschine** enthalten sind. Zum anderen werden die Schnittstellen der Komponenten ebenfalls durch untergeordnete **part**-Knoten dargestellt, weil sie Bestandteile der jeweiligen Waschmaschinen-Komponenten sind.

```
<xmi:XMI version='2.1'
         xmlns:xmi = 'http://www.omg.org/XMI'
         xmlns:mcl = 'http://ist.unibwm.de/MCL'>
  ...
  <mcl:Component xmi.id='001' name='Waschmaschine' whole='nil'>
    <part xmi.id='002' type='mcl:Component' name='Steuerung' whole='001'>
      <part xmi.id='003' type='mcl:ExportInterface'
            name='offen' whole='002' />
      <part xmi.id='004' type='mcl:ImportInterface'
            name='blockieren' whole='002'/>
      <part xmi.id='005' type='mcl:ImportInterface'
            name='spannung' whole='002'/>
      <part xmi.id='006' type='mcl:ExportInterface'
            name='drehzahl' whole='002'/>
       weitere Bestandteile der Komponente "Steuerung"
    </part>
    <part xmi.id='012' type='mcl:Component' name='Tuer' whole='001'>
      <part xmi.id='013' type='mcl:ImportInterface'
            name='offen' whole='012' />
      <part xmi.id='014' type='mcl:Interface'
            name='blockieren' whole='012' />
    </part>
    <part xmi.id='015' type='mcl:Connection' name=''
          export='003' import='013' />
     weitere Bestandteile der Komponente "Waschmaschine"
   </part>
  </mcl:Component>
</xmi:XMI>
```

Abbildung 4.10: Ausschnitt eines XMI-Dokumentes für Abb. 4.9

Bereitgestellte und geforderte Schnittstellen werden durch **part**-Elemente mit den **type**-Attributen **mcl:ExportInterface**- und **mcl:-Import-Interface** repräsentiert. Die Darstellung der Verbindungen zwischen bereitgestellten und geforderten Schnittstellen erfolgt durch **part**-Elemente mit dem **type**-Attribut **mcl:Connection**. Das Beispiel in Abb. 4.10 zeigt die XMI-Repräsentation der Verbindung zwischen den Schnittstellen zur Übergabe des aktuellen Status' der Tür an die Steuerung. Die Repräsentation der Komponente **Steuerung** mit dem Bezeichner **002** enthält ein **part**-Element mit dem Bezeichner **003**, das die bereitgestellte Schnittstelle **offen** darstellt. Das XMI-Element für die Komponente **Tuer** enthält ein **part**-Element mit dem Bezeichner **013**, das die geforderte Schnittstelle **offen** der Komponente **Tuer** darstellt. Die Verbindung der Schnittstellenrepräsentationen erfolgt durch das XMI-Element mit dem Bezeichner

015, das die Instanz der Klasse **Connection** repräsentiert, die die beiden Schnittstellen verbindet. Entsprechend sind die Attribute des XMI-Elementes mit den Bezeichnern der beiden XMI-Elemente belegt, die die Schnittstellen darstellen.

Der generierte XMI-Reader eines MOmo-MCL-Codegenerators liest das XMI-Dokument und erzeugt eine Menge von Metaobjekten, die das im Dokument enthaltene Modell repräsentieren. Die Metaobjekte werden an den Kontextgenerator weitergereicht, der die enthaltenen Informationen für den Schablonenausführer aufbereitet. Auf diese Weise ist das Konzept der MOmo-Codegeneratoren unabhängig von XMI, d. h. die Schablonen müssen die Eigenschaften eines XMI-Dokumentes nicht berücksichtigen. Dies ist vorteilhaft, weil XMI-Dokumente meist vollständige Modelle enthalten und die zu generierenden Artefakte dagegen meist für einzelne Metaobjekte bzw. Metaobjektgruppen erzeugt werden müssen. Zudem kann die Erzeugung von XMI-Dokumenten vielfältig konfiguriert werden, so dass in den Schablonen verschiedene Repräsentationen desselben Modellelements verarbeitet werden müssten. Eine geeignete Wahl der Repräsentation der Kontexte kann daher erheblich zur Vereinfachung der Schablonen beitragen, die die „eigentliche“ Codeerzeugung übernehmen. Da die Schablonen der einzige Teil sind, der bei der Entwicklung eines MOmo-Codegenerators manuell erstellt werden muss, wird eine deutliche Reduzierung des Entwicklungsaufwandes erreicht.

Das Konzept des MOmo-Generators sieht aus diesem Grund vor, jedes Metaobjekt durch einen Kontext zu repräsentieren. Entsprechende Kontexte können durch generische Kontextgeneratoren für jedes Modell automatisch erzeugt werden, das durch eine MOF-basierte Modellierungssprache beschrieben ist. Der konkrete Aufbau eines Kontextes hängt von der gewählten Realisierung des Schablonenausführers ab. Wird beispielsweise XSLT angewendet, ist jeder Kontext ein XML-Dokument. Ein entsprechender Kontextgenerator ist im MOmo-Baukasten enthalten. Wird dagegen der Schablonenmechanismus Velocity [Vel] benutzt, bestehen die Kontexte aus Mengen von „Schlüssel=Wert“-Paaren. Da hier das Prinzip und nicht die Implementierung eines MOmo-Codegenerators dargestellt werden soll, zeigt Abb. 4.11 eine abstrakte Darstellung der Kontexte, die für einige der Modellelemente aus Abb. 4.10 erzeugt werden.

Kontexte enthalten jeweils nur die Informationen, die direkt zu einem Metaobjekt gehören. In Abb. 4.11 sind die sechs Kontexte dargestellt, die zur Darstellung der Verbindung zwischen der von der Komponente **Steuerung** exportierten und von der Komponente **Tuer** importierten Schnittstelle **offen** erzeugt werden. Die verschiedenen Kontexttypen zur Repräsentation von Komponenten, Schnittstellen und Verbindungen sind durch unterschiedliche Grautöne gekennzeichnet. Alle Referenzierungen zwischen Metaobjekten werden durch Verweise zwischen Kontexten aufgelöst. Die erzeugten Verweise sind durch Pfeile von den Attributen der Kontexte zum jeweils referenzierten Kontext dargestellt. Die Darstellung eines Modells durch eine Menge von Kontexten ist also sehr ähnlich zur Repräsentation des Modells durch Objekte. Der Unterschied liegt in der Codierung der Modellelemente.

Auf die einzelnen Kontexte werden nachfolgend Schablonen angewendet, um die Artefakte der Zielsprache oder -technologie zu erzeugen. Nachfolgend werden drei Schablonen angegeben, um aus dem Modell in Abb. 4.9 Java-Code zu erzeugen. Die Schablonen sind in Pseudo-Code beschrieben und enthalten drei verschiedene Bestandteile. Alle normal dargestellten Bestandteile werden in den resultierenden Code übernommen, alle in spitzen Klammern („<“ bzw. „>“)

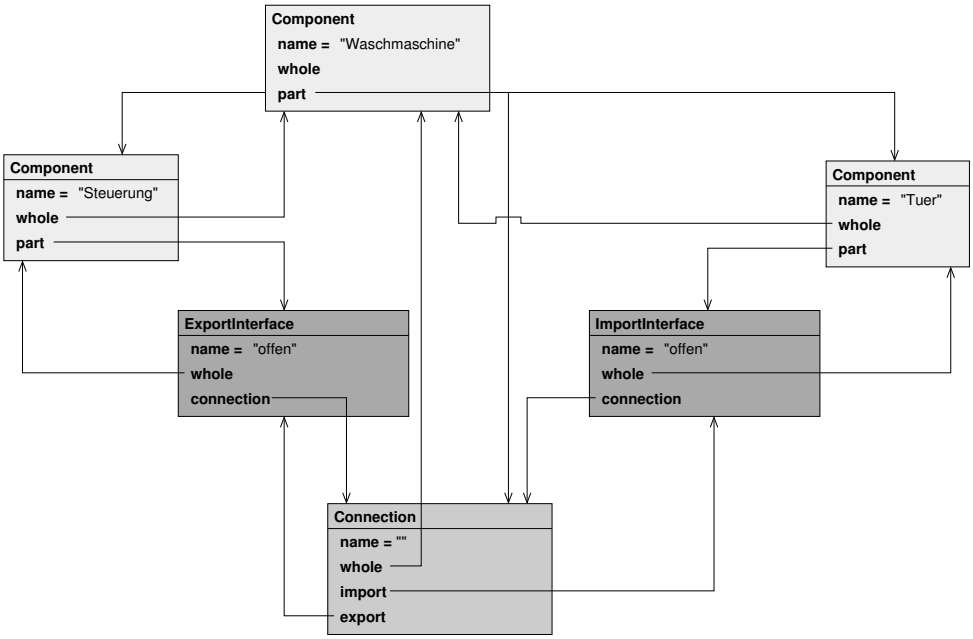


Abbildung 4.11: Kontexte für einige Modellelemente aus Abb. 4.10

dargestellten Anteile sind Zugriffe auf Informationen aus dem Modell, und alle in Großbuchstaben beschriebenen Bestandteile sind Pseudo-Code, der die Auswahl der Informationen aus dem Modell beschreibt.

```
public class <Name der Komponente> {

    public static main(String[] args) {
        <Name der Komponente> mainComponent = new <Name der Komponente>();
    }

    // WENDE SCHABLONE ZUR ERZEUGUNG DES KOMPONENTENRUMPFES AN
}
```

Abbildung 4.12: Schablonen für die Hauptkomponente

Die in Abb. 4.12 dargestellte Schablone wird auf die Kontexte der Komponenten der obersten Schachtelungsebene angewendet. Die Schablone erstellt eine Java-Klasse, die den Namen der Komponente erhält. Außerdem wird eine **main**-Methode erzeugt, die die Hauptkomponente instanziiert und eine weitere Schablone angewendet, die die weiteren Bestandteile der Klasse erzeugt. Diese Schablone wird in Abb. 4.13 dargestellt. Zur Erzeugung von Code für enthaltene Komponenten wird eine Schablone verwendet, die sich von der in Abb. 4.12 dargestellten Schablone lediglich durch den Wegfall der Erzeugung der **main**-Methode unterscheidet.

Die in Abb. 4.13 dargestellte Schablone erzeugt den vollständigen Rumpf einer Java-Klasse zur Repräsentation einer MCL-Komponente. Zunächst wird der Konstruktor erzeugt. Im Inneren des Konstruktors werden alle enthaltenen Komponenten instanziiert und alle Verbindungen

```
public <Name der Komponente> {
    // instanziiere enthaltene Komponenten
    // FÜR JEDE ENTHALTENE KOMPONENTE:
      k<zaehler> = new <Name der enthaltenen Komponente>();

    // verbinde Schnittstellen
    // FÜR JEDE ENTHALTENE VERBINDUNG:
      <Eigentümer der Importschnittstelle>.set \
          <Name der Importschnittstelle>(
              <Eigentümer der Exportschnittstelle>);
}

// enthaltene Komponenten
// FÜR JEDE ENTHALTENE KOMPONENTE:
  private <Name der enthaltenen Komponente> k<zaehler>

// set-Methoden für importierte Schnittstellen
// FÜR JEDE ENTHALTENE IMPORTSCHNITTSTELLE:
public void set_<Name der Importschnittstelle>(
    <Name des Eigentümers der verbundenen Exportschnittstelle> v) {
    <Name der Importschnittstelle>Import = v;
}

// Methoden für exportierte Schnittstellen
// FÜR JEDE ENTHALTENE EXPORTSCHNITTSTELLE:
public void <Name der Exportschnittstelle>() {}

// Methoden für importierte Schnittstellen
// FÜR JEDE ENTHALTENE IMPORTSCHNITTSTELLE:
private void <Name der Importschnittstelle>() {
    <Name der Importschnittstelle>Import. \
        <Name der verbundenen Exportschnittstelle>();
}

// verbundene Anbieter der importierten Schnittstellen
// FÜR JEDE ENTHALTENE IMPORTSCHNITTSTELLE:
private <Name des Eigentümers der verbundene Exportschnittstelle>
        <Name der Importschnittstelle>Import;
```

Abbildung 4.13: Schablone zur Erzeugung des Komponentenrumpfes

zwischen Schnittstellen der enthaltenen Komponenten ausgewertet, um in der importierenden Komponente einen Verweis auf die exportierende Komponente zu generieren. Die Verweise werden über Methodenaufrufe an die importierenden Komponenten übergeben. Auf die Konstruktorerzeugung folgt Generierung der Attribute für die enthaltenen Komponenten.

Anschließend erfolgt die Codeerzeugung für die Schnittstellen der Komponente. Jede Schnittstelle wird auf eine Methode abgebildet. Für Exportschnittstellen sind die erzeugten Methodenrümpfe leer, während sie für Importschnittstellen eine Weiterleitung an die exportierende Komponente enthalten. Zusätzlich wird für jede Importschnittstelle eine Methode zur Übergabe

eines Verweises auf die verbundene exportierende Komponente und ein Attribut zum Speichern dieses Verweises erzeugt. In den nachfolgenden Abbildungen 4.14 und 4.15 ist der Code dargestellt, der sich durch Anwendung der Schablonen auf die Komponenten **Waschmaschine** und **Steuerung** aus Abb. 4.9 ergibt.

```
class Waschmaschine() {

    public static void main(String[] args) {
      Waschmaschine mainComponent = new Waschmaschine();
    }

    public Waschmaschine() {
        // instanziiere enthaltene Komponenten
        k0 = new Steuerung();
        k1 = new Tuer();
        k2 = new Wahlschalter();
        k3 = new Pumpe();
        k4 = new EinAusSchalter();
        k5 = new Laugenbehaelter();
        k6 = new Motor();

        // verbinde Schnittstellen
        k0.setBlockieren(k1);
        k0.setProgramm(k2);
        k0.setPumpen(k3);
        k0.setOeffnen(k4);
        k0.setSpannung(k5);
        k1.setOffen(k0);
        k4.setEin(k0);
        k5.setGewicht(k0);
        k6.setDrehzahl(k0;
    }


    // enhaltene Komponenten
    private Steuerung k0;
    private Tuer k1;
    private Wahlschalter k2;
    private Pumpe k3;
    private EinAusSchalter k4;
    private Laugenbehaelter k5;
    private Motor k6;

    // set-Methoden für importierte Schnittstellen
    // Methoden für exportierte Schnittstellen
    // Methoden für importierte Schnittstellen
    // verbundene Anbieter der importierten Schnittstellen
}
```

Abbildung 4.14: Erzeugter Code für die Komponente **Waschmaschine**

```
class Steuerung() {

    public Steuerung() {
        // instanziiere enthaltene Komponenten
        // verbinde Schnittstellen
    }

    // set-Methoden für importierte Schnittstellen
    public setBlockieren(Tuer v) {
        blockierenImport = v;
    }

    public setProgramm(Wahlschalter v) {
        programmImport = v;
    }

    public setPumpen(Pumpe v) {
        pumpenImport = v;
    }

    public setSpannung(Motor v) {
        spannungImport = v;
    }

    // Methoden für exportierte Schnittstellen
    public void offen() {}
    public void ein() {}
    public void gewicht() {}
    public void drehzahl() {}

    // Methoden für importierte Schnittstellen
    private void blockieren() {
        blockierenImport.blockieren();
    }

    private void programm() {
        programmImport.programm();
    }

    private void pumpen() {
        pumpenImport.pumpen();
    }

    // verbundene Anbieter der importierten Schnittstellen
    private Tuer blockierenImport;
    private Wahlschalter programmImport;
    private Pumpe pumpenImport;
    private Laugenbehaelter oeffnenImport;
    private Motor spannungImport;
}
```

Abbildung 4.15: Erzeugter Code für die Komponente **Steuerung**

4.4 Zusammenfassung

In diesem Kapitel wurden die Anforderungen beschrieben, die der Entwicklung des MOmo-Baukastens zugrunde liegen. Anschließend wurden die Anforderungen schrittweise in die Architektur eines Codegenerators umgesetzt, um die zur Implementierung von Modelltransformatoren für MOF-basierte Modellierungssprachen notwendigen Bestandteile des Baukastens zu identifizieren.

Die wesentlichen Anforderungen an die Komponenten des Baukastens wurden wie folgt festgelegt:

- Das Eingabeformat für einen MOmo-Codegenerator ist XMI.
- Der Entwicklungsaufwand für neue Codegeneratoren soll minimiert werden.
- Kleine Änderungen an den zu erzeugenden Artefakten sollen leicht durchzuführen sein.
- Die Übersetzungsgeschwindigkeit soll mit dem Umfang des Modells und der Komplexität der zu erzeugenden Artefakte skalieren.
- Die Codegeneratoren sollen auf standardisierten Werkzeugen und Technologien aufbauen.

Um diese Anforderungen umzusetzen, wurde eine mehrstufige Architektur für einen MOmo-Codegenerator entworfen. Jeder Generator enthält mindestens einen XMI-Reader, einen Kontextgenerator und einen Schablonenausführer. Der Reader kann für jede MOF-basierte Modellierungssprache durch den im Baukasten enthaltenen MOF-Generator erzeugt werden. Dasselbe gilt für die Implementierung des Metamodells einer MOF-basierten Modellierungssprache, die zur Repräsentation der Objekte der zu transformierenden Modelle benötigt wird.

Der Reader baut eine Repräsentation des zu übersetzenden Modells auf, die aus Instanziierungen der generierten Metamodellimplementierung besteht. Die Instanziierungen werden als Metaobjekte bezeichnet. Metaobjekte können durch benutzerdefinierte Module erzeugt, gelöscht oder verändert werden. Diese Mechanismen werden verwendet, um Manipulationen eines Modells vor der Ausführung der Schablonen durchführen zu können. Ein Beispiel für den Einsatz eines benutzerdefinierten Moduls ist die Expansion von Vereinigungsbeziehungen zwischen Paketen.

Nach der Anwendung von benutzerdefinierten Modulen werden die Metaobjekte an den Kontextgenerator weitergereicht, der für jedes Metaobjekt einen Kontext erzeugt. Ein Kontext ist eine bestimmte Repräsentation der Informationen, die ein Metaobjekt enthält. Die Informationen werden so dargestellt, dass sie möglichst einfach durch Schablonen ausgewertet werden können. Die konkrete Ausprägung eines Kontextes ist daher abhängig von der gewählten Implementierung der Schablonen. Jeder Kontext wird an den Schablonenausführer übergeben, der durch Anwendung der Schablonen auf die Kontexte die gewünschten Artefakte erzeugt. Die erzeugten Artefakte können durch benutzerdefinierte Aktionen weiterverarbeitet werden. Dieser Mechanismus kann z. B. verwendet werden, um die Quelltexte einer Programmiersprache in einer bestimmten Art zu formatieren.

Kapitel 5

Meta Object Facility

Die *Meta Object Facility (MOF)* ist eine Modellierungssprache, die von der *Object Management Group (OMG)* standardisiert wird. Die wichtigste Anwendungsdomäne der MOF ist die Definition von Metamodellen. Im Rahmen dieser Arbeit stellt die MOF eine Kernkomponente für den MOmo-Baukastens dar, der in Kapitel 4 vorgestellt wurde. Der Baukasten vereinfacht die Implementierung von Codegeneratoren für alle Modellierungssprachen, deren abstrakte Syntax durch ein MOF-Modell festgelegt wird.

In diesem Kapitel wird die Version 2.0 des MOF-Standards beschrieben, die in [ACC+03] spezifiziert wird. Diese Version ist zur Zeit[1] noch nicht verabschiedet, so dass die endgültige Version möglicherweise einige Veränderungen gegenüber der hier beschriebenen enthält. Trotzdem wird die Version 2.0 der MOF als Grundlage für diese Arbeit verwendet, weil sie grundlegende Veränderungen gegenüber ihren Vorgängerversionen aufweist, die größer sind als die noch zu erwartenden Änderungen zwischen der hier beschriebenen und der endgültigen Version. Insbesondere ist die Version 2.0 der MOF besser an andere Modellierungssprachen, die ebenfalls von der OMG standardisiert werden, angeglichen worden.

Die Angleichung wird erreicht, indem die Version 2.0 der MOF im Unterschied zu den 1.x-Versionen nicht als alleinstehender Standard definiert wird. In den 1.x-Versionen wurde die MOF von Grund auf im Rahmen der MOF-Spezifikation definiert (siehe z. B. [Obj02b]). Die Version 2.0 baut dagegen auf der UML-Infrastrukturbibliothek auf, die die wichtigsten Mechanismen objektorientierter Modellierungssprachen bereitstellt.

Die UML-Infrastrukturbibliothek ist der grundlegende Bestandteil der Version 2.0 der UML, die sich analog zu MOF zur Zeit in der Endphase des Standardisierungsprozesses befindet. Die UML wurde in Infrastruktur und Superstruktur aufgeteilt, wobei die Superstruktur die Elemente der Infrastruktur erweitert und gleichzeitig zur Definition weiterer Modellelemente verwendet. Der derzeitige Stand der UML wird in [UP03a] und [UP03b] beschrieben.

In Abschnitt 5.1 wird die Metadatenarchitektur der OMG beschrieben, um eine Einordnung der MOF-basierten Metamodellierung vornehmen zu können. Anschließend wird in Abschnitt 5.2

[1] Stand: 22.09.2004

die Struktur der MOF vorgestellt, wobei insbesondere auf die Verbindungen zwischen der UML-Infrastruktur und MOF näher eingegangen wird. In Abschnitt 5.3 werden die wichtigsten Elemente der Modellierung mit der MOF beschrieben. Schließlich wird der Inhalt dieses Kapitels in Abschnitt 5.4 zusammengefasst.

5.1 Metadatenarchitektur

Im Allgemeinen kann eine Metadatenarchitektur aus beliebig vielen Schichten bestehen. In der Regel werden allerdings nur vier Schichten verwendet, weil weitere Schichten keine zusätzlichen Informationen liefern würden. Ein Beispiel für eine Metadatenarchitektur mit vier Schichten ist die OMG-Metadatenarchitektur, die in Abb. 5.1 dargestellt ist. Gegenüber dem allgemeinen Metamodellbegriff, der in Kapitel 2 auf Seite 15 vorgestellt wurde, sind die Ebenen der OMG-Metadatenarchitektur um eine Ebene nach unten verschoben, so dass sich Modelle eines Systems auf Ebene 1 anstatt Ebene 0 befinden.

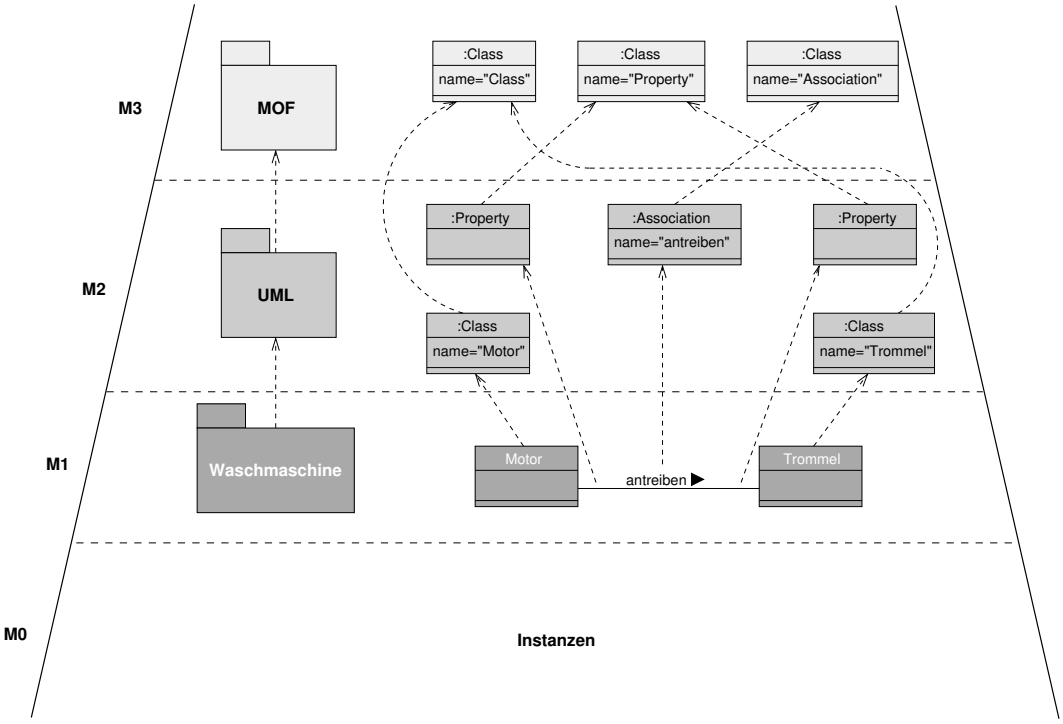


Abbildung 5.1: Die OMG-Metadatenarchitektur

Die unterste Schicht der OMG-Metadatenarchitektur ist die Informationsschicht, die in OMG-Spezifikationen häufig auch als M0-Schicht bezeichnet wird. In dieser Schicht befinden sich die Instanzen von Klassen, die Anwenderdaten modellieren. In Abb. 5.1 befinden sich in der Informationsschicht konkrete Ausprägungen der Klassen **Trommel** und **Motor**, die den Zustand des Antriebs einer Waschmaschine repräsentieren.

Die nächsthöhere Schicht ist die Modellschicht oder M1-Schicht. In dieser Schicht befinden sich anwendungsspezifische Modelle. In Abb. 5.1 ist in der Modellschicht ein Ausschnitt eines Modells einer Waschmaschine dargestellt. Der Modellausschnitt zeigt zwei Klassen **Trommel** und **Motor**, die über die Assoziation **antreiben** miteinander verbunden sind.

Die dritte Schicht der OMG-Metadatenarchitektur ist die Metamodellschicht oder M2-Schicht. Diese Schicht enthält Beschreibungen der Elemente, die verwendet werden können, um die Modelle der Modellschicht zu beschreiben. Abb. 5.1 zeigt, dass das UML-Metamodell[2] in dieser Schicht einzuordnen ist. Das UML-Metamodell beschreibt unter anderem den Aufbau von Klassen und Assoziationen durch die Klassen **Class** und **Association**, die zur Erstellung von Anwendermodellen verwendet werden können. Die Abbildung zeigt, wie die Elemente der Modellschicht durch Instanziierung von Elementen der Metamodellschicht erzeugt werden.

Die oberste Schicht der OMG-Metadatenarchitektur ist die Meta-Metamodellschicht. Das Verhältnis zwischen der Meta-Metamodellschicht und der Metamodellschicht ist analog zum Verhältnis zwischen der Metamodellschicht und der Modellschicht. Die Elemente der Metamodellschicht sind Instanzen der Elemente der Meta-Metamodellschicht. In Abb. 5.1 ist das MOF-Metamodell als ein Beispiel für ein Meta-Metamodell dargestellt. Die Elemente der MOF werden instanziiert, um die Elemente des UML-Metamodells zu erzeugen. Im dargestellten Beispiel werden je zwei Instanzen der MOF-Klassen **Class** und **Property** und eine Instanz der MOF-Klasse **Association** erzeugt.

Häufig wird das Modell der Meta-Metamodellebene als „festverdrahteten“ bezeichnet, um zu kennzeichnen, dass es eine Art Startpunkt markiert. Aus diesem Grund sind Metadatenarchitekturen meist auf die beschriebenen vier Ebenen beschränkt. Wenn weitere Ebenen eingefügt werden, wird das Problem lediglich weiter nach oben verschoben. In der Praxis ist das Modell auf der obersten Ebene einer Metadatenarchitektur häufig „durch sich selbst“beschrieben. Dies bedeutet, dass die Modellelemente, die zur Beschreibung des Modells verwendet werden, durch dasselbe Modell festgelegt werden. Daraus folgt, dass das Modell prinzipiell beliebig über die Ebenen der Metadatenarchitektur verschoben werden kann.

Innerhalb der OMG-Metadatenarchitektur übernimmt das MOF-Metamodell auf der Meta-Metamodellschicht die Rolle des „festverdrahteten“ Meta-Metamodells und definiert den Startpunkt der (Meta-) Modellierung. Seine Instanzen, die MOF-Modelle, befinden sich auf der Metamodellschicht und definieren die Metamodelle anderer Modellierungssprachen wie beispielsweise UML und CWM [Obj03a]. Solche Modellierungssprachen unterscheiden sich von MOF im Wesentlichen durch ihre Anwendungsdomäne. Während MOF speziell auf die Definition von Metamodellen zugeschnitten ist, ist die UML eine generell einsetzbare Sprache zur Modellierung von Anwendungssystemen und CWM eine Sprache zur Modellierung betrieblicher Informationssysteme.

[2] bezogen auf UML 2.0 ist hier das Metamodell der UML-Superstruktur gemeint

5.2 Struktur

Die 1.x-Versionen der MOF- und UML-Spezifikationen enthalten viele ähnliche Modellierungselemente, die sich aber in einigen Details unterscheiden. Diese leicht unterschiedlichen Definitionen führen häufig zu Missverständnissen. Daher ist eines der wesentlichen Ziele bei der Spezifikation der Version 2.0 von MOF und UML, eine Angleichung der Sprachdefinitionen zu erreichen, um den Umgang mit den Sprachen zu erleichtern.

Um die Angleichung der Sprachdefinitionen zu erreichen, müssen die Metamodelle von MOF und UML in den Bereichen übereinstimmen, die beide Sprachen enthalten. Aus diesem Grund wurde ein gemeinsamer Teil aus den Sprachen herausgelöst und in Form der *UML-Infrastruktur* spezifiziert. Diese bildet damit die gemeinsame Basis für die Spezifikation zukünftiger Modellierungssprachen, die von der OMG standardisiert werden. Abb. 5.2 zeigt die verschiedenen Formen der Abhängigkeiten zwischen MOF und UML.

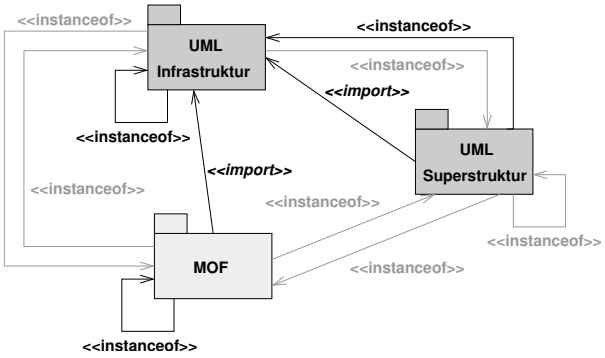


Abbildung 5.2: Beziehungen zwischen MOF und UML

Aufgrund der aufeinander aufbauenden Definitionen kann jede Sprache sowohl durch die jeweils anderen als auch durch sich selbst beschrieben werden. Die dunkel dargestellten Beziehungen sind diejenigen, die von der OMG zur Beschreibung der Sprachen verwendet werden. Da alle zur Beschreibung der UML notwendigen Beschreibungsmittel aufgrund der **«import»**-Beziehung zwischen UML-Infrastruktur und MOF auch in MOF enthalten sind, lässt sich jedes Modell, das durch die UML-Infrastruktur beschrieben werden kann, auch durch MOF beschreiben. Die MOF-Spezifikation verwendet einen Mechanismus zur Verknüpfung von Paketen (siehe Abschnitt 5.3.4), der nur in MOF zur Verfügung steht. Aus diesem Grund ist MOF im Unterschied zu UML-Infrastruktur und -Superstruktur in erster Linie eine MOF-Instanz. Da der entsprechende Mechanismus durch in der UML-Infrastruktur enthaltene Mechanismen nachgebildet werden kann, könnte die MOF allerdings auch durch ein UML-Modell beschrieben werden.

Neben dem Genannten stellt die MOF einige weitere, meist implementierungsnahe Mechanismen zum Umgang mit Modellen zur Verfügung und bildet die Basis von Abbildungen von Sprachen auf andere Technologien. Ein Beispiel ist der XMI-Standard zur Repräsentation von Modellen in XML. XMI und XML werden in den Kapiteln 6 und 7 beschrieben. Wegen der zusätzlichen Eigenschaften ist die MOF eine bessere Basis zur Implementierung von Werkzeugen

als die UML-Infrastruktur. Aus diesem Grund wird in diesem Kapitel die MOF und nicht die UML-Infrastruktur beschrieben.

Die UML-Infrastruktur wird durch die *UML-Infrastrukturbibliothek* bereitgestellt und kann durch Mechanismen, die Teil der UML-Infrastruktur sind, in andere Metamodelle integriert werden. Um eine möglichst flexible Basis für die Definition von Metamodellen bereitzustellen, enthält die UML-Infrastruktur mehrere Schichten, die unterschiedlich komplexe Modellierungselemente enthalten.

Die unterste Schicht der UML-Infrastruktur enthält vorwiegend abstrakte Metaklassen. Diese Schicht wird in der UML-Infrastrukturbibliothek durch das Paket **Abstractions** bereitgestellt, das 20 Pakete enthält, die einzelne Aspekte objektorientierter Sprachen modellieren. Jedes Paket enthält nur wenige zusammengehörige Modellierungselemente, so dass feingranulares Einbinden einzelner Aspekte in eigene Metamodelle ermöglicht wird.

Die nächsthöhere Schicht der UML-Infrastruktur enthält die Definition einer „minimalen klassenbasierten Modellierungssprache“ [UP03a], Seite 75, die als Basis zur Definition komplexerer objektorientierter Modellierungssprachen verwendet werden kann. Diese Schicht wird durch das Paket **Basic** der UML-Infrastrukturbibliothek repräsentiert. Das Paket **Basic** enthält überwiegend konkrete Klassen, die auf den abstrakten Klassen des Paketes **Abstractions** aufbauen.

Die höchste Schicht der UML-Infrastruktur beinhaltet erweiterte Definitionen der Modellierungselemente der **Basic**-Schicht und spezifiziert weitere Modellierungselemente. Diese Schicht repräsentiert den gemeinsamen Kern der Modellierungssprachen MOF und UML-Superstrukur und wird in der UML-Infrastrukturbibliothek durch das Paket **Constructs** bereitgestellt.

In Abb. 5.3 ist die Struktur der UML-Infrastrukturbibliothek im Überblick dargestellt. Neben den bereits beschriebenen Paketen, die alle innerhalb des Paketes **Core** definiert sind, enthält die UML-Infrastrukturbibliothek noch die Pakete **PrimitiveTypes** und **Profiles**. Das Paket **PrimitiveTypes** ist ebenfalls innerhalb des Paketes **Core** definiert und stellt einige Grunddatentypen zur Verfügung, die zur Definition der Modellierungselemente in den anderen Paketen der UML-Infrastrukturbibliothek benötigt werden. Das Paket **Profiles** stellt Mechanismen zur Verfügung, die zur Anpassung eines Metamodells an eine bestimmte Anwendung benötigt werden.

Die MOF-Spezifikation definiert zwei Modellierungssprachen, die auf den Paketen **Basic** bzw. **Constructs** der UML-Infrastrukturbibliothek aufbauen. *Essential MOF* (EMOF) baut auf den Elementen des Paketes **Basic** auf und definiert eine Modellierungssprache, die sich an den Eigenschaften objektorientierter Programmiersprachen und XML [XML00] orientiert. Dementsprechend ist EMOF leicht zu implementieren und stellt nur die wesentlichen Modellierungselemente zur Definition einfacher Metamodelle bereit. EMOF basiert nicht nur auf den Paketen der UML-Infrastrukturbibliothek, sondern auch auf den Paketen **Extension**, **Identity** und **Reflection**, die innerhalb der MOF-Spezifikation definiert werden.

Das Paket **Reflection** erweitert ein Modell um die Fähigkeit, sich selbst beschreiben zu können. Dies hat den Vorteil, dass Implementierungen von Metamodellen ohne Kenntnis der einzelnen Modellelemente entwickelt werden können, so dass beispielsweise Schnittstellen auf

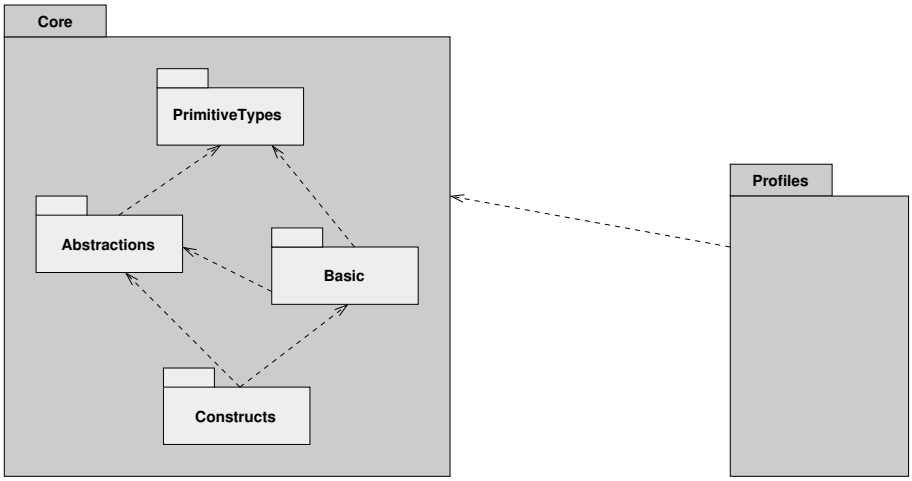


Abbildung 5.3: Struktur der UML-Infrastrukturbibliothek

Basis eines Metamodells implementiert werden können, die für jede Instanz des Metamodells funktionieren. Im Paket **Identity** wird eine implementierungsunabhängige Möglichkeit angegeben, Modellelemente eindeutig zu identifizieren, und das Paket **Extension** enthält ein Modellierungselement, das die Zuordnung herstellerspezifischer Informationen zu einem Modellierungselement ermöglicht.

Complete MOF (CMOF) baut dagegen auf den Elementen des Paketes **Constructs** der UML-Infrastrukturbibliothek auf und definiert eine Modellierungssprache zur Spezifikation komplexer Metamodelle. In Abb. 5.4 sind die Beziehungen zwischen den Paketen der UML-Infrastrukturbibliothek und den Paketen der MOF dargestellt. Die dargestellten Stereotypen **«merge»** und **«combine»** geben an, wie die Elemente der verbundenen Pakete kombiniert werden sollen. Ihre genaue Bedeutung wird in Abschnitt 5.3.4.3 beschrieben.

Im folgenden Abschnitt werden die Modellierungselemente der MOF vorgestellt. Da im Rahmen dieser Arbeit eine vollständige Unterstützung der MOF entwickelt werden soll, beziehen sich die Beschreibungen auf die Eigenschaften der Modellierungselemente der CMOF. Wenn Modellierungselemente in EMOF fehlen oder nur eingeschränkt verfügbar sind, wird dies angegeben.

5.3 Modellierungselemente

Die MOF-Spezifikation definiert zwei Modellierungssprachen, die auf die Anwendungsdomäne *Metamodellierung* zugeschnitten sind. Die wichtigsten Modellierungselemente beider Sprachen sind Datentypen, Klassen und Assoziationen sowie Pakete. Datentypen sind eine allgemeine Grundlage von Programmier- und Modellierungssprachen, unabhängig vom Paradigma der Sprache. In objektorientierten Sprachen werden Datentypen insbesondere zur Definition der Struktur von Klassen verwendet. Klassen sind der wichtigste Baustein objektorientierter Sprachen. Eine Klasse beschreibt den Aufbau ihrer Instanzen und modelliert dadurch den Auf-

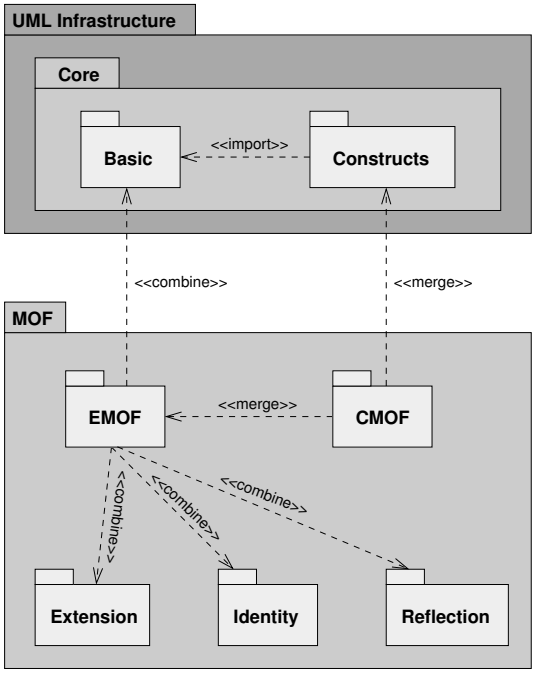


Abbildung 5.4: Beziehungen zwischen Paketen der UML-Infrastruktur und Paketen der MOF

bau eines objektorientierten Systems. Eng verknüpft mit den Klassen sind die Assoziationen, die Beziehungen zwischen Klassen ausdrücken. Alle bisher genannten Modellierungselemente müssen sich in MOF-Modellen direkt oder indirekt in Paketen befinden. Pakete sind der Mechanismus zur Strukturierung eines Modelles.

Im Weiteren werden die Eigenschaften der Grundelemente der MOF-basierten Metamodellierung vorgestellt. Wie in Abschnitt 5.2 dargestellt, definiert die MOF-Spezifikation die Modellierungssprachen EMOF und CMOF. Ziel dieser Arbeit ist die Unterstützung der MOF-basierten Metamodellierung und insbesondere die Werkzeugunterstützung bei der Implementierung domänenspezifischer Modellierungssprachen und -werkzeuge. Nach Verabschiedung der Version 2.0 der UML werden domänenspezifische Modellierungssprachen häufiger als bisher auf der UML aufbauen, weil die Modularität der Sprache deutlich verbessert wurde. Das UML-Metamodell verwendet die Modellierungselemente der UML-Infrastruktur bzw. CMOF. Aus diesem Grund werden die Modellelemente der CMOF vorgestellt. Zusätzlich wird angegeben, welche Eigenschaften den Modellierungselementen in EMOF fehlen.

5.3.1 Klassen

Objektorientiert strukturierte Softwaresysteme bestehen zur Laufzeit aus einer Menge von Objekten, die die Anwenderinformationen enthalten. Die Eigenschaften von Objekten werden in objektorientierten Programmier- und Modellierungssprachen durch Klassen festgelegt. Der *Zustand* eines Objektes wird durch die *Attribute* der Klasse vorgegeben, deren Instanz das Objekt

ist. Analog dazu wid das *Verhalten* eines Objektes durch die *Operationen* der Klasse bestimmt, deren Instanz das Objekt ist.

Neben der direkten Spezifikation der Struktur und des Verhaltens ihrer Instanzen kann eine Klasse die Eigenschaften eines Objektes auch indirekt festlegen. Objektorientierte Programmier- und Modellierungssprachen enthalten dazu den Mechanismus der *Vererbung*. Eine Klasse kann die Eigenschaften anderer Klassen erben. Es gibt Sprachen, die nur das direkte Erben von einer Klasse erlauben (z. B. Java, Smalltalk), und Klassen, die auch das direkte Erben der Eigenschaften mehrerer Klassen ermöglichen (Mehrfachvererbung, z. B. C++, MOF, UML).

Abb. 5.5 zeigt den Ausschnitt des Metamodells der UML-Infrastruktur, der die Eigenschaften einer Klasse festlegt. In der Spezifikation der CMOF wird keines der Elemente dieses Metamodellausschnitts verändert.

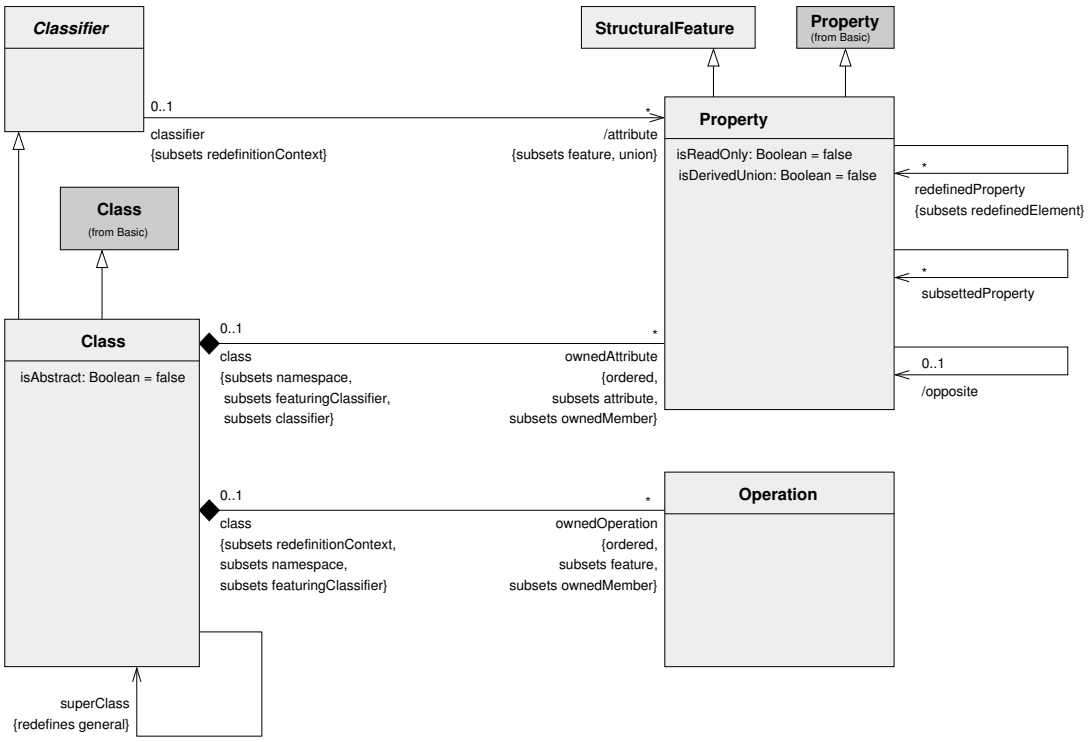


Abbildung 5.5: Klassendefinition im Paket **Constructs** der UML-Infrastruktur

Klassen werden innerhalb des Paketes **Constructs** durch die Metaklasse **Class** repräsentiert, die die Eigenschaften der abstrakten Klasse **Classifier** und der konkreten Klasse **Class** aus dem Paket **Basic** erbt. Die Vererbungen sind durch die Pfeile mit geschlossener Spitze dargestellt. Außerdem enthält eine Klasse Mengen von Attributen und Operationen. Attribute werden durch die Metaklasse **Property** und Operationen durch die Metaklasse **Operation** repräsentiert. Die Kompositionsbeziehungen werden durch eine Linie zwischen Container und Komponente dargestellt, wobei der Container durch eine gefüllte Raute gekennzeichnet ist. Jede Operation enthält wiederum eine Menge von Parametern, die durch die Metaklasse **Parameter** dargestellt

werden. Weiterhin zeigt das Diagramm einige Teilmengenbeziehungen zwischen Attributen, die durch **subsets** ausgedrückt werden. Die Bedeutung von Teilmengenbeziehungen und verwandten Beziehungen zwischen Attributen wird in Abschnitt 5.3.1.1 beschrieben.

Die Notation einer Klasse erfolgt durch ein Rechteck, das in mehrere Abschnitte unterteilt wird. Der oberste Abschnitt enthält den Namen der Klasse. Die darunter angeordneten Abschnitte enthalten die Spezifikationen der Attribute und Operationen der Klasse. In Abb. 5.6 ist die Klasse **Motor** dargestellt, die die Attribute **umdrehungen** und **drehrichtung** und die Operationen **setzeUmdrehungen** und **setzeDrehrichtung** enthält.

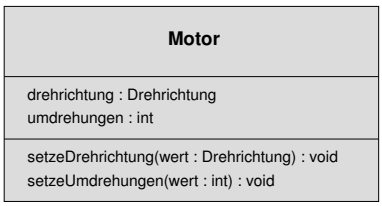


Abbildung 5.6: Notation einer MOF-Klasse

Klassen können als *abstrakt* gekennzeichnet werden. Eine abstrakte Klasse kann nicht instanziiert werden und spezifiziert im Allgemeinen nur Teile der Struktur oder des Verhaltens von Objekten. Andere Klassen erben diese Teilspezifikationen und vervollständigen sie. Abstrakte Klassen werden durch die Angabe von «abstract» oberhalb des Klassennamens oder einen kursiv geschriebenen Klassennamen notiert. Innerhalb eines Modells werden abstrakte Klassen durch Instanzen der Metaklasse **Class** repräsentiert, deren Attribut **isAbstract** mit dem Wert **„true“** belegt ist.

5.3.1.1 Attribute

Die Attribute einer Klasse werden im Wesentlichen durch einen Namen, einen Typ und eine Multiplizitätsangabe spezifiziert. Gemäß der Spezifikation [UP03a] sind sowohl der Name als auch der Typ optional. Was unter einem unbenannten und/oder untypisierten Attribut zu verstehen ist, wird dort nicht weiter erläutert.

Der Name eines Attributes muss innerhalb der Klasse eindeutig sein. Dies gilt insbesondere auch dann, wenn Attribute von anderen Klassen geerbt werden. In diesem Fall dürfen die Klassen, die geerbt werden, weder Namenskonflikte zueinander noch zu der erbenden Klasse aufweisen. Dies schließt auch das mehrfache Erben derselben Klasse aus.[3]

Was genau ein Namenskonflikt ist, wird in der Spezifikation nicht endgültig festgelegt. Es wird lediglich die Grundregel angegeben, dass zwei Modellelemente dann keinen Namenskonflikt verursachen, wenn sie verschiedene Namen oder verschiedene Typen besitzen. Verschiedene Typen bedeutet, dass die Typen zweier Attribute nicht identisch sind und außerdem keine Vererbungsbeziehung zwischen ihnen besteht. Je nach Art des Modellelementes kann es allerdings

[3] Eine Ausnahme ist die sogenannte „Diamantenregel“ (engl. diamond-rule), die das mehrfache Erben derselben Klasse erlaubt, wenn diese nicht direkt sondern indirekt geerbt wird.

auch andere Regeln zur eindeutigen Identifikation eines Modellelementes geben. Operationen können beispielsweise mit Hilfe ihrer vollständigen Signaturen unterschieden werden. Die Spezifikation legt leider nicht eindeutig fest, wie die Signaturen gebildet werden müssen. Dies führt zu Problemen, wenn MOF-Modelle implementiert werden müssen, weil Programmiersprachen in der Regel eine bestimmte Art der Signaturenbildung voraussetzen.

Der Typ eines Attributes kann entweder eine MOF-Klasse oder ein MOF-Datentyp sein. Attribute können also sowohl Beziehungen zwischen Klassen als auch Beziehungen zwischen Klassen und Datentypen repräsentieren. Beziehungen sind hier als gerichtete Assoziationen zu verstehen. Falls der Typ eines Attributes eine Klasse ist, können auch bidirektionale Beziehungen ausgedrückt werden, indem je ein Attribut des Typs als „gegenüberliegend“ angegeben wird. Die Möglichkeit, Beziehungen zwischen Klassen und Datentypen auszudrücken, unterscheidet Attribute von Assoziationen, die ausschließlich Beziehungen zwischen Klassen darstellen können.

Durch Angabe der Multiplizität können Attribute als optional, ein- oder mehrwertig gekennzeichnet werden. Multiplizitäten werden durch eine obere und eine untere Schranke festgelegt. Die Schranken geben an, wieviele Werte die Instanz eines Attributes minimal speichern können muss und maximal speichern kann. Die Instanzen optionaler Attribute können maximal einen Wert speichern, während die Instanzen einwertiger Attribute genau einen Wert speichern. Mehrwertige Attribute erlauben ihren Instanzen die Anzahl von Werten zu speichern, die durch die Schranken vorgegeben wird. Für mehrwertige Attribute kann in einer Multiplizitätsspezifikation zusätzlich angegeben werden, ob ein Wert mehrfach in dem Attribut gespeichert werden kann, und ob die Werte eine Ordnung aufweisen. Diese Angaben sind bei optionalen und einwertigen Attributen bedeutungslos.

Für jedes Attribut kann angegeben werden, ob es eine Kompositionsbeziehung repräsentiert. In diesem Fall ist das Attribut ein Teil der Klasse, d. h. das Attribut wird erst nach der Erzeugung eines Objektes instanziiert und vor dem Löschen des Objektes entfernt. Andernfalls ist der Lebenszyklus des Attributes unabhängig vom Lebenszyklus des Objektes. Meist definieren Attribute, deren Typ ein Datentyp ist, implizit eine Kompositionsbeziehung zwischen der Klasse und dem Datentyp. Im Gegensatz zur Spezifikation von MOF 1.x muss dies in der MOF 2.0 explizit angegeben werden.

Attribute können von anderen Attributen abgeleitet und schreibgeschützt werden. Der Wert eines abgeleiteten Attributes wird zur Laufzeit des Systems nicht direkt gespeichert, sondern aus den aktuellen Werten anderer Attribute berechnet. Ein schreibgeschütztes Attribut wird bei der Erzeugung mit einem Wert belegt, der über die gesamte Lebensdauer des Attributes nicht verändert werden kann. In der Regel sind abgeleitete Attribute auch schreibgeschützt. Ist dies jedoch nicht der Fall, muss die Implementierung sicherstellen, dass bei der Veränderung des abgeleiteten Attributes ein konsistenter Zustand erhalten bleibt.

Außerdem kann die Sichtbarkeit eines Attributes für andere Modellelemente eingeschränkt werden. Die UML-Infrastruktur unterscheidet die Sichtbarkeiten „public“ und „private“ . Die Sichtbarkeit „public“ bedeutet, dass alle Modellelemente, die auf den Inhalt des umgebenden Namensraumes zugreifen können, auf das Modellelement zugreifen können. Modellelemente,

deren Sichtbarkeit mit „private“ angegeben ist, sind dagegen nur innerhalb des umgebenden Namensraumes sichtbar.

Alle bisher genannten Eigenschaften sind sowohl in der **Basic**- als auch in der **Constructs**-Schicht definiert. Auf der **Constructs**-Schicht werden den Attributspezifikationen drei weitere Eigenschaften hinzugefügt, die zur Definition von Redefinitions-, Teilmengen- und Vereinigungsbeziehungen verwendet werden können.

Wenn zwei Attribute typkompatibel sind, kann eine Redefinitionsbeziehung zwischen ihnen spezifiziert werden. Die Typkompatibilität verlangt, dass die Typen der beiden Attribute zueinander passen. Das redefinierende Attribut muss entweder denselben Typ besitzen wie das redefinierte Attribut, oder der Typ des redefinierenden Attributes muss eine Spezialisierung des Typs des redefinierten Attributes sein. Die Redefinition wird durch Angabe von **redefines** an einem Attribut notiert und bedeutet eine Einschränkung der möglichen Instanzen des redefinierten Attributes auf Instanzen, die auch Instanz des redefinierenden Attributes sein können. Redefinitionsbeziehungen werden im MOF-Metamodell durch eine Assoziation ausgedrückt, die ein Assoziationsende **redefinedProperty** enthält und die zwei Instanzen der Metaklasse **Property** miteinander verbindet. Redefinitionsbeziehungen lassen sich verwenden, um innerhalb einer Klasse den Typ eines geerbten Attributes geeignet einzuschränken. Vererbungen werden in Abschnitt 5.3.1.3 beschrieben.

Zwischen typkonformen Attributen kann auch eine Teilmengenbeziehung definiert werden, so dass Instanzen eines als Teilmenge deklarierten Attributes ebenso als Instanzen des als Obermenge fungierenden Attributes geführt werden. Wie oben bereits beschrieben, werden Teilmengenbeziehungen durch **subsets** angegeben. Im Metamodell werden diese Beziehungen durch die Assoziation mit dem Assoziationsende **subsettedProperty** ausgedrückt, die zwei Instanzen der Metaklasse **Property** miteinander verbindet.

Neben Redefinitions- und Teilmengenbeziehungen können Vereinigungsmengenbeziehungen zwischen einer Menge von Attributen spezifiziert werden. Ein vereinigendes Attribut wird durch **union** gekennzeichnet. Die Belegung eines Attributes ist die Menge aller Belegungen der Attribute, die als Teilmenge des vereinigenden Attributes definiert sind. Im Metamodell wird ein vereinigendes Attribut durch Setzen des Metaattributes **isDerivedUnion** auf **„true“** gekennzeichnet.

5.3.1.2 Operationen

Die UML-Infrastruktur ist als allgemein verwendbare Basis zur Definition von Modellierungssprachen konzipiert worden. Obwohl sie in erster Linie als Basis der Spezifizierung von Metamodellen genutzt wird und in Metamodellen der Schwerpunkt auf der Modellierung von Strukturen liegt, enthält die UML-Infrastruktur das Modellierungselement *Operation* zur Spezifikation des Verhaltens von Objekten. Genau genommen wird allerdings keine Möglichkeit zur Spezifikation des Verhaltens angeboten, sondern nur zur Spezifikation eines Auslösemechanismus' für ein bestimmtes Verhalten.

Im Gegensatz zu Programmiersprachen und anderen auf der UML-Infrastruktur basierenden Modellierungssprachen konzentriert sich MOF auf die Definition der Struktur eines Informa-

tionsmodells. Konsequenterweise enthält MOF daher keinerlei Sprachmittel zur Spezifikation von Objektverhalten. Aus diesem Grund kann Verhalten in MOF nur informell modelliert werden. Die Operationen dienen lediglich als Platzhalter, über die ein bestimmtes Verhalten aufgerufen werden kann. Die tatsächliche Spezifikation des Verhaltens erfolgt erst in der Implementierung, bzw. durch eine textuelle Spezifikation des Operationsrumpfes innerhalb eines Modells.

Eine Operation spezifiziert den Namen, den Typ, die Parameter und zusätzliche Bedingungen für den Aufruf eines bestimmten Verhaltens. Genau wie die Attribute einer Klasse müssen auch die Operationen eindeutig identifizierbar sein. Als Kennzeichen für die Identifikation von Operationen wird in der Spezifikation der UML-Infrastruktur die Signatur einer Operation vorgeschlagen. Signaturen werden aus dem Typ der Operation (Typ des Rückgabewertes) und den Typen ihrer Parameter gebildet. Dies entspricht den Regelungen, die objektorientierte Programmiersprachen zur Identifikation von Methoden vorsehen.

Die Parameter einer Operation sind den Attributen einer Klasse in vielerlei Hinsicht ähnlich. Sie besitzen ebenfalls einen eindeutigen Namen, einen Typ und eine Multiplizitätsangabe. Im Unterschied zum Attribut ist der Namensraum, in dem die Eindeutigkeit des Namens gelten muss, allerdings nicht durch die Klasse, sondern durch die Operation definiert. Aus diesem Grund definiert ein Parameter auch *keine* Beziehungen zwischen Klassen bzw. Klassen und Datentypen. Im Unterschied zur Parameterspezifikation in MOF 1.x enthält eine Parameterspezifikation keine Richtungsangabe mehr, um die Richtung der Parameterübergabe zwischen Aufrufer und Aufrufendem festzulegen. Parameter werden ausschließlich vom aufrufenden Kontext an die Operation übergeben. Soll ein Parameter trotzdem verwendet werden, um Werte von der Operation an den aufrufenden Kontext zu liefern, müssen Hilfsklassen definiert werden. Dies entspricht der Semantik von Parametern in der Abbildung der CORBA-IDL [Obj02a] auf Java.

Obwohl Operationen in erster Linie eine Syntax für den Aufruf eines bestimmten Verhaltens festlegen, können auch Eigenschaften über die Wirkung des Aufrufs einer Operation angegeben werden. Dazu können drei Arten von Bedingungen definiert werden, die bei der Ausführung einer Operation eingehalten werden müssen. Vorbedingungen geben an, unter welchen Umständen eine Operation aufgerufen werden darf. Nachbedingungen geben an, welche Bedingungen nach einer erfolgreichen Ausführung einer Operation gelten müssen. Wenn die Operation eine Abfrage ist, kann eine Bedingung für den Rumpf der Operation angegeben werden. Solche Bedingungen definieren Einschränkungen des Wertes, den der Rückgabewert bei einer erfolgreichen Ausführung der Operation annehmen darf.

Vor- und Nachbedingungen sowie Bedingungen, die für den Rumpf einer Operation angegeben werden können, unterscheiden sich bei Redefinitionen von Operationen. Während die Rumpfbedingungen in Spezialisierungen einer Operation überschrieben werden dürfen, gelten Vor- und Nachbedingungen immer auch für Spezialisierungen einer Operation. Es dürfen lediglich neue Vor- und Nachbedingungen hinzugefügt werden. Wird eine Operation in einer spezialisierten Klasse überschrieben, dürfen neben den genannten Veränderungen der Vor-, Nach- und Rumpfbedingungen die Typen der formalen Parameter und des Rückgabewertes spezialisiert werden.

5.3.1.3 Vererbung

Vererbung ist ein wichtiger Mechanismus, der objektorientierte Programmier- und Modellierungssprachen von Sprachen unterscheidet, die auf der Grundlage eines anderen Paradigmas definiert sind. Wenn eine Klasse von anderen Klassen erbt, geht sie mit diesen eine *„ist ein“*-Beziehung[4] ein. Die Instanzen der erbenden Klasse können überall anstatt der Instanzen der vererbenden Klassen verwendet werden, weil sie deren Eigenschaften durch die „ist ein“-Beziehung erben.

Klassen können in MOF die Eigenschaften beliebig vieler anderer Klassen erben. Eine Vererbungsbeziehung wird durch einen Pfeil mit durchgehender Linie und geschlossener Pfeilspitze von der erbenden Klasse (z. B. **Wankelmotor**) zur vererbenden Klasse (z. B. **Motor**) gekennzeichnet. Dies ist in Abb. 5.7 dargestellt.

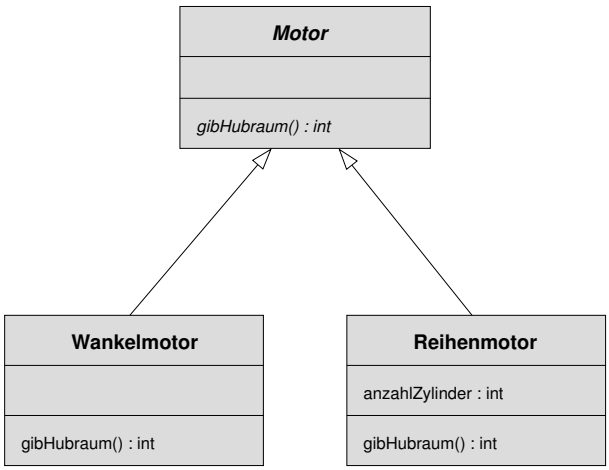


Abbildung 5.7: Notation einer Vererbung

In der MOF-Spezifikation wird das Vererben von Eigenschaften als *Generalisierung* bezeichnet. Wenn die Klassen **Wankelmotor** und **Reihenmotor** die Eigenschaften einer Klasse **Motor** erben, dann generalisiert die Klasse **Motor** die Klassen **Wankelmotor** und **Reihenmotor**. Allgemein formuliert generalisiert eine Klasse alle Klassen, die von ihr abgeleitet werden.

Im Unterschied zu den Vorgängerversionen dürfen in MOF 2.0 Eigenschaften einer geerbten Klasse überschrieben werden. Durch Überschreiben eines Elementes im Rahmen einer Vererbung soll die Semantik des überschriebenen Elementes nicht verändert werden, damit die „ist ein“-Beziehung gewährleistet bleibt. In der Regel werden in den erbenden Klassen zusätzliche Einschränkungen definiert oder das Verhalten von Objekten an veränderte Strukturen angepasst. Dies entspricht der Semantik des Überschreibens von Methoden in objektorientierten Programmiersprachen. Das Überschreiben von Eigenschaften wird insbesondere bei der Vereinigung von Paketen (siehe Seite 86) genutzt. Der Erhalt der Semantik bei Redefinitionen wird in der MOF-Spezifikation nicht festgelegt. Diese Aufgabe fällt daher dem Modellierer zu.

[4] Dies gilt allerdings nur dann, wenn keine Einschränkungen der erbenden Klasse durch *redefines* oder *subsets* definiert worden sind.

5.3.2 Assoziationen

Beziehungen zwischen den Instanzen von Klassen lassen sich durch Assoziationen beschreiben. Jede Assoziation besitzt mindestens zwei Assoziationsenden, die verschiedene Rollen innerhalb der Assoziation beschreiben. Die Assoziationsenden besitzen einen Typ, der die Klasse angibt, die die jeweilige Rolle ausfüllt. Es ist erlaubt, dass mehrere Assoziationsenden einer Assoziation denselben Typ haben.

In der UML-Infrastruktur werden Assoziationen erst auf der Schicht definiert, die durch das Paket **Constructs** festgelegt wird. In den tiefer liegenden Schichten **Abstractions** und **Basic** sind keine Assoziationen verfügbar. Dies bedeutet auch, dass EMOF ebenfalls keine Assoziationen bereitstellt, sondern diese nur in CMOF verwendet werden können. Abb. 5.8 zeigt die Repräsentation von Assoziationen im Metamodell der UML-Infrastruktur.

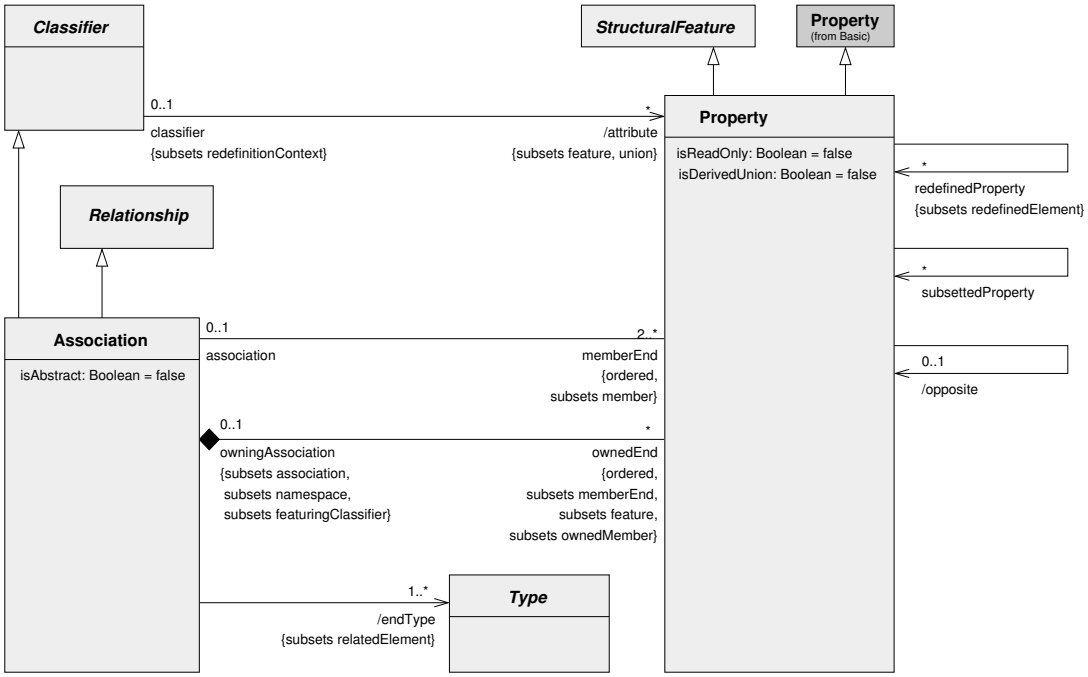


Abbildung 5.8: Repräsentation von Assoziationen im Metamodell der UML-Infrastruktur

Assoziationen werden im MOF-Metamodell durch die Klassen **Association** und **Property** repräsentiert. Die Instanzen der Klasse **Association** modellieren die Assoziation, während die Instanzen der Klasse **Property** die Assoziationsenden darstellen. Die Assoziationen zwischen den Klassen **Association** und **Property** bedeuten, dass jeder Assoziation mindestens zwei Assoziationsenden zugeordnet sind und eine beliebige Anzahl davon zu der Assoziation in einer Kompositionsbeziehung steht. Jedes **Property**-Objekt, das zu einem **Association**-Objekt in einer Kompositionsbeziehung steht, repräsentiert ein nicht-navigierbares Assoziationsende. Navigierbare Assoziationsenden werden durch **Property**-Objekte dargestellt, die in einer Kompositionsbeziehung zu einem **Class**-Objekt stehen.

Im Unterschied zur Modellierung von Beziehungen zwischen Klassen durch Attribute, werden die Beziehungen explizit repräsentiert. Wird eine Beziehung zwischen zwei Klassen durch Attribute modelliert, existieren im MOF-Modell lediglich die beiden **Property**-Objekte, die die Attribute beider Klassen repräsentieren und Verweise aufeinander enthalten. Im Unterschied dazu enthält das Modell eine Instanz der Klasse **Association**, die die Beziehung explizit repräsentiert, und die **Property**-Objekte enthalten jeweils einen Verweis auf das **Association**-Objekt.

Durch die gemeinsame Repräsentation von Attributen und Assoziationsenden im MOF-Metamodell gelten die meisten der auf Seite 73 beschriebenen Eigenschaften von Attributen auch für Assoziationsenden. Insbesondere kann auch für jedes Assoziationsende eine Multiplizität angegeben werden. Diese begrenzt die Anzahl der Objekte an jedem Ende der Assoziation auf ein Intervall, das durch Angabe einer unteren und einer oberen Grenze spezifiziert wird. Dieses Intervall wird in der Nähe des Assoziationsendes in der Form **„<untereGrenze>..<obereGrenze>“** notiert.

Für jedes Assoziationsende kann spezifiziert werden, ob es mehrere Instanzen geben darf, die auf dieselbe Instanz einer Klasse verweisen. Sobald eines der Assoziationsenden einer Assoziation dies erlaubt, können mehrere Instanzen der Assoziation dieselben Instanzen der beteiligten Klassen verbinden. In diesem Fall besitzen die Instanzen der Assoziation zusätzliche Kennungen, die eine eindeutige Identifizierung ermöglichen.

Ein Assoziationsende kann zudem als geordnet gekennzeichnet werden. Die Werte, die in den Instanzen eines solchen Assoziationsendes existieren, sind sequentiell geordnet, so dass eine Abbildung von der Menge der positiven ganzen Zahlen auf die Menge der Werte existiert.

Assoziationen dürfen beliebig viele Assoziationsenden besitzen. Wenn eine Assoziation n Assoziationsenden besitzt, besteht das entgegengesetzte Ende für eine Instanz eines Assoziationsendes aus den Instanzen der anderen $n - 1$ Assoziationsenden, die auf die jeweilige Instanz verweisen. Diese Eigenschaft wird in der CMOF-Spezifikation auf $n = 2$ eingeschränkt, so dass in MOF ausschließlich binäre Assoziationen modelliert werden können.

In früheren Versionen der MOF-Spezifikation sind ebenfalls nur binäre Assoziationen erlaubt. Allerdings wurde die Erweiterung um n-äre Assoziationen dort als eine der geplanten Änderungen in zukünftigen Versionen der Spezifikation angegeben (siehe [Obj02b], Seite C-4). Es wird kein Grund für die nachträgliche Einschränkung, die einem der früheren Ziele widerspricht, angegeben. Vermutlich ist die stärkere Angleichung der Semantik von MOF an die Semantik objektorientierter Programmiersprachen ein Grund für den Verzicht auf Assoziationen in EMOF und die eingeschränkte Verfügbarkeit in CMOF.

Binäre Assoziationen werden in der Regel durch eine durchgezogene Linie zwischen zwei Klassen notiert. Sowohl die Assoziation als auch die beiden Assoziationsenden können einen Namen enthalten. Der Name der Assoziation wird mittig in der Nähe der durchgezogenen Linie angegeben und die Namen der Assoziationsenden an den Enden derselben Linie. Abb. 5.9 zeigt die zwei Klassen **Trommel** und **Motor** sowie die Assoziation **antreiben** mit den Assoziationsenden **trommel** und **motor**. Der Pfeil neben dem Namen der Assoziation gibt die Leserichtung an. Diese Information lässt sich im Metamodell der MOF leider nur durch die Reihenfolge der Assoziationsenden repräsentieren.

Abbildung 5.9: Grundlegende Notation von Assoziationen

Assoziationen können einfache Beziehungen zwischen Klassen oder Kompositionsbeziehungen beschreiben. Wenn zwei Klassen über eine Assoziation verbunden sind, deren Assoziationsenden beide nicht als Verbund gekennzeichnet sind, dann sind die Lebenszyklen ihrer Instanzen vollständig voneinander entkoppelt. Auf Implementierungsebene bedeutet dies in der Regel auch, dass das Kopieren einer Instanz der einen Klasse nicht zum Kopieren der Instanz der anderen Klasse führt. Einfache Assoziationen werden nicht besonders gekennzeichnet.

Die stärkere Form einer Assoziation ist die Komposition. Eine Komposition ist asymmetrisch, da ein Assoziationsende auf die Komponente und das entgegengesetzte Assoziationsende auf den Verbund verweist, der die Komponente enthält. Es kann daher nur jeweils ein Assoziationsende einer Assoziation als Verbund gekennzeichnet werden. Die Kennzeichnung erfolgt im Modell durch Setzen des Attributes **isComposite** in der entsprechenden Instanz der Metaklasse **Property**. Die Notation einer Kompositionsbeziehung erfolgt durch eine gefüllte Raute an dem Assoziationsende, das auf den Verbund verweist. Kompositionsbeziehungen führen zu Einschränkungen der Komponenten. Diese können nur zu genau einem Verbund gehören, und ihr Lebenzzyklus ist an den Lebenszyklus des Verbundes gekoppelt. Auf Implementierungsebene bedeutet dies, dass Erzeugen bzw. Löschen eines Verbundes gleichzeitig das Erzeugen bzw. Löschen seiner Komponenten bedeutet.

Assoziationen können von anderen Assoziationen abgeleitet werden. Abgeleitete Assoziationen definieren eine Beziehung zwischen zwei Klassen, die aus anderen Eigenschaften der Klasse ermittelt werden kann. Dies ist vergleichbar mit abgeleiteten Attributen, die auf Seite 73 vorgestellt werden. Genau wie abgeleitete Attribute werden abgeleitete Assoziationen durch einen dem Namen vorangestellten Schrägstrich gekennzeichnet.

Neben der Möglichkeit, Assoziationen abzuleiten oder als Teilmenge anderer Assoziationen zu definieren, können Assoziationen die Eigenschaften anderer Assoziationen erben. Wenn eine Assoziation eine andere Assoziation erbt, korrespondiert jedes Assoziationsende der erbenden Assoziation mit einem Assoziationsende der vererbenden Assoziation. Die Typen der Assoziationsenden der erbenden Assoziation stehen in einer *„ist ein“*-Beziehung zu den Typen der korrespondierenden Assoziationsenden. Vererbung zwischen Assoziation wird analog zu der Vererbung zwischen Klassen durch einen Pfeil von der erbenden zur vererbenden Assoziation notiert.

5.3.3 Datentypen

Datentypen sind die Grundbausteine der Modellierung mit der UML-Infrastrukturbibliothek. Der wesentliche Unterschied von Datentypen gegenüber Klassen (siehe Abschnitt 5.3.1) ist, dass Datentypen eine Menge von Werten repräsentieren, während Klassen die Struktur und das

Verhalten einer Menge von Objekten mit eigener Identität darstellen. Typische Datentypen sind Zahlen und Zeichenketten.

Die UML-Infrastrukturbibliothek unterstützt die Definition sowohl einfacher als auch komplexer (strukturierter) Datentypen. Außerdem können Aufzählungstypen spezifiziert werden. Einfache Datentypen sind z. B. ganze Zahlen, Fließkommazahlen, Zeichen und Zeichenketten. Strukturierte Datentypen sind in ihrem Aufbau vergleichbar mit den strukturierten Datentypen in Programmiersprachen wie *PASCAL* und *C/C++*. Aufzählungstypen bezeichnen Datentypen, deren Wertemenge aus einer Anzahl von Bezeichnungen besteht. Aufzählungstypen entsprechen Datentypen, die durch **„typedef enum“** in C++ definiert werden können.

Ähnlich den Klassen können Datentypen Attribute und Operationen enthalten. Durch die Attribute wird das Modellieren strukturierter Datentypen ermöglicht. Weil die Instanzen eines Datentyps im Unterschied zu den Instanzen einer Klasse keine Identität besitzen, sind die Operationen, die auf Datentypen definiert sind, reine Rechenoperationen. Die Operationen eines Datentyps verwenden den Wert einer Instanz, um eine Berechnung durchzuführen, ändern den Wert der Instanz aber nicht.

Im MOF-Metamodell werden einfache und strukturierte Datentypen durch die Metaklasse **Datatype** und Aufzählungstypen durch die Metaklassen **Enumeration** und **EnumerationLiteral** repräsentiert. Strukturierte Datentypen werden erst im Paket **Constructs** der UML-Infrastruktur eingeführt und können daher nur in CMOF-Modellen verwendet werden. Abb. 5.10 zeigt die Repräsentation von Datentypen im **Constructs**-Paket der UML-Infrastruktur. Diese Definitionen werden in der CMOF-Spezifikation unverändert übernommen.

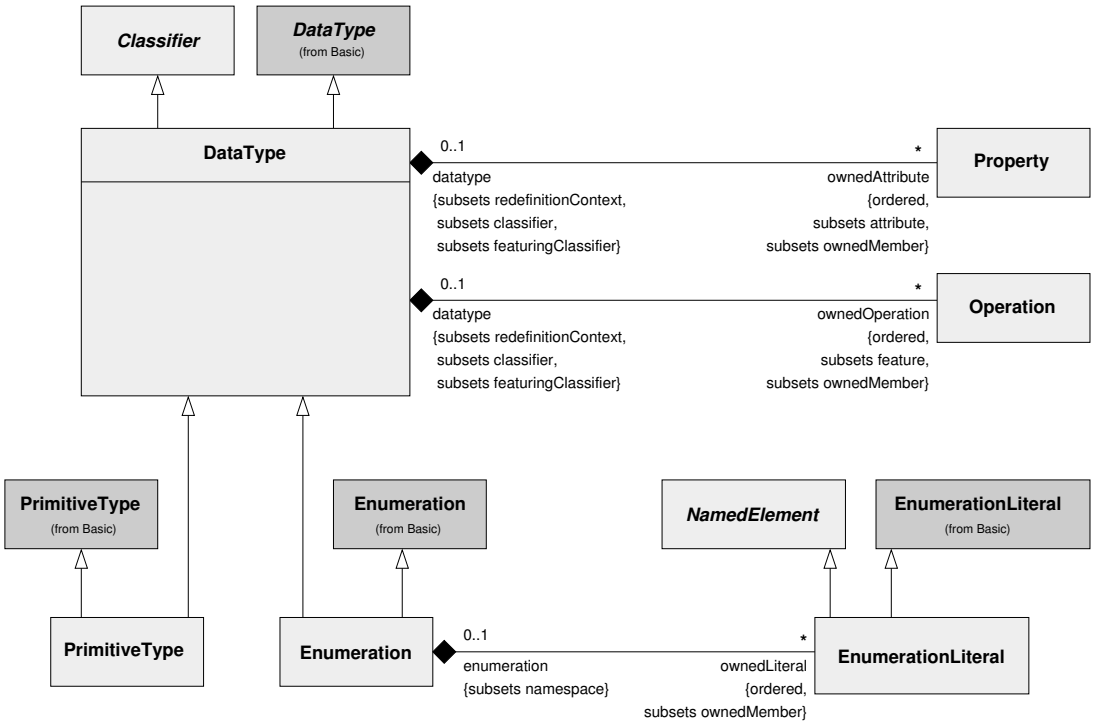


Abbildung 5.10: Repräsentation von Datentypen im Metamodell der UML-Infrastruktur

Ein einfacher Datentyp wird durch eine Instanz der Metaklasse **Datatype** repräsentiert. Einfache Datentypen enthalten weder Attribute noch Operationen, sondern werden auf Modellierungsebene lediglich durch einen Namen beschrieben. Die Semantik eines einfachen Datentyps wird erst durch die Abbildung auf eine Implementierungstechnologie festgelegt. Einfache Datentypen sind der Startpunkt aller Modelle der UML-Infrastruktur und bilden die Grundlage für die Abbildung von Modellen auf Programmiersprachen.

Im Paket **PrimitiveTypes** der UML-Infrastruktur werden die vier Grunddatentypen **Boolean**, **Integer**, **String** und **UnlimitedNatural** definiert. **Boolean** definiert einen Aufzählungstyp mit den Werten **true** und **false**. **Integer** repräsentiert den Wertebereich der ganzen Zahlen, während **UnlimitedNatural** die Menge der natürlichen Zahlen zuzüglich des Wertes ***** zur Kennzeichnung des Wertes $+\infty$ darstellt. Der Datentyp **String** stellt alle Zeichenketten dar.

Strukturierte Datentypen werden ebenfalls durch Instanzen der Metaklasse **Datatype** repräsentiert. Ein strukturierter Datentyp in MOF repräsentiert einen Verbund, der aus einem oder mehreren Feldern besteht. Die einzelnen Felder eines strukturierten Datentyps werden durch Attribute modelliert, die durch Instanzen der Metaklasse **Property** dargestellt werden.

Aufzählungstypen besitzen eine endliche Anzahl von Werten, die durch eine geordnete Menge von Namen repräsentiert wird. Die Aufzählungstypen werden durch Instanzen der Metaklasse **Enumeration** dargestellt und die Wertemenge durch Instanzen der Metaklasse **EnumerationLiteral**. Jede Instanz der Klasse **EnumerationLiteral** ist höchstens einer Instanz der Klasse **Enumeration** zugeordnet.

Die 1.x-Versionen der MOF-Spezifikation enthalten weitere Datentyparten, die in MOF 2.x ebenfalls durch die Klasse **Datatype** repräsentiert werden. Der **CollectionType** aus MOF 1.x wird in MOF 2.0 durch Angabe einer „passenden“ Multiplizität beschrieben und der **AliasType** durch einen Datentyp, der den Grunddatentyp des **AliasType** erbt und durch weitere Bedingungen entsprechend einschränkt.

5.3.4 Pakete

Moderne Klassenbibliotheken wie das *Java Development Kit (JDK)* bestehen aus einer Vielzahl von Klassen. Zudem nimmt mit jeder neuen Version einer Klassenbibliothek deren Funktionsumfang und damit in der Regel auch die Anzahl der Klassen deutlich zu. Ähnliches lässt sich im Bereich der Modellierungssprachen beobachten. Während das Metamodell der MOF-Version 1.4 lediglich 29 Klassen enthält, sind es in der Version 2.0 bereits 122.[5]

Um trotz des zunehmenden Umfangs der Bibliotheken bzw. Modelle die Verständlichkeit, Wartbarkeit und Wiederverwertbarkeit zu gewährleisten, muss eine Einteilung in logische Einheiten erfolgen. Diese muss es ermöglichen, nur die Teile des gesamten Modells zu verwenden, die zur Lösung eines bestimmten Problems erforderlich sind. Innerhalb der UML-Infrastrukturbibliothek stellt das Modellierungselement *Paket* die notwendigen Mechanismen bereit, die zur Aufteilung von Modellen in logische Einheiten benötigt werden.

[5] einschließlich des Metamodells der UML-Infrastrukturbibliothek

Pakete definieren einen Namensraum, der die im Paket enthaltenen Modellelemente umschließt. Die Namen der enthaltenen Modellelemente sind innerhalb des umgebenden Namensraumes eindeutig. Dies ermöglicht eine Kapselung von Modellelementen und damit eine Aufteilung eines Modells in logische Einheiten. Da Pakete ebenfalls zu den Modellelementen gehören, die von einem Paket umgeben und nach außen gekapselt werden können, wird durch diesen Mechanismus eine hierarchische Schachtelung von Paketen ermöglicht. Die Repräsentation des Schachtelungsmechanismus' ist im Metamodell der UML-Infrastruktur durch das Modellierungselement **Namespace** implementiert und wird in Abb. 5.11 dargestellt.

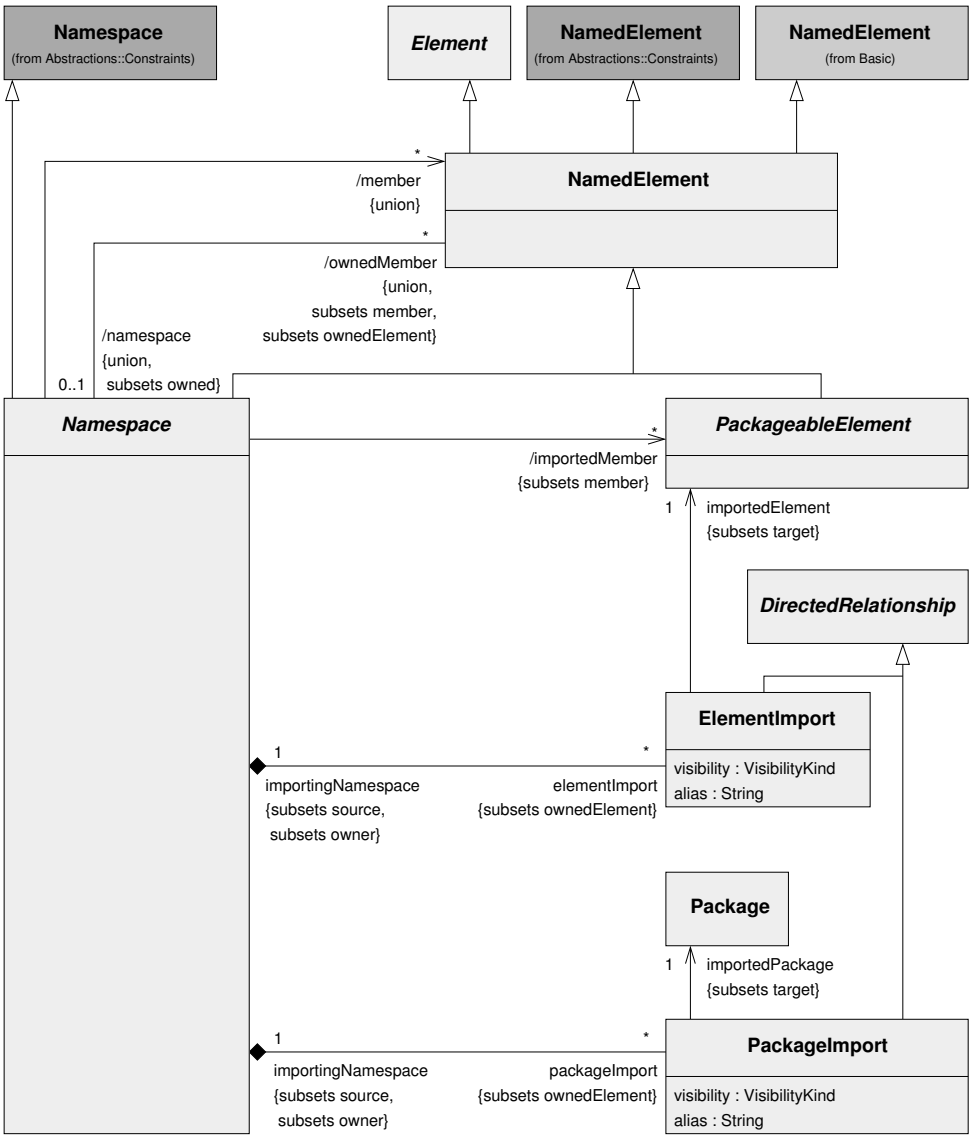


Abbildung 5.11: Namensräume im Metamodell der UML-Infrastruktur

Um die in einem Paket gekapselten Modellelemente wiederverwenden zu können, werden Mechanismen benötigt, die es ermöglichen, von außen auf gekapselte Modellelemente zuzugreifen.

Zu diesem Zweck stellt die UML-Infrastrukturbibliothek zwei Mechanismen bereit. Pakete können sowohl einzelne Modellelemente als auch vollständige Pakete importieren und mit anderen Paketen vereinigt werden.

Die Möglichkeit, einzelne Modellelemente importieren zu können, wird durch die Metaklasse **ElementImport** repräsentiert. Ein Namensraum kann eine Menge von Instanzen der Klasse **ElementImport** enthalten, die jeweils den Import eines einzelnen Modellelementes darstellen. Es können nur Modellelemente importiert werden, die Instanzen der Metaklasse **PackageableElement** bzw. deren Subklassen sind. Der Import vollständiger Pakete wird durch die Metaklasse **PackageImport** bereitgestellt, die analog zur Klasse **ElementImport** definiert ist. Der einzige Unterschied ist, dass jede Instanz der Klasse **PackageImport** den Import eines kompletten Pakets repräsentiert. Beide Möglichkeiten des Imports sind in Abb. 5.11 dargestellt.

Zusätzliche Möglichkeiten, Pakete miteinander zu verknüpfen, stellt die Metaklasse **PackageMerge** bereit. Diese symbolisiert eine bestimmte Art, Pakete miteinander zu „verschmelzen“. Die Klasse **PackageMerge** wird im **Constructs**-Paket eingeführt und in CMOF um Varianten erweitert. In Abb. 5.12 sind die Metaklasse **PackageMerge** und ihre Beziehungen zur Klasse **Package** dargestellt. Das Diagramm zeigt sowohl die Definitionen im Paket **Constructs** als auch die Definitionen aus CMOF. Alle Elemente, die in CMOF neu hinzukommen, sind grau hinterlegt.

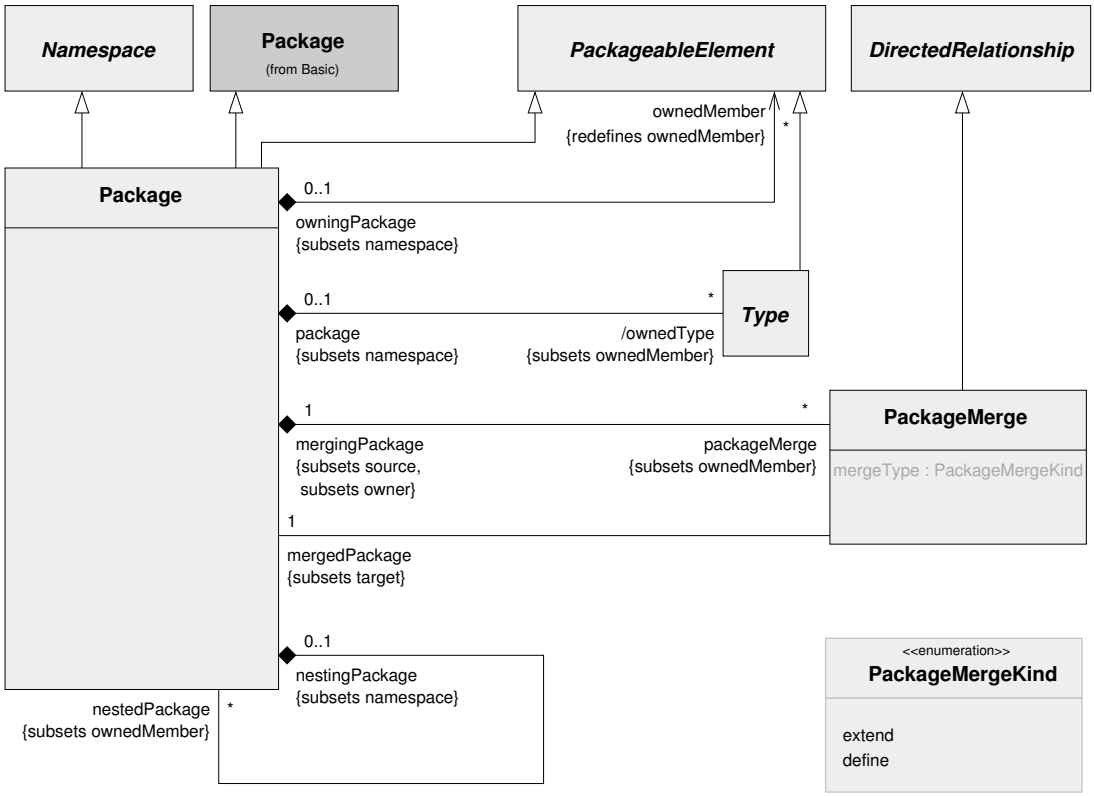


Abbildung 5.12: Paketrepräsentation im Metamodell der CMOF

Im Folgenden werden die verschiedenen Möglichkeiten erläutert, die in CMOF zur Aufteilung eines Modells verwendet werden können. Mit Ausnahme der Schachtelung werden alle angegebenen Möglichkeiten erst auf der **Constructs**-Schicht bzw. in CMOF selbst definiert, so dass sie in EMOF nicht zur Verfügung stehen.

5.3.4.1 Schachtelung

Der Schachtelungsmechanismus ist sowohl in EMOF als auch in CMOF verfügbar. Er stellt eine einfache und stark an Programmiersprachen erinnernde Möglichkeit bereit, Modelle zu strukturieren. Pakete werden bei einer Schachtelung durch eine Kompositionsbeziehung miteinander verbunden. Das äußere Paket ist ein Container, der das innere Paket enthält.

Die wesentliche Funktion der Verschachtelung von Paketen ist die Kapselung des innen liegenden Paketes. Alle Klassen und Datentypen, die in einem inneren Paket existieren, sind nach außen gekapselt. Dies gilt allerdings nur dann, wenn ein Element als nach außen nicht sichtbar („private“) markiert wird. Andernfalls ist ein Modellelement nach außen sichtbar und kann in andere Pakete importiert oder über seinen vollständigen Namen, d. h. die Angabe aller umgebenden Pakete in Schachtelungsreihenfolge zuzüglich des Elementnamens referenziert werden.

5.3.4.2 Import

Eine Importbeziehung zwischen einem Paket und einem Modellelement bedeutet, dass eine Referenz auf das Modellelement in den Namensraum des importierenden Paketes eingefügt wird. Da nur eine Referenz und nicht das Modellelement selbst eingefügt wird, lassen sich importierte Modellelemente nicht verändern, sondern nur benutzen. Importierte Modellelemente können wiederum importiert werden.

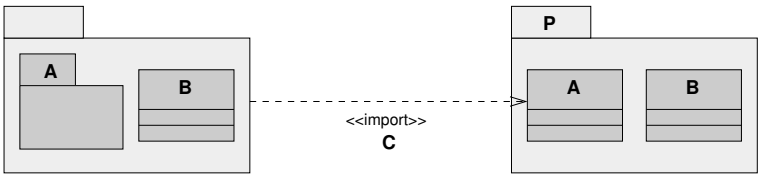


Abbildung 5.13: Notation einer Importbeziehung

Abb. 5.13 zeigt die Notation einer Importbeziehung zwischen einem importierenden Paket und einer importierten Klasse. Das Paket **Q** importiert die Klasse **A** aus dem Paket **P**. Dies wird durch einen gestrichelten Pfeil vom Paket **Q** zur Klasse **A** dargestellt. Für jede Importbeziehung kann ein Aliasname vergeben werden, der anstelle des Namens des Modellelementes in den Namensraum des importierenden Paketes eingefügt wird. Auf diese Weise können Namenskonflikte zwischen importierten Modellelementen und bereits im Namensraum des Paketes enthaltenen Modellelementen vermieden werden. In Abb. 5.13 wird für den Import der Klasse **A** aus dem Paket **P** in das Paket **Q** der Aliasname **C** vergeben, weil sonst ein Namenskonflikt zu dem in **Q** definierten Paket **A** entstehen würde.

Importbeziehungen können Namenskonflikte mit anderen Modellelementen hervorrufen. Die Modellelemente können innerhalb des importierenden Paketes oder in einem das importierende Paket einschließenden Paket definiert sein. Tritt der Namenskonflikt zwischen einem Modellelement des importierenden Paketes und einem importierten Element auf, muss das importierte Element über den vollständigen Namen angesprochen werden. Der lokale Name bezieht sich in diesem Fall auf das Modellelement, das zu dem importierenden Paket gehört. Dies ist in Abb. 5.14 dargestellt.

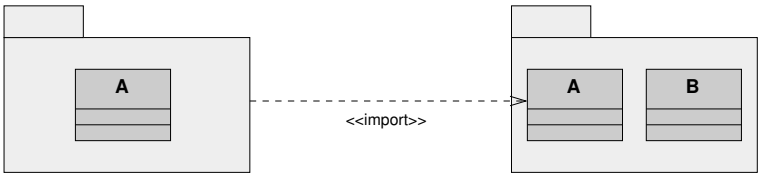


Abbildung 5.14: Namenskonflikt zwischen enthaltenem und importiertem Modellelement

Die zweite Möglichkeit eines Namenskonfliktes zeigt Abb. 5.15. Hier wird in einem Paket, das das importierende Paket einschließt, ein Modellelement definiert, das in einem Namenskonflikt zu dem importierten Modellelement steht. In diesem Fall verdeckt das importierte Modellelement das im äußeren Paket definierte Modellelement und kann mit dem lokalen Namen angesprochen werden. Das im äußeren Paket definierte Modellelement muss über den vollständigen Namen referenziert werden.

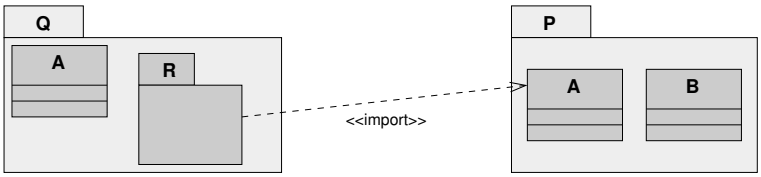


Abbildung 5.15: Namenskonflikt zwischen sichtbarem und importiertem Modellelement

Eine Importbeziehung zwischen zwei Paketen fügt alle Namen des importierten Paketes zu dem Namensraum hinzu, den das importierende Paket definiert. Der Import eines gesamten Paketes ist semantisch äquivalent zum Import aller in dem importierten Paket enthaltenen Modellelemente. Sämtliche oben beschriebenen Eigenschaften für den Import einzelner Modellelemente gelten auch bei Importen gesamter Pakete.

5.3.4.3 Vereinigung

Eine Vereinigungsbeziehung zwischen zwei Paketen bedeutet, dass der Inhalt des einen Paketes mit dem Inhalt des zweiten Paketes durch Spezialisierung und Redefinition vereinigt wird. Der Mechanismus der Vereinigung sollte nur dann angewendet werden, wenn die Elemente verschiedener Pakete dieselbe Bedeutung haben. Das vereinigende Paket nimmt die gleichnamigen Modellelemente der zu vereinigenden Pakete auf und fügt diese zu einem Element zusammen.

Vereinigungsbeziehungen sind eine Art Abkürzungsnotation, die anstelle expliziter Spezifikationen der Vererbungs- und Redefinitionsbeziehungen zwischen den Modellelementen der beteiligten Pakete angegeben werden kann. In Abb. 5.16 ist die Notation für die Vereinigung eines Quellpaketes mit einem Zielpaket angegeben. Sie besteht aus einem gestrichelten Pfeil mit der zusätzlichen Angabe des Schlüsselwortes **«merge»** vom Quellpaket zum Zielpaket. Der Pfeil gibt an, dass die Modellelemente des Zielpaketes in das Quellpaket „eingeschmolzen“ werden.

Abbildung 5.16: Notation für die Vereinigung von Paketen

Die Vereinigung eines Quellpaketes mit einem Zielpaket kann auf eine Importbeziehung vom Quell- zum Zielpaket und eine Menge von Regeln zurückgeführt werden. Die Regeln beschreiben, wie die Modellelemente des Zielpaketes in das Quellpaket eingefügt werden. Sie werden auf alle Modellelemente des Zielpaketes angewendet, die einen Sichtbarkeitsbereich haben, der sie auch außerhalb des Paketes sichtbar sein lässt. Da die Elemente des Zielpaketes lediglich in das Quellpaket eingefügt werden, haben Vereinigungen keinerlei Auswirkungen auf die beteiligten Zielpakete.

Für jede Klasse, jeden Datentyp und jede Assoziation des Zielpaketes wird im Quellpaket ein gleichnamiges Modellelement erzeugt, falls durch die Erzeugung kein Namenskonflikt verursacht wird. Im Folgenden wird nur das Vorgehen bei der Vereinigung von Klassen angegeben. Bei der Vereinigung von Datentypen und Assoziationen wird in gleicher Weise verfahren.

Die neu erzeugten Klassen spezialisieren die gleichnamigen Klassen des Zielpaketes. Entsprechend werden paarweise Generalisierungsbeziehungen zwischen gleichnamigen Klassen in Ziel- und Quellpaket erzeugt. Falls durch die Erzeugung einer Klasse im Quellpaket ein Namenskonflikt entsteht, wird keine neue Klasse, sondern lediglich eine Generalisierungsbeziehung zwischen der Klasse im Zielpaket und dem gleichnamigen Typ im Quellpaket erzeugt. Da Pakete auch Modellelemente beinhalten können, die nicht generalisierbar sind, kann dieses Verfahren nicht für alle Modellelemente verwendet werden. Die Spezifikation schreibt vor, dass nicht-generalisierbare Modellelemente einfach in das Quellpaket kopiert werden sollen. Wie in diesem Fall Namenskonflikte aufgelöst werden sollen, bleibt unspezifiziert.

In Abb. 5.17 ist ein Beispiel dargestellt. Es enthält die grundlegenden Fälle, die bei der Vereinigung von Paketen auftreten können, wenn die Pakete ausschließlich Klassen enthalten. Das Paket **Q** vereinigt das Paket **P**, das die Klassen **A** und **B** enthält. **P** selbst enthält ebenfalls eine Klasse **A**. Die Vereinigung wird in eine Importbeziehung zwischen **Q** und **P** umgewandelt. Da das Paket **Q** bereits eine Klasse **A** enthält, muss diese nicht neu erzeugt werden, sondern es wird lediglich eine Vererbungsbeziehung von der in **Q** bereits enthaltenen zu der von **P** importierten Klasse **A** erzeugt. **Q** enthält keine Klasse **B**. Daher wird sowohl eine Klasse **B** neu erzeugt als auch eine Vererbungsbeziehung von der neu erzeugten Klasse zur importierten Klasse **B** aus dem Paket **P**.

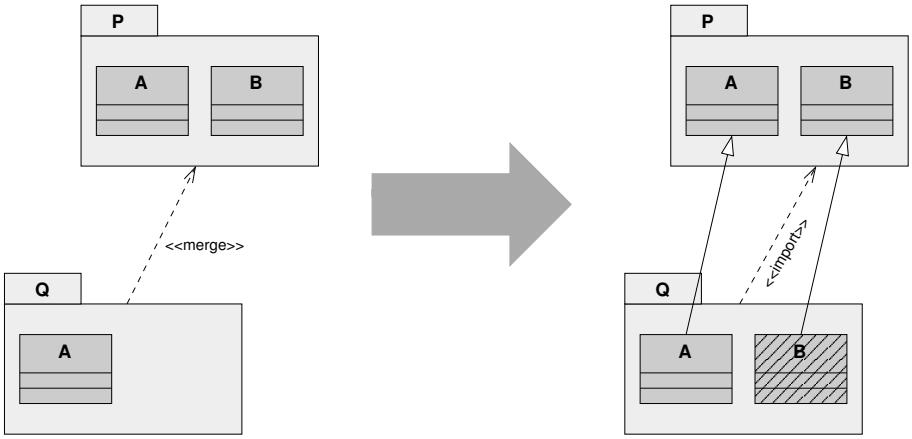


Abbildung 5.17: Umsetzung geschachtelter Klassen in Vereinigungsbeziehungen

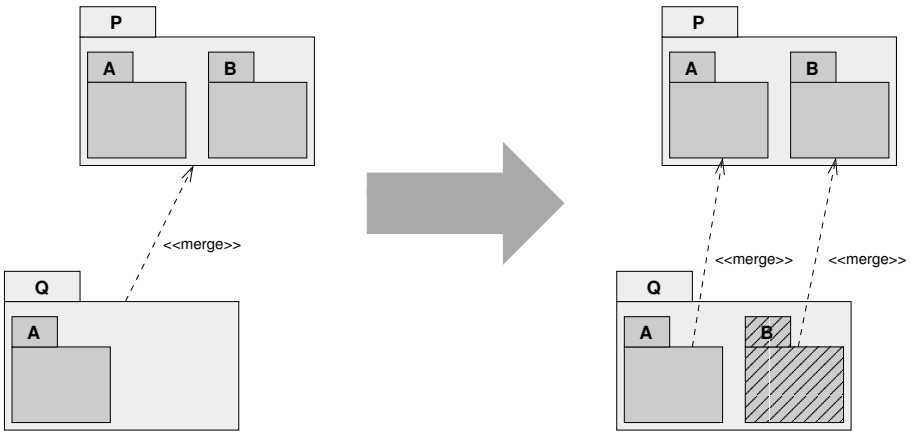


Abbildung 5.18: Umsetzung geschachtelter Pakete bei Vereinigungsbeziehungen

Die Klassen im Quellpaket erhalten Redefinitionen der Attribute und Operationen, die alle Referenzen der Klasse, Datentypen und Assoziationen des Zielpaketes durch Referenzen auf die neu erzeugten Klassen, Datentypen und Assoziationen im Quellpaket ersetzen. Außerdem werden weitere Generalisierungsbeziehungen zwischen den Klassen des Quellpaketes und Klassen des Zielpaketes erzeugt. Jede Klasse des Quellpaketes, die von einer Klasse des Zielpaketes erbt, erbt zusätzlich von allen Superklassen der Superklasse, die ebenfalls im Zielpaket definiert sind. Falls mehrere Zielpakete gleichnamige Klassen enthalten, wird im Quellpaket eine neue Klasse erzeugt, die die Eigenschaften sämtlicher gleichnamiger Klassen innerhalb der Zielpakete erhält.

Für jedes geschachtelte Paket des Zielpaketes wird ein geschachteltes Paket im Quellpaket erzeugt, falls dadurch kein Namenskonflikt zu einem bereits existierenden geschachtelten Paket verursacht wird. In jedem Fall wird eine Vereinigung der gleichnamigen geschachtelten Pakete in Quell- und Zielpaket durchgeführt.

Abb. 5.18 zeigt eine Vereinigungsbeziehung zwischen den Paketen **P** und **Q**, wobei **P** das Ziel- und **Q** das Quellpaket ist. Das Paket **P** enthält die Pakete **A** und **B**; das Paket **Q** enthält ein Paket **A**. Die Vereinigungsbeziehung wird aufgelöst, indem in Paket **Q** ein Paket **B** eingefügt wird und für die gleichnamigen Pakete in **P** und **Q** jeweils Vereinigungsbeziehungen erzeugt werden.

Für jede Importbeziehung des Zielpaketes wird eine gleichnamige Importbeziehung im Quellpaket erzeugt. Die Modellelemente, die in den importierten Paketen des Zielpaketes definiert sind, werden nicht in das Zielpaket „eingeschmolzen“. Modellelemente, die durch Importbeziehungen im Quellpaket sichtbar werden, werden von durch Vereinigung erzeugten Modellelementen überdeckt. Tritt also ein Namenskonflikt zwischen einem Modellelement aus einer Importbeziehung und einem Modellelement aus einer Vereinigungsbeziehung auf, kann das Modellelement aus der Importbeziehung nur über seinen vollständigen Namen referenziert werden.

In CMOF wird eine zusätzliche Variante der Vereinigung von Paketen eingeführt. Die oben beschriebene Variante wird dort als *Erweiterung* bezeichnet und durch **PackageMergeKind::extend** gekennzeichnet. Grafisch wird diese Art der Paketvereinigung durch Angabe des Stereotyps **merge** dargestellt. Zusätzlich wird eine als *Definition* bezeichnete Variante eingeführt, die durch **PackageMergeKind::define** spezifiziert und grafisch durch Angabe des Stereotyps **combine** notiert wird. Der Unterschied zwischen beiden Varianten ist, dass die Elemente des Zielpaketes in das Quellpaket rekursiv kopiert (engl. „deep-copy“) anstatt vererbt und redefiniert werden. Im Unterschied zur Erweiterung entstehen bei einer Definition keine zusätzlichen Beziehungen zwischen den Elementen der beteiligten Pakete.

Der Mechanismus der Vereinigung von Paketen bietet die Möglichkeit, mit wenig Schreibarbeit komplexe Strukturen auszudrücken. Allerdings lässt die Genauigkeit der Spezifikation etwas zu wünschen übrig, insbesondere bez. der Voraussetzungen für die Anwendbarkeit der Mechanismen und dem Vorgehen bei der Auflösung von Namenskonflikten. Möglicherweise werden diese Schwachpunkte bis zur Freigabe der MOF-Version 2.0 noch beseitigt.

5.3.5 Bedingungen

Durch Pakete, Klassen, Assoziationen und Datentypen kann ein statisches Modell beschrieben werden. Konsistenzbedingungen auf einem Modell lassen sich mit diesen Modellelementen allerdings nicht ausdrücken. Dazu werden weitere Elemente benötigt. In MOF werden zur Repräsentation von Bedingungen die Modellelemente verwendet, die in Abb. 5.19 dargestellt sind.

Durch die Vererbungsbeziehung zur Metaklasse **Elements** können Bedingungen allen Elementen zugeordnet werden, die ebenfalls von dieser Klasse erben. Dies sind insbesondere die Grundbausteine der MOF, die in den vorangegangenen Abschnitten beschrieben wurden. Eine Möglichkeit, eine Bedingung in einem MOF-Modell strukturiert auszudrücken, ist ein Baum aus Instanzen der Metaklasse **Expressions**. Jede einzelne Instanz repräsentiert entweder ein terminales oder ein nicht-terminales Symbol und besitzt eine beliebige Anzahl von Operanden. Die andere Möglichkeit wird durch die Metaklasse **OpaqueExpression** bereitgestellt. Instanzen dieser Klasse repräsentieren eine Bedingung, die in einer beliebigen Sprache ausgedrückt

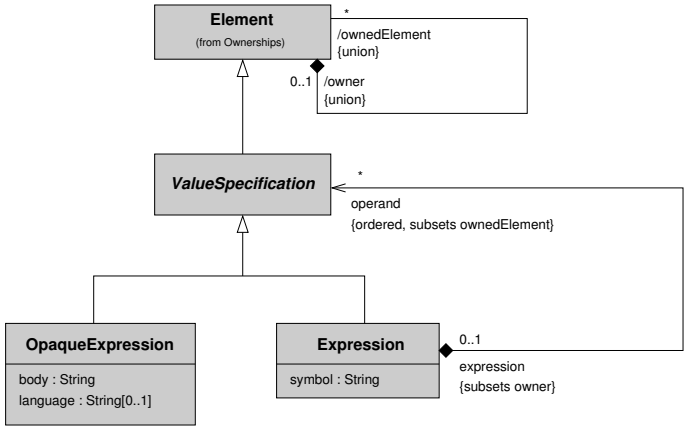


Abbildung 5.19: Elemente zur Definition von Bedingungen

werden darf. Die Bedingung wird im Attribut **body** spezifiziert, und die zur Definition der Bedingung verwendete Sprache wird im Attribut **language** angegeben. In den meisten Fällen wird die *Object Constraint Language* (OCL) [OCL03] verwendet, um Bedingungen innerhalb von MOF- und UML-Modellen zu spezifizieren.

5.4 Zusammenfassung

In diesem Kapitel wurde die Metamodellierungssprache MOF vorgestellt. Diese wird in der Version 2.0 auf Basis der UML-Infrastruktur definiert. Die UML-Infrastruktur [UP03a] stellt die grundlegenden Bestandteile objektorientierter Modellierungssprachen bereit, die von der MOF [ACC+03] genutzt und erweitert werden. Die UML-Infrastruktur ist modular angelegt, um als Basis für eine große Menge domänenspezifischer Modellierungssprachen verwendet werden zu können. In gleicher Weise ist auch die MOF aufgebaut.

Die MOF-Spezifikation definiert zwei Sprachen, die auf unterschiedliche Einsatzbereiche zugeschnitten sind. *Essential MOF (EMOF)* baut auf der **Basic**-Schicht der UML-Infrastruktur auf und ist eine eng an die Eigenschaften von Programmiersprachen angelehnte Modellierungssprache. Dadurch ist EMOF leicht auf Programmiersprachen abbildbar und bietet die Möglichkeit, einfache Modelle mit einfachen Sprachmitteln zu modellieren. *Complete MOF (CMOF)* setzt auf der **Constructs**-Schicht der UML-Infrastruktur auf und erweitert die EMOF um viele Ausdrucksmöglichkeiten. Diese sind geeignet, auch komplexe Metamodelle ausdrücken und übersichtlich darstellen zu können.

Leider bringt die modulare Aufteilung der UML-Infrastruktur- und MOF-Spezifikationen auch schwerwiegende Nachteile bezüglich Lesbarkeit und Verständlichkeit mit sich, da sich die zum Verständnis der Modellierungselemente notwendigen Informationen in vielen Fällen nur schwer finden lassen. Dies liegt zum einen an der komplexen Vererbungs- und Importstruktur und zum anderen an der alphabetischen Auflistung der Elemente der Metamodelle.

Die Grundbausteine der Modellierung in MOF sind *Datentypen*, *Klassen* und *Assoziationen*. Datentypen stellen die Grundlage für die objektorientierte Modellierung bereit, insbesondere zur Definition von Klassen. Klassen bestimmen die Struktur und das Verhalten der Objekte, die die abzubildenden Informationen repräsentieren. Da MOF eine Modellierungssprache zur Definition von Metamodellen ist, und in Metamodellen die Struktur im Vordergrund steht, spielt die Definition des Objektverhaltens im Vergleich zur Definition der Struktur der Objekte eine untergeordnete Rolle.

Neben den genannten Grundbausteinen werden in den meisten MOF-Modellen die Modellierungselemente *Paket* und *Bedingung* genutzt. Pakete werden zur Strukturierung eines Modells eingesetzt und können auf vielfältige Weise kombiniert werden, um Modularität und Wiederverwendbarkeit eines Modells zu ermöglichen. Den meisten Modellelementen können außerdem Bedingungen zugeordnet werden, die Eigenschaften der Modellelemente genauer spezifizieren. Bedingungen werden meist durch die OCL [OCL03] ausgedrückt, können aber auch Anweisungen einer beliebigen anderen Sprache enthalten.

Kapitel 6

Extensible Markup Language

Die *Extensible Markup Language* (XML) ist eine allgemein verwendbare Sprache zur Beschreibung der Struktur eines Dokumentes. XML ist aus der *Standard Generalized Markup Language* (SGML) [Int86] hervorgegangen und wird durch eine Spezifikation [XML00] festgelegt, die vom *World Wide Web Committee* (W3C) [W3C] verabschiedet wurde.

Aufgrund der allgemeinen Verwendbarkeit wird XML in vielen verschiedenen Bereichen eingesetzt. Aktuelle Versionen von Büroanwendungen verwenden XML ebenso zur Speicherung der erzeugten Dokumente wie spezialisierte Werkzeuge zur Softwareentwicklung. Auch Modelle, die durch eine MOF-basierte Modellierungssprache beschrieben werden, lassen sich durch XML-Dokumente darstellen. Die Spezifikation, die den Aufbau entsprechender XML-Dokumente beschreibt, verwendet die meisten der Elemente aus XML und den verwandten W3C-Spezifikationen. Um das Verständnis des XMI-Standards zu erleichtern, der in Kapitel 7 beschrieben wird, erfolgt in diesem Kapitel eine Einführung in XML.

Die Bestandteile von XML, die zur Strukturierung von Daten verwendet werden können, werden in Abschnitt 6.1 vorgestellt. Da sich die Strukturen der in verschiedenen Anwendungsfällen zu speichernden Daten meist ebenso stark unterscheiden wie die Anwendungsfälle selbst, werden zusätzliche Informationen benötigt, um die Struktur eines XML-Dokumentes zu beschreiben. Diese Informationen müssen den Aufbau der Elemente eines XML-Dokumentes festlegen, so dass eine Anwendung herausfinden kann, ob ein Dokument Informationen enthält, die die Anwendung verarbeiten kann.

Anfangs wurde zur Beschreibung der Struktur einer Klasse von XML-Dokumenten *Dokumenttypdefinitionen* (DTDs) eingesetzt. DTDs enthalten Definitionen von Elementen, Attributen und Entitäten. Elemente und Attribute beschreiben den Aufbau von XML-Elementen und XML-Attributen der XML-Dokumente, die konform zu der DTD sind. Durch Entitäten lassen sich DTDs modularisieren. Eine Entität ist ein Baustein, der in einer DTD mehrfach verwendet werden und wiederum Elemente und andere Entitäten enthalten kann.

Aufgrund einiger Nachteile der DTDs, im Wesentlichen fehlende Ausdrucksmöglichkeiten und die von XML abweichende Syntax, wurde eine neue Sprache entworfen, um den Aufbau von

XML-Dokumenten zu spezifizieren. Diese Sprache, *XML-Schema*, basiert selbst auf XML und stellt umfassende Möglichkeiten bereit, die zulässigen Bestandteile eines XML-Dokumentes zu definieren. Die erste Version von XML-Schema wurde vom W3C im Mai 2001 verabschiedet und hat seitdem in vielen Bereichen die DTDs abgelöst. Die Sprachelemente von XML-Schema werden in Abschnitt 6.2 beschrieben.

6.1 Bestandteile

Die Bestandteile eines XML-Dokumentes werden in der XML-Spezifikation [XML00] festgelegt. Daneben existieren einige weitere Spezifikationen, die auf der XML-Spezifikation aufsetzen und weitere Bestandteile definieren. Die „Namensräume in XML“-Spezifikation [Nam99] beschreibt einen Mechanismus zur Vergabe eindeutiger Namen für XML-Elemente. Die *XLinks/XPointer*-Spezifikationen [XLi01, XPo01] definieren verschiedene Möglichkeiten, XML-Elemente zu referenzieren.

Im Folgenden werden die wesentlichen Bestandteile von XML beschrieben, die in den genannten Spezifikationen festgelegt werden. Zunächst wird der Aufbau von XML-Elementen und XML-Attributen beschrieben, die die wesentlichen Bausteine zur Repräsentation von Daten in XML-Dokumenten darstellen. Anschließend werden die Mechanismen zur eindeutigen Benennung und Referenzierung von XML-Elementen beschrieben.

6.1.1 Elemente und Attribute

Die XML-Spezifikation beschreibt den Aufbau von XML-Dokumenten. Ein XML-Dokument besteht aus hierarchisch angeordneten XML-Elementen. Es muss genau ein Wurzelelement existieren, damit ein XML-Dokument gemäß der XML-Spezifikation *wohlgeformt* ist. Die XML-Elemente eines Dokumentes werden häufig auch als *Knoten* bezeichnet.

XML-Elemente bestehen aus einer einleitenden Kennzeichnung, dem Elementinhalt und einer beendenden Kennzeichnung. Eine einleitende Kennzeichnung beginnt mit dem „kleiner“-Zeichen **<** und endet mit dem „größer“-Zeichen **>**. Eine beendende Kennzeichnung beginnt mit **</** und endet mit **>**. Die einleitenden und beendenen Kennzeichnungen eines XML-Elementes müssen gleichlautend benannt sein. Der Name steht jeweils zwischen den Markierungszeichen **<** bzw. **</** und **>**.

```
<Waschmaschine>
    ...
</Waschmaschine>
```

(a) mit Inhalt

```
<Waschmaschine/>
```

(b) ohne Inhalt

Abbildung 6.1: Aufbau eines XML-Elementes

Abb. 6.1 zeigt zwei Varianten des Aufbaus eines XML-Elementes, das eine Waschmaschine repräsentieren könnte. Das XML-Element in Abb. 6.1(a) besteht aus einer einleitenden

Kennzeichnung **<Waschmaschine>**, dem Inhalt des Elementes, der durch **...** dargestellt ist, und der beendenden Kennzeichnung **</Waschmaschine>**. Die Kennzeichnungen eines XML-Elementes können zusammengefasst werden, wenn das Element keinen Inhalt enthält. In diesem Fall lässt sich das Beispiel-Element durch eine einzige Kennzeichnung **<Waschmaschine/>** ausdrücken (siehe Abb. 6.1(b)).

Der Inhalt eines XML-Dokumentes ist in den XML-Elementen enthalten. Die Strukturierung des Inhalts erfolgt durch hierarchische Schachtelung der XML-Elemente ausgehend vom Wurzelknoten. Bei der Schachtelung von XML-Elementen ist darauf zu achten, dass sich die einleitende und die beendende Kennzeichnung eines Knotens auf derselben Ebene befinden und denselben übergeordneten Knoten besitzen müssen. Abb. 6.2 zeigt Beispiele für eine gültige und zwei ungültige Schachtelungen von XML-Elementen.

```
<Waschmaschine>
  <Trommel>
  </Trommel>
</Waschmaschine>
```
(a) gültig

```
<Waschmaschine>
  <Trommel>
</Waschmaschine>
</Trommel>
```
(b) ungültig

```
<Waschmaschine>
  <Trommel>
</Waschmaschine>
<Waschmaschine>
  </Trommel>
</Waschmaschine>
```
(c) ungültig

Abbildung 6.2: Schachtelung von XML-Elementen

Abb. 6.2(a) zeigt eine korrekte Schachtelung zweier XML-Elemente **Waschmaschine** und **Trommel**. Das XML-Element **Trommel** ist vollständig innerhalb des Elementes **Waschmaschine** definiert. In den Abb. 6.2(b) und 6.2(c) sind zwei Verstöße gegen die oben angegebenen Regeln für eine korrekte Schachtelung dargestellt. In Abb. 6.2(b) befindet sich die einleitende Kennzeichnung nicht auf derselben Schachtelungsebene wie die beendende Kennzeichnung, und in Abb. 6.2(c) besitzen die einleitende und die beendende Kennzeichnung verschiedene übergeordnete Elemente.

Jedes XML-Element kann beliebig viele XML-Attribute enthalten, die nähere Informationen über die repräsentierten Daten oder das Element selbst beschreiben. XML-Attribute werden durch eine Zuweisung der Form **AttributName=AttributWert** innerhalb der einleitenden Kennzeichnung eines XML-Elementes angegeben. Jeder Attributname darf innerhalb eines Elementes nur einmal auftreten. Name und Baujahr einer durch das XML-Element aus Abb. 6.1 dargestellten Waschmaschine können so durch Erweiterung der einleitenden Kennzeichnung durch die XML-Attribute **name** und **baujahr** angegeben werden. In Abb. 6.3 ist eine um diese Attribute erweiterte Darstellung einer Waschmaschine dargestellt.

```
<Waschmaschine name="WV-0815" baujahr="1990"/>
```

Abbildung 6.3: Repräsentation von Attributen in XML

Die Daten eines XML-Elementes sind also entweder in den Attributen oder dem Inhalt des Elementes enthalten. Für den Inhalt eines Elementes definiert die XML-Spezifkation verschiedene Modelle. Ein XML-Element kann entweder leer sein, nur Kindelemente besitzen oder eine Mischung aus Textelementen und XML-Elementen enthalten. In letzterem Fall wird das Element mit dem Attribut **mixed** gekennzeichnet, dessen Wert auf „true“ gesetzt wird. Für Elemente mit gemischtem Inhalt können zwar die Typen der enthaltenen XML-Elemente durch eine DTD beschränkt werden, nicht aber Reihenfolge und Anzahl ihres Auftretens.

Die XML-Spezifikation schränkt auch die Menge der zur Benennung von XML-Elementen und XML-Attributen verwendbaren Zeichen ein. Namen von XML-Elementen und XML-Attributen dürfen nur aus Buchstaben, Zahlen sowie Bindestrich (**-**), Unterstrich (_), Punkt (**.**) und Doppelpunkt (**:**) bestehen. Zudem muss jeder Name mit einem Buchstaben, einem Unterstrich oder einem Doppelpunkt beginnen. Es ist zu beachten, dass die „Namensräume in XML“-Spezifikation dem Doppelpunkt eine besondere Bedeutung zuweist. Die Verwendung von Doppelpunkten in Namen sollte daher vermieden werden.

Einige Zeichen bekommen in einem XML-Dokument eine besondere Bedeutung. Wie bereits beschrieben, markieren die **<**- und **>**-Zeichen beispielsweise Anfang und Ende der Kennzeichnungen von XML-Elementen. Um solche Zeichen innerhalb eines XML-Dokumentes als Text und nicht als Kennzeichnungsanfang oder -ende interpretieren zu können, lassen sich sogenannte *Entitäten* verwenden oder die Zeichen in eine *Character Data*-Umgebung einkapseln.

Eine XML-Entität wird durch eine Zeichenkette benannt und kann in XML-Dokumenten referenziert werden. Während des Einlesens eines XML-Dokumentes durch einen XML-Parser werden die Referenzen durch die Zeichen ersetzt. Die Zeichen **<** und **>** werden z. B. durch die vordefinierten Entitäten **lt** bzw. **gt** ersetzt. Eine Referenz auf eine Entität wird durch ein **&**, gefolgt vom Namen der Entität und einem Semikolon angegeben. Soll also das Zeichen **<** in einem XML-Dokument als Text interpretiert werden, wird **<** anstelle des **<** im Text angegeben.

Alternativ kann eine Behandlung als Text erzwungen werden, indem der Text innerhalb eines CDATA (Character Data) Abschnittes angegeben wird. Innerhalb einer CDATA-Umgebung interpretiert der XML-Parser keinerlei besondere Bedeutungen von Zeichen, sondern immer nur das Zeichen selbst. Eine CDATA-Sektion wird durch **<![CDATA[** eingeleitet und durch **]]>** geschlossen. In Abb. 6.4 sind zwei Beispiele dargestellt, die die Kennzeichnung **<Class>** innerhalb eines XML-Elementes angeben.

```
<Beschreibung>
   ein &lt;Trommel&gt; - Element
   beschreibt die Trommel einer Waschmaschine
</Beschreibung>

<Beschreibung>
   <![CDATA[ein <Motor> - Element
   beschreibt einen Motor einer Waschmaschine]]>
</Beschreibung>
```

Abbildung 6.4: Verwendung von Sonderzeichen in einem Textelement

Abb. 6.4 zeigt zwei XML-Elemente des Typs **Beschreibung**, die zur Beschreibung der XML-Elementtypen **Trommel** und **Motor** verwendet werden. In der Beschreibung des Typs **Trommel** werden das **<**- und das **>**-Zeichen durch die korrespondierenden XML-Entities ausgedrückt. Zur Beschreibung des XML-Elementtyps **Motor** wird der vollständige Text in eine **CDATA**-Umgebung eingeschlossen, so dass die Sonderzeichen direkt angegeben werden können.

6.1.2 Namensräume

XML ist eine vielfältig einsetzbare Dokumentenbeschreibungssprache. Sie ist geeignet, um unterschiedlichste Informationen zu repräsentieren. Häufig beinhaltet ein XML-Dokument verschiedene Informationen und wird von verschiedenen Softwaresystemen verarbeitet. Aus diesem Grund treten leicht Kollisionen zwischen den Namen von XML-Elementen und XML-Attributen auf. Ein Beispiel dafür sind verschiedene objektorientierte Modellierungssprachen. Jede dieser Sprachen enthält die grundlegenden Merkmale objektorientierter Sprachen wie Klasse und Objekt. Allerdings kann die konkrete Ausprägung dieser Merkmale von Sprache zu Sprache leicht voneinander abweichen. Verarbeitet nun eine Software für eine Modellierungssprache Daten, die in einer anderen Modellierungssprache formuliert sind, wird es in der Regel zu Problemen kommen. Da die grundlegenden Merkmale aber ähnlich oder sogar gleich sind, kann die Software nur schwer erkennen, dass die Daten nicht für sie bestimmt sind. Dieses Problem kann in nahezu allen Bereichen auftreten, in denen XML zur Repräsentation komplexer Datenstrukturen verwendet wird. Aus diesem Grund werden global eindeutige Bezeichner benötigt, die die Unterscheidung gleichnamiger Elemente in verschiedenen Zusammenhängen ermöglichen.

Die XML-Namensräume sind ein Mechanismus, der die Angabe global eindeutiger Bezeichner für XML-Elemente und XML-Attribute erlaubt. Gemäß der Spezifikation des W3C ist „ein XML-Namensraum eine Zusammenstellung von Namen, die in XML-Dokumenten als Elementtypen und Attributnamen verwendet und durch durch einen Verweis auf einen *Universal Resource Identifier* (URI) [BLFM98] identifiziert werden“. Ein URI ist nach Definition entweder ein *Universal Resource Name* (URN) oder ein *Universal Resource Locator* (URL). Ein URN ist ein Name, der einer bestimmten Ressource zugeordnet wird, um diese eindeutig zu identifizieren. Dabei bleibt der Name konstant, auch wenn sich die Ressource verändert. Ein URN besteht aus vier Teilen, die durch Doppelpunkte voneinander getrennt werden. Die einzelnen Teile sind die Angabe URN, die Angabe des Standards bzw. der Normbezeichnung (z. B. ISBN), die Angabe der verantwortlichen Stelle und der eigentlichen Bezeichnung des Objektes. Ein Beispiel einer URN ist **URN:ISBN:derVerlag:14355678921**. Die gebräuchlichste Verwendung von URL´s sind die Adressen des *World Wide Web (WWW)*. Ein Beispiel ist **http://www.w3c.org**, die Internetseite des W3C. Zwei URIs werden genau dann als identisch angesehen, wenn sie zeichenweise identisch sind.

Die URIs werden einem Präfix zugeordnet, das anschließend zur Bezeichnung von XML-Elementen und -Attributen verwendet werden kann. Dazu wird das reservierte Attribut **xmlns** verwendet. Eine Deklaration eines Namensraumes hat daher die Form: **xmlns:*prefix*=*URI***. Ein eindeutiger Name kann nach der Deklaration eines Namensraumes aus einem Präfix und ei-

nem lokalen Namen zusammengesetzt werden. Die beiden Bestandteile werden durch einen Doppelpunkt (**:**) voneinander getrennt. Das Präfix für einen Namensraum kann auch die leere Zeichenkette sein. Dies entspricht der Deklaration eines voreingestellten Namensraumes. Jedem innerhalb eines Dokumentes angegebenen Element, dessen Name kein Präfix enthält, wird dieser Namensraum zugeordnet. Wird dagegen die leere Zeichenkette als URI angegeben, ist für den entsprechenden Geltungsbereich kein Namensraum voreingestellt.

Der Geltungsbereich einer Deklaration eines Namensraumes umfasst das Element, in dem sie angegeben wird, und alle Elemente, die sich innerhalb dieses Elementes befinden. Dies gilt allerdings nur dann, wenn die Deklaration nicht überschrieben wird. Überschreiben lässt sich eine Deklaration, indem einem Präfix ein neuer URI zugewiesen wird. In ähnlicher Weise gilt dies auch für den voreingestellten Namensraum. Allerdings gelten voreingestellte Namensräume nur für XML-Elemente, nicht aber für XML-Attribute.

```
<document xmlns:T = "http://www.trommeln.de"
          xmlns:M = "http://www.motoren.de" >

   <M:Motor id= "1" >
      ...
   </M:Motor>
   <T:Trommel id= "2" >
      ...
   </T:Trommel>

   ...

 </document>
```

Abbildung 6.5: Verwendung von Namensräumen in XML-Dokumenten

In Abb. 6.5 ist ein Ausschnitt eines XML-Dokumentes dargestellt, das den Aufbau einer Waschmaschine repräsentiert. Im Wurzelelement des Dokumentes werden zwei Namensräume **T** und **M** deklariert, die die Hersteller von Trommel und Motor der Waschmaschine bezeichnen. Die Präfixe der Namensdeklarationen werden zur eindeutigen Bezeichnung der XML-Elemente verwendet. Die Trommel wird durch ein **T:Trommel**-Element und der Motor durch ein **M:Motor**-Element repräsentiert. Hätte der Motorhersteller ebenfalls Trommeln im Angebot und würde diese in seiner Definition als **Trommel** definieren, könnten diese durch Verwendung des Präfix eindeutig identifiziert werden.

Meist werden Namensräume im Wurzelelement eines XML-Dokumentes deklariert. Alternativ kann die Deklaration auch im XML-Element selbst erfolgen. In Abb. 6.6 ist die Deklaration eines XML-Elementes zur Definition der Trommel einer Waschmaschine angegeben, die die Deklaration des Namensraumes enthält.

```
<T:Trommel id= "2" xmlns:T = "http://www.trommeln.de" />
```

Abbildung 6.6: Deklaration eines Namenraumes in einem XML-Element

Gemäß der Spezifikation „Namensräume in XML“ müssen auch Attribute eindeutig benannt sein. Dies bedeutet, dass ein Element sowohl keine zwei Attribute desselben Namens enthalten darf als auch keine zwei Attribute besitzen darf, deren qualifizierte Namen identische lokale Teile haben und deren Präfixe demselben Namensraum zugeordnet sind. In Abb. 6.7 sind zulässige und unzulässige Attributkombinationen innerhalb eines Elementes dargestellt.

```
<Waschmaschine xmlns:="http://www.dieWaschen.de"
               xmlns:baujahr="http://www.dieWaschen.de">
    <eintrag name="WV-0814" baujahr="1991"/>
    <eintrag name:value="WV-0816" baujahr:value="1993"/>
</Waschmaschine>
```

(a) zulässig

```
<Waschmaschine xmlns:name="http://www.dieWaschen.de"
               xmlns:baujahr="http://www.dieWaschen.de">
    <eintrag value="WV-0814" value="1991"/>
    <eintrag name:value="WV-0814" baujahr:value="1991"/>
</Waschmaschine>
```

(b) unzulässig

Abbildung 6.7: Benennung von XML-Attributen (aus [Nam99])

In Abb. 6.7(a) ist eine Liste von Waschmaschinen dargestellt, die zwei korrekte Einträge enthält. Der erste Eintrag ist korrekt, weil die beiden Attribute unterschiedlich benannt sind. Der zweite Eintrag ist korrekt, weil sich der voreingestellte Namensraum nicht auf Attribute auswirkt, so dass die beiden **value**-Attribute unterschiedlichen Namensräumen zugeordnet und daher eindeutig benannt sind. Abb. 6.7(b) zeigt eine Liste von Waschmaschinen mit zwei unzulässigen Einträgen. Der erste Eintrag ist unzulässig, weil er das Attribut **value** zweimal enthält, und der zweite Eintrag ist unzulässig, weil die beiden Präfixe **baujahr** und **name** demselben Namensraum zugeordnet sind. In beiden Fällen sind die Attribute nicht eindeutig benannt und daher nicht unterscheidbar.

6.1.3 Referenzen

Ein XML-Element repräsentiert in der Regel eine Information. Häufig ist diese Information erst im Zusammenhang mit anderen Informationen verwertbar. Um Zusammenhänge zwischen Informationen in XML-Dokumenten ausdrücken zu können, müssen die entsprechenden XML-Elemente miteinander verknüpft werden. Daher werden Möglichkeiten benötigt, ein XML-Element sowohl mit anderen XML-Elementen innerhalb desselben Dokumentes als auch mit XML-Elementen in anderen XML-Dokumenten zu verknüpfen. Die XML-Spezifikation [XML00] und die auf XML aufsetzende W3C-Spezifikation *XML Linking Language* (XLink) [XLi01] definieren verschiedene Mechanismen zur Referenzierung von XML-Elementen.

Die Möglichkeit, ein XML-Element zu referenzieren, setzt voraus, dass das Element eindeutig identifizierbar ist. Um dies zu gewährleisten, darf jedes XML-Element genau ein Attribut des Typs **ID** besitzen. Der Wert dieses Attributes wird als eindeutige Identifikation eines XML-Elementes innerhalb eines XML-Dokumentes verwendet. Dieser Wert kann zum Referenzieren von XML-Elementen verwendet werden. Das Attribut, das zur Identifikation eines XML-Elementes verwendet wird, wird häufig **id** genannt. Es kann aber jeder beliebige andere Name verwendet werden.

Referenzen werden in XML-Dokumenten durch XML-Attribute angegeben, die einen der Typen **IDREF** oder **IDREFS** besitzen. Eine Referenz genau eines XML-Elementes wird ausgedrückt, indem ein XML-Attribut des Typs **IDREF** denselben Wert erhält wie das XML-Attribut des Typs **ID** des referenzierten XML-Elementes. In Abb. 6.8 sind erneut die zwei XML-Elemente **Motor** und **Trommel** dargestellt, die sich gegenseitig referenzieren.

```
<Motor id="1" trommel="2"/>

<Trommel id="2" motor="1"/>
```

Abbildung 6.8: Referenz eines Elementes innerhalb eines XML-Dokumentes

Das XML-Element **Motor** besitzt ein XML-Attribut **id**, das mit dem Wert **1** belegt ist. Analog dazu besitzt das XML-Element **Trommel** ein XML-Attribut **id**, das mit dem Wert **2** belegt ist. Außerdem enthält das XML-Element **Motor** ein XML-Attribut **trommel**, das den Wert **2** besitzt und damit das XML-Element **Trommel** referenziert. Umgekehrt wird durch das XML-Attribut **motor**, das innerhalb des XML-Elementes **Trommel** definiert ist, das XML-Element **Motor** referenziert.

Sollen mehrere XML-Elemente gleichzeitig referenziert werden, wird ein XML-Attribut des Typs **IDREFS** verwendet. Die Belegung eines solchen XML-Attributes besteht aus einer Liste von Schlüsselwerten, die durch Leerzeichen voneinander getrennt werden. In Abb. 6.9 ist ein Beispiel angegeben, das die Verwendung eines Attributes des Typs **IDREFS** zeigt. Das XML-Element **Waschmaschine** referenziert durch sein Attribut **bestandteile** sowohl das XML-Element **Trommel** als auch das XML-Element **Motor** aus Abb. 6.8.

```
<Waschmaschine id="42" komponenten="1 2"/>
```

Abbildung 6.9: Referenz mehrerer Elemente innerhalb eines XML-Dokumentes

Die in Abb. 6.9 dargestellte Referenz ist nur dann möglich, wenn das XML-Element **Waschmaschine** und zwei XML-Elemente mit den IDs 1 und 2, z. B. die XML-Elemente **Motor** und **Trommel** aus Abb. 6.8, innerhalb eines Dokumentes deklariert sind. Andernfalls müssen Mechanismen aus den W3C-Spezifikationen XLink und XPointer verwendet werden, um XML-Elemente dokumentenübergreifend zu identifizieren.

Die XLink-Spezifikation definiert verschiedene Mechanismen zur Verknüpfung von XML-Elementen. Jede Verknüpfung wird durch ein XML-Element repräsentiert, das ein Attribut **type** besitzt. Grundsätzlich wird zwischen einfachen und erweiterten Verknüpfungen unterschieden.

Eine einfache Verknüpfung wird gekennzeichnet, indem das Attribut **type** den Wert **simple** zugewiesen bekommt. Einfache Verknüpfungen weisen immer von genau einem lokalen zu genau einem entfernten Element und können durch Angabe folgender Attribute näher beschrieben werden:

- **href** enthält eine URI zur Identifikation des entfernten Elementes
- **role**, **arcrole** und **title** werden zur näheren Beschreibung der Semantik einer Verknüpfung verwendet. **role** und **arcrole** sind URIs, die auf Beschreibungen des Zielelementes der Verknüpfung bzw der Verknüpfung verweisen. **title** ist eine Zeichenkette, die verwendet werden kann, um der Verknüpfung einen Namen zuzuweisen.
- **show** und **actuate** beschreiben das Verhalten einer Anwendung bei der Verfolgung einer Verknüpfung. Die XLink-Spezifikation definiert eine Reihe von möglichen Belegungen der Attribute, die jeweils ein bestimmtes Anzeige- bzw. Ladeverhalten der Anwendung festlegen. Diese Attribute sind daher in der Regel nur bei WWW-Anwendungen von Interesse.

In Abb. 6.10 ist ein Beispiel einer einfachen Verknüpfung gemäß dem XLink-Standard dargestellt. Das dargestellte XML-Element **Waschmaschine** repräsentiert wiederum eine Waschmaschine. Die Bestandteile dieser Waschmaschine werden durch XML-Elemente repräsentiert, die innerhalb des **Waschmaschine**-Elementes deklariert werden. Das angegebene XML-Element referenziert die Trommel der Waschmaschine, die im Dokument **Trommel.xml** deklariert ist. Dazu wird der XLink-Namensraum importiert und dem Präfix **xlink** zugewiesen. Die XLink-Attribute **type** und **href** werden verwendet, um eine einfache Verknüpfung zur Trommel zu beschreiben.

```
<Waschmaschine id= "42">
  <komponente
    xmlns:xlink = "http://www.w3c.org/1999/xlink"
    xlink:type = "simple"
    xlink:href = "Trommel.xml" />
    ...
</Waschmaschine>
```

Abbildung 6.10: Einfache Verknüpfung nach XLink-Standard

Eine erweiterte Verknüpfung kann eine beliebige Anzahl von XML-Elementen miteinander verbinden. Die Verknüpfung wird von einem XML-Element repräsentiert, dessen **type**-Attribut den Wert **extended** besitzt. Innerhalb dieses XML-Elementes sind weitere XML-Elemente deklariert, deren **type**-Attribut auf einen der Werte **locator**, **arc**, **resource** oder **title** gesetzt ist.

Die verschiedenen Attributwerte beschreiben die verschiedenen Eigenschaften der Bestandteile einer erweiterten Verknüpfung. XML-Elemente des Typs **locator** beschreiben Verknüpfungen zu externen Elementen, während XML-Elemente des Typs **resource** Verknüpfungen zu lokalen

Elementen spezifizieren. Ein XML-Element des Typs **arc** beschreibt, wie die Verknüpfung traversiert werden kann bzw. darf. Durch Angabe eines XML-Elementes des Typs **title** kann einer Verknüpfung ein Titel zugewiesen werden.

Die enthaltenen XML-Elemente beschreiben die Bestandteile der Verknüpfung. In Abb. 6.11 ist ein Beispiel für eine erweiterte Verknüpfung gemäß der XLink-Spezifikation angegeben, das die Referenzen der Bestandteile einer Waschmaschine in einer erweiterten Verknüpfung zusammenfasst.

```
<Waschmaschine id= "42">
  xmlns:xlink = "http://www.w3c.org/1999/xlink" >

  <Komponenten xlink:type= "extended" >
    <extern xlink: type= "locator"
            xlink: href= "Trommel.xml"/>
    <extern xlink: type= "locator"
            xlink: href= "Motor.xml"/>
    <Gehaeuse xlink:type= "resource">
      <farbe value= "blau"/>
      ...
    <Gehaeuse/>
   ...
  </Komponenten>
</Waschmaschine>
```

Abbildung 6.11: Erweiterte Verknüpfung nach XLink-Standard

Die erweiterte Verknüpfung wird durch das XML-Element **Komponenten** umschlossen. Jeder Bestandteil einer Waschmaschine, der in einem eigenen Dokument deklariert ist, wird durch ein XML-Element des Typs **locator** referenziert. Die lokale Definition des Waschmaschinengehäuses wird durch ein Element des Typs **resource** dargestellt.

6.2 Dokumentklassen

Alle bisher vorgestellten Elemente und Mechanismen der XML beschreiben lediglich den Aufbau der Elemente eines XML-Dokumentes im Allgemeinen. Anwendungen verarbeiten jedoch in der Regel nur XML-Dokumente, die anwendungsspezifische XML-Elemente und -Attribute enthalten. Daher wird ein Mechanismus benötigt, um die Menge aller XML-Dokumente in Klassen zu unterteilen und die Zugehörigkeit eines XML-Dokumentes zu einer bestimmten Klasse entscheiden zu können. Ein XML-Dokument muss nicht explizit zu einer Dokumentklasse zugeordnet werden. Es genügt, dass der Inhalt des Dokumentes die durch die Dokumentklasse vorgegebenen Einschränkungen erfüllt.

Wie am Beginn dieses Kapitels bereits erwähnt, stammt XML von SGML ab und verwendete ursprünglich die Dokumenttypdefinitionen (DTDs) der SGML zur Beschreibung von Dokumentklassen. Aufgrund fehlender Ausdrucksmöglichkeiten und der von XML abweichenden

Syntax der DTDs wurde XML-Schema vom W3C als Nachfolger für die DTDs standardisiert. Der Standard besteht aus Spezifikationen der zulässigen Strukturen [XML01b] und Datentypen [XML01c] sowie einer allgemeinen Einführung [XML01a]. Trotz der Verfügbarkeit von XML-Schema seit dem Jahr 2001 sind nach wie vor viele Dokumentarten nur durch DTDs beschrieben. Da neuere XML-basierte Standards und Werkzeuge, wie z. B. XMI 2.1 (siehe Kapitel 7) aber meist auf XML-Schema basieren, werden die Elemente von DTDs im Rahmen dieser Arbeit nicht näher behandelt, sondern es wird im Folgenden ausschließlich auf die Elemente von XML-Schema eingegangen.

Ein XML-Schema beschreibt eine Dokumentklasse, die aus der Menge aller XML-Dokumente besteht, die konform zu dem Schema sind. Dazu werden in einem XML-Schema Aufbau und Anordnung aller Elemente und Attribute beschrieben, die in einem konformen Dokument vorkommen dürfen. Eine besondere Dokumentklasse ist die Klasse der XML-Schemata. Diese wird ebenfalls durch ein XML-Schema beschrieben, so dass XML-Schema „durch sich selbst“ definiert wird. Die Beziehung zwischen XML-Schema und Dokumentklassen ist vergleichbar mit der Beziehung zwischen MOF und Modellierungssprachen, die eine Instanz von MOF sind (siehe Kapitel 5). XML-Dokumente, die konform zu einem XML-Schema sind, werden daher auch als *Instanzdokumente* bezeichnet.

In den folgenden Abschnitten wird zunächst der grundsätzliche Aufbau eines XML-Schemas beschrieben (Abschnitt 6.2.1). Anschließend wird auf die wichtigsten Bestandteile eines XML-Schemas eingegangen. Abschnitt 6.2.2 beschreibt den Aufbau von Element- und Attributdeklarationen, und Abschnitt 6.2.3 erläutert das Typsystem von XML-Schema.

6.2.1 Aufbau eines XML-Schemas

Ein XML-Schema beginnt mit einem **schema**-Element, das allgemeine Informationen über das Schema enthält. Innerhalb dieses Elementes können verschiedene Grundeinstellungen für das Schema angegeben werden, die sich auf alle Elemente des Schemas auswirken. Unterhalb des Wurzelelementes befinden sich eine optionale Beschreibung des Schemas sowie die Typdefinitionen und Elementdeklarationen, die den Inhalt der Instanzdokumente beschreiben. Zur Beschreibung eines Schemas wird ein **annotation**-Element eingefügt, das Kommentare und Anmerkungen in untergeordneten **appInfo**- und/oder **documentation**-Elementen enthält. Alle Typdefinitionen und Elementdeklarationen müssen eindeutige Namen besitzen, die zu einem bestimmten Namensraum gehören. Dieser Namensraum wird als Ziel-Namensraum bezeichnet. Wenn in einem Schema kein Ziel-Namensraum angegeben wird, können die Elemente ohne Angabe eines Namensraumes, also unqualifiziert, referenziert werden. Abb. 6.12 zeigt den prinzipiellen Aufbau eines XML-Schemas.

Ein XML-Schema für komplexe Instanzdokumente ist häufig ebenso komplex und daher unübersichtlich und schwer zu lesen. Außerdem können häufig Typdefinitionen und Elementdeklarationen in mehreren verwandten Fällen verwendet werden. Um ein XML-Schema in kleine Einheiten aufteilen zu können, bietet die XML-Schema Spezifikation einen Mechanismus an, der das „Einbinden“ eines Schemas in ein anderes Schema ermöglicht. Durch **include schemaLocation=URL** kann ein XML-Schema in ein anderes XML-Schema eingebunden werden.

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema" >
  <xsd:annotation>
    <xsd:documentation>
      Erläuterungen zum Schema
    </xsd:documentation>
  </xsd:annotation>

  Elementdeklarationen und Typdefinitionen
</xsd:schema>
```

Abbildung 6.12: Aufbau eines XML-Schemas

Anschließend können die Definitionen und Deklarationen des eingebundenen Schemas im einbindenden Schema verwendet werden. Dazu werden die eingebundenen Definitionen und Deklarationen in den Ziel-Namensraum eingefügt. Dies setzt voraus, dass alle eingebunden Schemata mit dem gleichen Namensraum deklariert sind.

Eine weiterer Mechanismus zur Unterstützung der Modularität von XML-Schemata ist das **redefine**-Element. Durch ein **redefine**-Element können sowohl einfache und komplexe Elementtypen als auch Gruppen und Attributgruppen redefiniert werden. Ein **redefine** wird syntaktisch wie das **include** zum Einbinden eines Schemas angegeben. Alle Typdefinitionen des redefinierten Schemas werden durch gleichnamige Typdefinitionen des redefinierenden Schemas ersetzt. Aus diesem Grund muss **redefine** vorsichtig angewendet werden, weil leicht Typkonflikte zwischen gleichnamigen Elementen auftreten können. Außerdem lassen sich Elemente in sogenannten Ersetzungsgruppen zusammenfassen. Die Elemente einer Ersetzungsgruppe können wechselseitig für einander eingesetzt werden.

In einem Schema kann außerdem festgelegt werden, ob lokal deklarierte Elemente durch einen Namensraum qualifiziert werden müssen. Durch die Attribute **elementFormDefault** und **attributeFormDefault** des **schema**-Elementes kann für alle Elemente bzw. Attribute eines Instanzdokumentes spezifiziert werden, ob sie qualifiziert sein müssen oder nicht qualifiziert sein dürfen. Alternativ kann auch eine lokale Spezifikation durch das Attribut **form** innerhalb einer Element- bzw. Attributdeklaration erfolgen. In der Grundeinstellung sind alle Elemente und Attribute als nicht qualifiziert angegeben.

6.2.2 Elementdeklarationen

Elemente werden durch **element** deklariert und beschreiben die Elemente, die in einem Instanzdokument auftreten dürfen. Jedes deklarierte Element besitzt einen Typ, der die Elementstruktur festlegt. Die Typangabe kann entweder über das Attribut **type** oder durch einen anonymen Typ erfolgen. Anonyme Typen entsprechen in ihrem Aufbau den in Abschnitt 6.2.3 beschriebenen Typdefinitionen, haben aber keinen Namen und werden innerhalb der Elementdeklaration definiert.

Globale Elemente und Attribute werden direkt unterhalb des **schema**-Elementes angeordnet und können über das **ref**-Attribut referenziert werden. Eine Deklaration, die ein globales Element referenziert, ermöglicht dem referenzierten Element, an derselben Stelle im Dokument zu erscheinen wie die Referenz. Globale Elemente können an oberster Stelle in einem Instanzdokument erscheinen und enthalten selbst keine Referenzen, sondern müssen ihren Typ direkt identifizieren. Dazu wird das **type**-Attribut anstelle des **ref**-Attributes angegeben. Für globale Elemente und Attribute können keine Kardinalitäten festgelegt werden.

Elemente enthalten Informationen in Attributen oder untergeordneten Elementen. Wenn ein Element keine untergeordneten Elemente enthält, wird sein Inhalt als *„leer“* bezeichnet. Leere Elemente werden häufig verwendet, wenn alle Informationen, die durch ein Element ausgedrückt werden sollen, durch Attribute angegeben werden können. Ein Element mit leerem Inhalt besitzt einen Typ, der durch Ableitung des allgemeinsten Elementtyps **anyType** (siehe Abschnitt 6.2.3) definiert werden kann.

Ohne weitere Einschränkung kann jedes Element an jeder Stelle beliebig häufig auftreten. Durch den Einsatz der Attribute **minOccurs** und **maxOccurs** kann die Häufigkeit auf den durch diese Grenzen festgelegten Bereich eingeschränkt werden. Ähnliche Wirkung hat das Attribut **use** für Attributdefinitionen. Dieses kann mit einem der Werte **required**, **optional** oder **prohibited** belegt werden, um festzulegen, dass das Attribut auftreten muss, auftreten darf, bzw. nicht auftreten darf.

```
<xsd:sequence>
    <xsd:element name= "schalter" type= "Schalter"
                 minOccurs="1" maxOccurs= "unbounded"/>
    <xsd:element name= "motor"  type= "Motor"
                 minOccurs="1" maxOccurs= "1"/>

    ...
</xsd:sequence>
...
<xsd:attribute name="Kennung" type="xsd:string" use="required"/>
```

Abbildung 6.13: Einschränkungen von Attributen und Elementen

Abb. 6.13 zeigt wieder einmal einen Ausschnitt einer Typdefinition für eine Waschmaschine. Es wird festgelegt, dass die Waschmaschine eine Menge von Schaltern enthält und genau einen Motor besitzt. Letzteres wird durch Setzen der Attribute **minOccurs** und **maxOccurs** auf „1“ festgelegt. Die Anzahl der Schalter ist nach oben unbegrenzt, da die obere Grenze nicht beschränkt ist; die Waschmaschine muss aber mindestens einen Schalter (Ein-/Aus) besitzen. Außerdem muss jede Waschmaschine eine Kennung erhalten, was durch Setzen des Attributes **use** auf den Wert „required“ erreicht wird.

Vorgabewerte können durch **default** sowohl für Elemente als auch für Attribute definiert werden. Für Elemente wird in einem Instanzdokument der Vorgabewert eingefügt, wenn der Inhalt des Elementes leer ist. Für Attribute wird der Vorgabewert eingefügt, wenn das Attribut nicht angegeben wird. Dies impliziert, dass Vorgabewerte nur für optionale Attribute sinnvoll sind.

Durch das Attribut **fixed** können konstante Elemente bzw. Attribute festgelegt werden. Jedes Attribut bzw. Element wird innerhalb eines Instanzdokumentes durch den gegebenen Wert ersetzt. Daraus folgt, dass **default** und **fixed** sich gegenseitig ausschließen, so dass nur eines der beiden Attribute angegeben werden darf. Beispiele für Vorgabewerte und Konstanten sind in Abb. 6.14 dargestellt.

```
<xsd:attribute name="status" type="Status" default="geplant"/>
<xsd:attribute name="sprache" type="Sprache" fixed="US Englisch"/>
```

Abbildung 6.14: Vorgabewerte und Konstanten

Die Abbildung zeigt die Deklaration eines Elementes mit zwei Attributen. Das Attribut **status** erhält den Vorgabewert „geplant“ und das Attribut **sprache** wird als Konstante deklariert. Diese Angaben wirken sich aus, wenn ein Schemaprozessor ein Instanzdokument einliest. Immer dann, wenn das Attribut **status** im Dokument nicht angegeben ist, wird es eingefügt und mit dem Vorgabewert „geplant“ belegt. Ähnlich wird mit dem Attribut **sprache** verfahren. Der Unterschied ist lediglich, dass eine explizite Deklaration des Attributes **sprache** nur mit dem Wert „US Englisch“ erfolgen darf.

Neben der Möglichkeit, einzelne Elemente zu definieren, bietet XML-Schema auch Mechanismen zur Definition von Elementgruppen an. Auf diese Weise können zusammengehörige Elemente zu einer Einheit verbunden werden, wodurch ein modularer Aufbau von XML-Schemata unterstützt wird. Die XML-Schema-Spezifikation beschreibt drei verschiedene Arten von Gruppen, die durch **choice**-, **sequence**- und **all**-Elemente beschrieben werden. Ein **choice**-Element beschreibt eine Auswahl aus einer Menge von Elementen. In einem Instanzdokument darf für jede Auswahl genau eines der untergeordneten Elemente eingesetzt werden. Ein **sequence**-Element definiert, dass in einem Instanzdokument alle untergeordneten Elemente in der angegebenen Reihenfolge auftreten müssen. Durch Kombination von **choice**- und **sequence**-Elementen können beliebige geschachtelte Strukturen abgebildet werden. Die dritte Möglichkeit, eine Gruppe zu definieren, das **all**-Element, stellt einen Sonderfall dar. In einem Instanzdokument dürfen alle unterhalb des **all**-Elementes angeordneten Elemente in beliebiger Reihenfolge vorkommen. Ein **all**-Element kann nur als einziges Element unterhalb eines **complexContent**-Elementes auftreten.

Ähnlich den Elementen lassen sich auch Attribute zu Gruppen zusammenfassen. Attributgruppen werden durch **attributeGroup**-Elemente definiert. Sie können benannt und über die Namen referenziert werden. Auch dieser Mechanismus kann eingesetzt werden, um die Modularität und Lesbarkeit von XML-Schemata zu verbessern.

Ein Element kann in einem Instanzdokument den Wert **nil** annehmen. Diese Möglichkeit wird in einem Schema angegeben, indem das Attribut **nillable** der Elementdefinition auf den Wert **true** gesetzt wird. Im Instanzdokument wird der Wert des Elementes durch **xsi:nil=“true“** angegeben.

6.2.3 Typdefinitionen

XML-Schema unterstützt die Definition und Verwendung einfacher (skalarer) und komplexer (strukturierter) Typen. Die „eingebauten“ einfachen Typen bilden die Grundlage der Definition sowohl einfacher als auch komplexer Typen. Aufgrund des breiten Anwendungsspektrums der XML enthält XML-Schema eine Vielzahl vordefinierter einfacher Typen. Diese reichen für die meisten Anwendungsdomänen aus und decken insbesondere den betriebswirtschaftlichen Bereich ab. In Abb. 6.15 ist das Typsystem aus XML-Schema abgebildet.

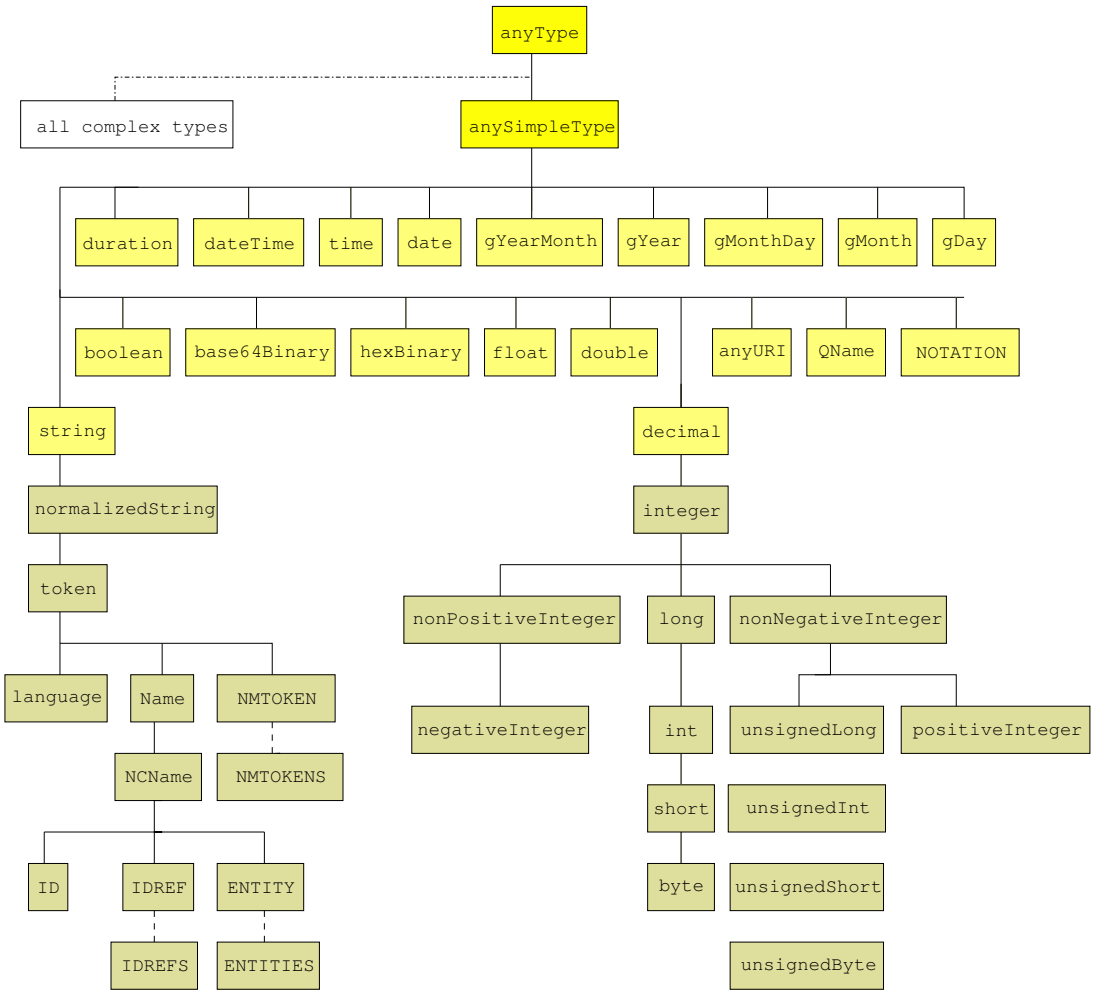


Abbildung 6.15: Typdefinitionen in XML-Schema (aus [XML01c])

Die vordefinierten Datentypen werden in *Urtypen*, *einfache Datentypen* und *abgeleitete Datentypen* unterschieden. Die Urtypen **anyType** und **anySimpleType** bilden den Ursprung des Typsystems in XML-Schema. Ein Element oder Attribut, dessen Typ durch einen der Urtypen definiert ist, kann in einem Instanzelement jeden Typ bzw. jeden einfachen Typ besitzen. Die einfachen Datentypen stellen häufig genutzte Datentypen bereit. Hauptsächlich werden Zahlenformate, Zeichenketten sowie Zeit- und Datumsangaben unterstützt, weil diese in nahezu allen

Anwendungsgebieten benötigt werden, und ein fehlender skalarer Typ nur schwer durch einen benutzerdefinierten Typ ersetzt werden kann. Abgeleitete Datentypen stellen die dritte Gruppe einfacher Datentypen dar. Sie unterscheiden sich strukturell nicht von den vordefinierten einfachen Datentypen, sind aber auf einen bestimmten Anwendungsfall zugeschnitten. Daher fallen neben den vordefinierten Datentypen auch alle benutzerdefinierten einfachen Datentypen in diese Gruppe.

Anwendungsspezifische einfache Datentypen lassen sich durch ein **simpleType**-Element in einem XML-Schema definieren. Benutzerdefinierte einfache Typen bestehen immer aus einer Einschränkung eines bereits existierenden einfachen Datentyps auf eine Teilmenge der Wertemenge des Basistyps. Einschränkungen werden durch ein **restriction**-Element angegeben, das den Basistyp angibt und in Subelementen die Art der Einschränkung definiert. Da zur Einschränkung einer Zeichenkette andere Mechanismen benötigt werden als zur Einschränkung der Wertemenge eines Zahlentyps, sieht die XML-Schema-Spezifikation verschiedene Möglichkeiten der Definition einer Einschränkung vor. Je nach Basistyp kann nur eine Teilmenge der gebotenen Möglichkeiten benutzt werden. Eine Übersicht über die jeweils anwendbaren Einschränkungen gibt Abschnitt 4.1.5 der XML-Schema Spezifikation für Datentypen [XML01c]. In den Abb. 6.16 und 6.17 sind zwei Beispiele für die Ableitung von einfachen Datentypen durch Einschränkung dargestellt. Eine weitere Möglichkeit der Einschränkung von Zeichenketten wird in Kapitel 7 vorgestellt.

```
<xsd:simpleType name="Messbereich">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0" />
    <xsd:maxInclusive value="99"/>
  </xsd:restriction>
</xsd:simpleType>
```

Abbildung 6.16: Einschränkung des Wertebereichs eines Zahlentyps

In Abb. 6.16 wird ein neuer Datentyp **Messbereich** definiert, dessen Wertebereich den darstellbaren Bereich eines Thermometers abdecken soll. Dazu wird der Datentyp von dem vordefinierten Datentyp **integer** abgeleitet und geeignet eingeschränkt. Zur Einschränkung werden durch die Elemente **minInclusive** und **maxInclusive** die untere und die obere Grenze des darstellbaren Intervalls angegeben.

```
<xsd:simpleType name="Kennung"        >
  <xsd:restriction base="xsd:string" >
    <xsd:pattern value="[WV|WT]{0-9}" />
  </xsd:restriction>
</xsd:simpleType>
```

Abbildung 6.17: Einschränkung des Wertebereichs einer Zeichenkette

Abb. 6.17 zeigt die Definition eines Datentyps, der eine in einer bestimmten Art aufgebaute Waschgerätekennung repräsentiert. Der Datentyp **Kennung** wird von dem eingebauten Da-

tentyp **xsd:string** durch Einschränkung abgeleitet. Die konkrete Einschränkung des Wertebereichs des Datentyps **Kennung** gegenüber dem Datentyp **xsd:string** wird durch das Muster **xsd:pattern** bestimmt. Der Wert des Musters ist ein regulärer Ausdruck, der das Vorkommen der einzelnen Zeichen einer konformen Zeichenkette vorgibt. In diesem Fall setzt sich jede konforme Zeichenkette aus „WV“ (für Waschvollautomat) oder „WT“(für Waschtrockner) und einer beliebig langen ganzen Zahl zusammen.

Komplexe Datentypen entsprechen in ihrem Aufbau weitgehend den strukturierten Datentypen aus MOF (siehe Abschnitt 5.3.3). Ihre Definition erfolgt durch ein **complexType**-Element, das aus Elementdeklarationen, Elementreferenzen und Attributdeklarationen besteht. In Abb. 6.18 ist der Aufbau eines komplexen Datentyps dargestellt, der eine Waschmaschine repräsentiert. Für jede Waschmaschine werden der Typ, die Produktionsnummer, das Herstellungsdatum und weitere Informationen gespeichert.

```
<xsd:complexType name="Waschmaschine">
  <xsd:sequence>
    <xsd:element name= "typ" type= "Kennung"/>
    <xsd:element name= "nummer" type= "xsd:string"/>
    <xsd:element name= "herstellungsdatum" type= "xsd:date" />
    ...
  </xsd:sequence>
</xsd:complexType>
```

Abbildung 6.18: Aufbau eines komplexen Datentyps

Komplexe Typen können sowohl von einfachen als auch von komplexen Typen abgeleitet werden. Durch Angabe von **simpleContent** lässt sich kennzeichnen, dass ein komplexer Typ nur einfachen Inhalt enthält. Mit Hilfe solcher Typen lässt sich ein einfacher Inhalt mit einem Attribut verbinden. Als sogenannter gemischter Inhalt („Mixed Content“) wird der Fall bezeichnet, dass ein Element neben Subelementen auch Text enthalten kann. Dies wird durch Setzen des Attributes **mixed** im **complexType**-Element angegeben. Im Unterschied zum „mixed“-Modell aus XML (siehe Seite 96) müssen Anzahl und Anordnung der Kindelemente mit der Spezifikation übereinstimmen, damit auch gemischte Elementinhalte validiert werden können.

Typen können in XML-Schema sowohl durch Erweiterung als auch durch Einschränkung bestehender Typen definiert werden. Das Ableiten eines neuen Typs durch Erweiterung eines bestehenden Typs entspricht der Vererbung in objektorientierten Programmiersprachen. Der Inhalt des abgeleiteten Typs enthält alle Elemente des ursprünglichen Typs und kann um weitere Elemente ergänzt werden. Wird ein Typ durch Einschränkung von einem anderen Typ abgeleitet, bilden die möglichen Instanzen des abgeleiteten Typs eine Teilmenge der möglichen Instanzen des Basistyps.

In objektorientierten Modellen werden häufig abstrakte Klassen verwendet, um gemeinsame Eigenschaften einer Reihe von konkreten Unterklassen in einer Oberklasse zusammenzufassen. Analog dazu können in einem XML-Schema abstrakte Elemente und Typen deklariert bzw. definiert werden. Abstrakte Elemente einer Ersetzungsgruppe werden in einem Instanz-

dokument durch ein konkretes Element derselben Ersetzungsgruppe ersetzt. Abstrakte Typen müssen durch einen abgeleiteten konkreten Typ ersetzt werden.

In XML-Schema kann die Erzeugung und Benutzung abgeleiteter Typen explizit beeinflusst werden. Durch Setzen des Attributes **final** auf einen der Werte „extension“, „restriction“ oder „#all“ in einer Typdefinition kann festgelegt werden, welche Art von Ableitungen erlaubt ist. Alternativ kann diese Festlegung auch global durch Setzen des Attributes **finalDefault** im **schema**-Element getroffen werden. Die Werte „extension“ und „restriction“ geben an, dass abgeleitete Typen nur durch Erweiterung bzw. nur durch Einschränkung des Basistyps erzeugt werden dürfen. „#all“ verbietet jegliche Form von Ableitungen. Neben **final** lassen sich weitere feinere Einschränkungen erlaubter Ableitungen durch das Attribut **fixed** definieren. Ein als **fixed** markiertes Element darf in einem abgeleiteten Typ nicht verändert werden. Mit Hilfe des Attributes **block** lassen sich auf Elementen Einschränkungen definieren, die sich analog zu den durch **final** definierten Einschränkungen auf Typen verhalten. Entsprechend gibt es auch ein Attribut **blockDefault** im **schema**-Element, das eine schemaweite Voreinstellung ermöglicht.

Neben einfachen und komplexen Typen können weitere Typen durch die Elemente **list** und **union** definiert werden. Ein **list**-Element definiert einen Listentyp, der nur von einem einfachen Datentyp abgeleitet werden darf und durch die Attribute **minLength** und **maxLength** oder alternativ durch das Attribut **length** in der Länge beschränkt werden kann. Durch ein **union**-Element kann eine Typalternative ausgedrückt werden. Im Unterschied zu ähnlichen Konstrukten in Programmiersprachen wie z. B. *C* dürfen nur einfache Typen verwendet werden. Ein Element, das einen **union**-Typ besitzt, ist konform zu genau einer der in dem **union**-Typ angegebenen Alternativen.

```
<xsd:simpleType name="LagerListe">
  <xsd:list itemType= "Waschmaschine"/>
</xsd:simpleType>

<xsd:simpleType name="LokaleLagerListe">
  <xsd:restriction base= "LagerListe">
    <xsd:minLength= "1"/>
    <xsd:minLength= "10" />
  </xsd:restriction>
</simpleType>

<xsd:simpleType name= "Schluessel" >
  <xsd:union memberType= "xsd:double xsd:string" />
</xsd:simpleType>
```

Abbildung 6.19: Listen und Vereinigungen

In Abb. 6.19 ist jeweils ein Beispiel für die Definition eines Listentyps bzw. eines Vereinigungstyps angegeben. Der Listentyp **LagerListe** ist eine Liste mit einer unbeschränkten Anzahl von Waschmaschinen. In einem abgeleiteten Typ **LokaleLagerListe** wird die zulässige Anzahl von Waschmaschinen auf einen Wert zwischen eins und zehn eingeschränkt. Der Vereinigungstyp

Schluessel definiert ein Merkmal zur Identifikation eines Elementes. Elemente dieses Typs sind entweder Zeichenketten oder Fließkommazahlen des Typs **xsd:double**.

In vielen Fällen benötigt man eine Möglichkeit, Elemente in einem Instanzdokument eindeutig zu identifizieren. Dazu werden Elemente und/oder Attribute mit der Eigenschaft versehen, innerhalb des Dokumentes eindeutig zu sein. Die Spezifikation der Eindeutigkeit erfolgt durch ein **unique**-Element in einem XML-Schema. Dieses wählt eine Anzahl von Elementen aus und bestimmt die Elemente und Attribute, die für diese Elemente eindeutig sein müssen. Dabei bezieht sich die Eindeutigkeit nicht auf einzelne Elemente oder Attribute, sondern auf die ausgewählte Menge. Analog zu **unique** lässt sich durch ein **key**-Element ein Schlüssel definieren, der durch **keyRef**-Attribute referenziert werden kann.

6.3 Zusammenfassung

In diesem Kapitel wurde die *Extensible Markup Language* (XML) beschrieben, deren Sprachumfang durch eine Spezifikation des *World Wide Web Committee* (W3C) [XML00] festgelegt wird. Ein XML-Dokument besteht aus einer Menge von XML-Elementen und Attributen, die hierarchisch angeordnet sind. Jedes XML-Dokument enthält genau ein Wurzelelement. Durch Verwendung von XML-Namensräumen [Nam99] können den Elementen und Attributen eindeutige Namen zugeordnet werden. Um Beziehungen zwischen den Elementen ausdrücken zu können, stellt die *XML Linking Language* (XLink) [XLi01] verschiedene Mechanismen zur Verfügung. Diese Mechanismen erlauben es, sowohl innerhalb eines Dokumentes als auch dokumentenübergreifend Elemente zu referenzieren.

XML definiert lediglich grundlegende Regeln, die die Struktur der einzelnen XML-Elemente und -Attribute festlegen. XML-basierter Datenaustausch benötigt daher weitere Mechanismen, um die in einem Dokument erlaubten Elemente und Attribute für bestimmte Anwendungsfälle zu spezifizieren. Zu diesem Zweck wurden zunächst die Dokumenttypdefinitionen (DTD) aus dem XML-Vorgänger *Standard Generalized Markup Language* (SGML) [Int86] verwendet. Da die Möglichkeiten der DTDs eingeschränkt sind und DTDs keine XML-Dokumente sind, wurde eine weitere Sprache, XML-Schema, definiert, um Klassen von XML-Dokumenten spezifizieren zu können.

Ein XML-Schema besteht im Wesentlichen aus Elementdeklarationen und Typdefinitionen. Die Elementdeklarationen legen fest, welche Elemente in einem konformen XML-Dokument auftreten dürfen. Typdefinitionen beschreiben die Struktur der Elemente. Sie können durch Ableitung aus den von XML-Schema bereitgestellten Basistypen spezifiziert werden. XML-Schema ermöglicht sowohl die Einschränkung der Wertemenge eines existierenden Typs als auch eine der Vererbung in objektorientierten Programmiersprachen ähnliche Erweiterung von Typen.

Kapitel 7

XML Metadata Interchange

Modellbasierte Entwicklungsprozesse durchlaufen in der Regel mehrere Phasen. Die meisten Entwicklungswerkzeuge unterstützen nur einzelne Entwicklungsphasen. Um trotzdem den vollständigen Entwicklungsprozess werkzeugunterstützt durchführen zu können, müssen daher mehrere Werkzeuge verwendet und die Modelldaten zwischen den Werkzeugen ausgetauscht werden. Da die meisten Entwicklungsprozesse zudem iterativ sind, reicht der Austausch zum jeweils die nächste Phase unterstützenden Werkzeug nicht aus. Ein ähnliches Problem tritt häufig bei organisationsübergreifenden Entwicklungsprozessen auf, weil unterschiedliche Organisationen meist auch verschiedene Werkzeuge einsetzen. In diesem Fall müssen zusätzlich die während einer Phase entstehenden Teilmodelle zwischen den Werkzeugen ausgetauscht werden.

Nahezu jedes verfügbare Modellierungswerkzeug speichert seine Daten in einem eigenen Format. Der Austausch von Modelldaten zwischen zwei Werkzeugen erfordert daher meist das Konvertieren von Daten vom Datenformat des exportierenden Werkzeuges in das Datenformat des importierenden Werkzeuges. Entwicklung und Wartung der entsprechenden Import- und Exportschnittstellen sind aufwändig und fehleranfällig. Aus diesem Grund hat die OMG ein konfigurierbares, standardbasiertes Datenformat, *XML Metadata Interchange* (XMI), entworfen und als Standard verabschiedet. XMI basiert auf der *Extensible Markup Language* (XML), die in Kapitel 6 beschrieben wurde.

Bisher[1] wurden vier Versionen der XMI-Spezifikation von der OMG verabschiedet. Die fünfte Version wird voraussichtlich zusammen mit den nächsten Versionen von MOF und UML standardisiert. Die verschiedenen Versionen der XMI-Spezifikation unterscheiden sich durch die MOF-Version, die sie abbilden, und die verwendeten Technologien. Die ersten beiden XMI-Versionen, [Obj00a] und [Obj00b] bilden die MOF-Version 1.3, [Obj02c] und [Obj03c] die MOF-Version 1.4, und [Obj03b] die MOF-Version 2.0 ab. Ab der Version 1.1 verwendet XMI die XML-Namensräume [Nam99] zur eindeutigen Bezeichnung der XML-Elemente und -Attribute. Die 1.x-Versionen der XMI-Spezifikation definieren DTDs, die den Aufbau der

[1] Stand: 24. September 2005

XMI-Dokumente festlegen. In den 2.x-Versionen wird zu diesem Zweck XML-Schema verwendet. In Tab. 7.1 werden die grundlegenden Eigenschaften der verschiedenen XMI-Versionen noch einmal zusammengefasst.

XMI-Version	MOF-Version	XML-Namensräume	DTD	XML-Schema
1.0	1.3	-	+	-
1.1	1.3	+	+	-
1.2	1.4	+	+	-
2.0	1.4	+	-	+
2.1	2.0	+	-	+

Tabelle 7.1: Eigenschaften der verschiedenen XMI-Versionen

Obwohl sich die verwendeten Standards zwischen verschiedenen XMI-Versionen unterscheiden, ist die grundsätzliche Funktion von XMI gleich geblieben. XMI definiert eine Abbildung zwischen MOF und XML. MOF-Modelle werden auf spezielle XML-Schemata abgebildet, deren Instanzdokumente die Instanzen der MOF-Modelle repräsentieren. Da XMI ein Austauschformat ist, gelten diese Beziehungen in beide Richtungen, sowohl von MOF nach XML als auch von XML nach MOF. In Abb. 7.1 sind die Zusammenhänge zwischen MOF und den Bestandteilen des XMI auf den verschiedenen Ebenen der Modellierung im Überblick dargestellt. Die linke Seite der Abbildung zeigt einige Ebenen der MOF-Metadatenarchitektur (siehe Abschnitt 5.1). Auf oberster Ebene befindet sich das MOF-Metamodell, das den Sprachumfang der MOF festlegt. Instanzen des MOF-Metamodells, also MOF-Modelle, sind beispielsweise das UML- [UP03b] oder das CWM-Metamodell [Obj03a]. Diese Modelle verwenden die durch das MOF-Metamodell festgelegten Modellierungselemente, um ihrerseits die Elemente einer Modellierungssprache festzulegen. Ein Sonderfall ist das MOF-Metamodell, weil es ebenfalls durch MOF-Modellierungselemente beschrieben wird. Aus diesem Grund ist das MOF-Metamodell auch ein MOF-Modell und in Abb. 7.1 zweimal dargestellt. Auf der untersten der dargestellten Ebenen befinden sich die Anwendermodelle, die in Modellierungssprachen definiert sind, deren Sprachumfang durch ein MOF-Modell festgelegt ist. Dies sind z. B. alle UML-Modelle.

Auf der rechten Seite der Abbildung ist die XML-Domäne dargestellt. Die Anordnung der Ebenen zeigt, welche Elemente der XML-Spezifikation zur Repräsentation der Modelle der verschiedenen MOF-Ebenen verwendet werden. Dies sind zum einen die XML-Schemata und zum anderen die XML-Dokumente, die Instanzdokumente eines XML-Schemas sein können. Jedes MOF-Modell kann durch ein XML-Schema beschrieben werden. Die Instanzdokumente dieses XML-Schemas repräsentieren die Modelle, die durch die Elemente des MOF-Modells beschrieben sind.

XMI schafft die Verbindung zwischen der MOF-Domäne und der XML-Domäne. Dies ist durch den Pfeil zwischen den beiden Domänendarstellungen angegeben. Die Pfeile zwischen den einzelnen Elementen der Domänen zeigen die durch XMI spezifizierten Abbildungen von MOF auf XML im Detail. Der gestrichelte Pfeil ist angegeben, um die Sonderstellung des MOF-Metamodells noch einmal hervorzuheben. Die Abbildung des MOF-Metamodells auf ein XML-

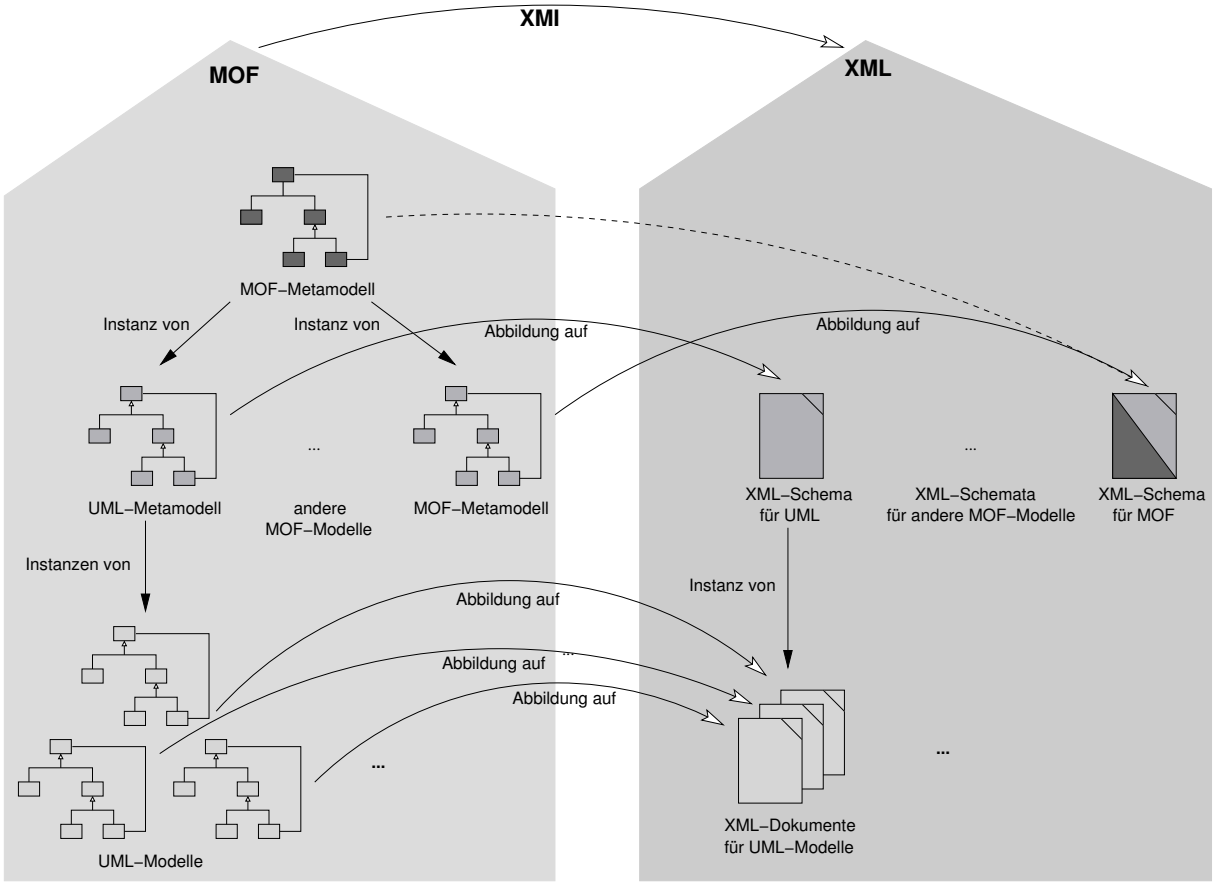


Abbildung 7.1: Beziehungen zwischen MOF und XML

Schema ist somit zweimal in der Abb. enthalten. Um zu kennzeichnen, dass sich das XML-Schema für MOF auf zwei verschiedenen Metaebenen einordnen lässt, ist es zweifarbig dargestellt.

In den folgenden Abschnitten werden die Bestandteile des XMI-Standards anhand des Beispiels aus Kapitel 4, Abschnitt 4.3 erläutert. In Abschnitt 7.1 wird dargestellt, wie aus einem MOF-Modell ein XMI-Schema erzeugt werden kann, und in Abschnitt 7.2 wird die Generierung eines XMI-Dokumentes für ein Modell beschrieben. Sowohl die Erzeugung von XMI-Schemata für ein MOF-Modell als auch die Erzeugung von Schnittstellen zum Austausch von XMI-Dokumenten kann durch den MOmo-MOF-Codegenerator durchgeführt werden. Abschnitt 7.3 fasst den Inhalt dieses Kapitels noch einmal kurz zusammen.

7.1 Schemata

Ein XMI-Schema ist eine Repräsentation eines MOF-Modells in XML-Schema. Das Schema gibt die Struktur der Elemente der Instanzdokumente vor, die wiederum Instanzen des MOF-Modells repräsentieren. Ein Beispiel dafür ist die UML, die eine Instanz der MOF ist. Die Bestandteile der UML können durch ein XMI-Schema repräsentiert werden. Dieses Schema definiert den Aufbau der Elemente und Attribute, die in einem XMI-Dokument ein UML-Modell repräsentieren.

Die Verwendung von XML-Schemata ermöglicht die Validierung der XML-Dokumente, die Instanzen eines MOF-Modells repräsentieren. Mit Hilfe einer geeigneten Konfiguration können die Möglichkeiten der Validierung erhöht werden. Der wesentliche Vorteil der Verwendung von XML-Schemata zur Darstellung von MOF-Modellen ist, dass verfügbare XML-Werkzeuge zur Validierung von Dokumenten genutzt werden können. Dies kann den Implementierungsaufwand für eine XMI-Schnittstelle erheblich verringern, weil durch Validierung überprüft werden kann, ob alle von XMI verlangten Elemente in einem Dokument vorhanden sind, und ob die notwendigen Attribute mit korrekten Werten belegt sind.

Bis zu einem bestimmten Maße kann durch die Validierung eines Instanzdokuments anhand eines XMI-Schemas überprüft werden, ob das repräsentierte Modell eine Instanz des Metamodells ist, das durch das Schema dargestellt wird. Eine vollständige Verifikation ist allerdings nicht möglich, weil die Semantik eines MOF-Modells nicht vollständig in XMI-Schema ausgedrückt werden kann. Dies ist wiederum in der automatischen Erzeugung von XMI-Schemata begründet, die die Verwendung einiger Elemente aus XML-Schema nicht ermöglicht.

Ein XMI-konformes Schema für ein MOF-Modell kann automatisch erzeugt werden. Dazu gibt die XMI-Spezifikation einen Satz von Produktionsregeln an. Diese legen fest, wie die Elemente des Schemas aus den Informationen des MOF-Modells generiert werden können. Durch Angabe von Tags, die an einzelne Modellelemente eines MOF-Modells „angeheftet“ werden, kann die Erzeugung eines XMI-Schemas auf einen bestimmten Einsatzzweck zugeschnitten werden.

7.1.1 Aufbau

In Kapitel 3 der XMI 2.1 Spezifikation werden Produktionsregeln zur Erzeugung eines XMI-konformen XML-Schemas für ein MOF-konformes Modell definiert. Die Regeln sind in einer an die *Erweiterte Backus-Naur-Form* (EBNF) angelehnten Syntax angegeben. Die Definition der Produktionsregeln ist in acht Abschnitte aufgeteilt, die die verschiedenen Bestandteile eines MOF-Modells auf XML-Schema-Elemente abbilden. Die grundlegende Struktur eines XMI-Schema-Dokumentes besteht aus dem Wurzelknoten **schema**, der Attribute enthält, die die innerhalb des Schemas verwendeten Namensräume deklarieren. Es müssen mindestens zwei Namensräume, je einer für XMI- und XML-Schema, deklariert werden. In der Spezifikation und im Folgenden werden diese mit **xsd** bzw. **xmi** bezeichnet; es können aber auch andere Bezeichner vereinbart werden. Neben diesen beiden vorgeschriebenen Namensräumen darf noch eine beliebige Anzahl weiterer Namensräume und ein Ziel-Namensraum (siehe Abschnitt 6.1.2) deklariert werden.

Unterhalb des Wurzelknotens sind die Schemaelemente angeordnet, um andere Namensräume zu importierten bzw. einzuschließen. Jedes XMI-Schema muss die in der XMI-Spezifikation festgelegten XML-Elemente und -Attribute importieren. Diese sind in der XMI-Spezifikation mit dem Ziel-Namensraum **http://schema.omg.org/XMI/2.1** angegeben. Nachdem damit alle zur Repräsentation eines MOF-Modells benötigten Informationen definiert sind, erfolgt die Repräsentation der Elemente des MOF-Modells. Diese besteht auf der obersten Ebene aus einer Reihe allgemein festgelegter Deklarationen und den Repräsentationen der äußersten Pakete des MOF-Modells.

Die festgelegten Deklarationen ermöglichen es, Objekte zu identifizieren und zu referenzieren. In der Grundeinstellung werden das Attribut **xmi:id** und die Attributgruppe **xmi:ObjectAttribs** deklariert. Durch die Konfigurationsvariable **idName** kann anstelle von **xmi.id** ein anderes Attribut des Typs **xsd:ID** zur Identifikation von Objekten deklariert werden. Dieses Attribut erhält den durch **idName** definierten Namen. Die Attributgruppe **xmi:ObjectRefs** setzt sich aus den Attributgruppen **xmi:IdentityObjects** und **xmi:LinksAttribs** sowie den Attributen **version** und **type** zusammen. **xmi:IdentityObject** enthält die Attribute **xmi:label** und **xmi:uuid**, die zur Identifikation von Objekten verwendet werden können. Die Attributgruppe **LinkAttribs** besteht aus den Attributen **href** und **idref**, die das Referenzieren von Objekten ermöglichen. Die Attribute **version** und **type** ermöglichen Versionierung und Typangabe für ein Objekt.

```
<xsd:schema
  xmlns:xsd = 'http://www.w3.org/2001/XMLSchema'
  xmlns:xmi = 'http://schema.omg.org/XMI/2.1'
  xmlns:mcl = 'http://ist.unibw-muenchen.de/mcl'
  targetNamespace= 'http://ist.unibw-muenchen.de/mcl' >

  <import schemaLocation= 'http://schema.omg.org/XMI/2.1' />

  Bestandteile der Sprache

</xsd:schema>
```

Abbildung 7.2: Aufbau eines XMI-Schemas

In Abb. 7.2 ist der grundsätzliche Aufbau eines XMI-Schemas anhand des Beispiels aus Abschnitt 4.3 dargestellt. Im **schema**-Knoten werden die Namensräume **xsd**, **xmi** und **mcl** deklariert. Zusätzlich wird **mcl** zum Ziel-Namensraum erklärt, so dass die Elemente der MCL innerhalb des Schemas ohne Präfix benannt werden können. Unterhalb des **schema**-Knotens werden der verlangte Import der festgelegten Elemente aus der XMI-Spezifikation sowie die Repräsentationen der Elemente der MCL angeordnet.

7.1.1.1 Pakete

Die Repräsentation eines MOF-Paketes wird erzeugt, indem zunächst Schemarepräsentationen für jedes enthaltene Paket, jede enthaltene Klasse und jeden enthaltenen Aufzählungstyp erzeugt werden. Anschließend wird eine Elementdefinition für das Paket selbst erzeugt. Die Elementdefinition für ein MOF-Paket besteht aus einem Schemaelement, dessen Attribut **name** mit dem Namen des Paketes belegt ist. Der Aufbau der Instanzen dieses Elementes wird durch einen anonymen komplexen Typ (**xsd:complexType**) vorgegeben. Die Elemente des Paketes werden innerhalb einer Auswahl (**xsd:choice**) angegeben.

Für jede Klasse, die innerhalb eines Paketes definiert ist, wird ein XML-Element erzeugt. Entsprechende XML-Elemente besitzen keinen Inhalt, sondern nur ein Attribut **ref**, das die enthaltene Klasse über Namensraum und Name referenziert. Für jedes enthaltene Paket wird analog verfahren. Die enthaltenen Assoziationen werden dagegen durch Assoziationsdefinitionen repräsentiert, die durch einen gesonderten Satz von Produktionsregeln erzeugt werden. Neben den vom definierten Inhalt eines Paketes abhängigen Elementen werden einige konstante Attributdefinitionen erzeugt, die Identifikation und Referenzierung von XML-Elementen ermöglichen. In Abb. 7.3 ist die Elementdefinition für das Paket MCL aus Abb. 4.8 dargestellt.

```
<xsd:element name= 'MCL' >
  <xsd:complexType>
    <xsd:choice minOccurs='0' maxOccurs= 'unbounded'>
      <xsd:element ref= 'ModelElement'/>
      <xsd:element ref= 'Component'/>
      <xsd:element ref= 'Connection'/>
      <xsd:element ref= 'ExportInterface'/>
      <xsd:element ref= 'ImportInterface'/>
      <xsd:element ref= 'xmi.extension' />
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Abbildung 7.3: Repräsentation des Paketes **MCL** in XMI-Schema

Entsprechend der angegebenen Definition wird ein XML-Element **MCL** deklariert, dessen Aufbau durch einen anonymen komplexen Typ definiert wird. Dieser wiederum besteht aus einer Auswahl, die Referenzen auf die im Paket **MCL** enthaltenen Klassen **ModelElement**, **Component** und **Connection**, **ExportInterface** und **ExportInterface** enthält.

7.1.1.2 Klassen

Der komplexeste Teil der XMI-Abbildung von MOF-Modellen auf XML-Schemata ist die Repräsentation von Klassen. Dies liegt an der Menge der Konfigurationsmöglichkeiten, die angegeben werden können, um die Ausprägungen der generierten Repräsentationen auf verschiedene Anwendungsfälle anzupassen. Dazu werden an die Elemente eines MOF-Modells sog. *Tags*

angehängt, die den Wert einer Konfigurationsvariable festlegen. Grundsätzlich wird jede Klasse eines MOF-Modells in einem XMI-Schema durch eine Typdefinition und eine Elementdeklaration repräsentiert. Die Elementdeklaration besteht aus einem XML-Element mit den Attributen **name** und **type**. Das Attribut **name** enthält den Namen der Klasse, und das Attribut **type** enthält eine Referenz auf die Typdefinition der Klasse. Abb. 7.4 zeigt die Elementdeklaration für die Klasse **Component** aus Abb. 4.8.

```
<xsd:element name='Component' type='Component'/>
```

Abbildung 7.4: Elementdeklaration für die Klasse **Component**

Wie in Abb. 7.4 dargestellt, referenziert jede Elementdeklaration eine gleichnamige Typdefinition. Diese ist ein komplexer Typ, dessen Struktur durch die Konfigurationsvariablen **useSchemaExtension**, **enforceMinimumMultiplicity**, **enforceMaximumMultiplicity** und **contentType** konfiguriert werden kann.

Die Belegung der Variable **useSchemaExtensions** entscheidet, ob der Vererbungsmechanismus aus XML-Schema verwendet wird. In der Grundeinstellung werden Vererbungsbeziehungen durch Kopieren der Elemente und Attribute der Superklassen in die Typdefinition der abgeleiteten Klasse abgebildet, weil der Vererbungsmechanismus aus XML-Schema im Unterschied zu MOF keine Mehrfachvererbung unterstützt. Der Aufbau der Typdefinition besteht aus einem **complexType**-Knoten, der einen **choice**- oder einen **sequence**-Knoten enthält. Innerhalb des **choice**- bzw. **sequence**-Knotens werden Elementdeklarationen für jede Eigenschaft der Klasse erzeugt. Abb. 7.5 zeigt einen Ausschnitt der Repräsentation der Klasse **Component** aus Abb. 4.8.

```
<xsd:complexType name='Component'>
  <xsd:choice minOccurs='0' maxOccurs='unbounded'>
    <xsd:element name='name' type='xsd:string'/>
    <xsd:element name='whole' type='xmi:Any'/>
    <xsd:element name='part'type='xmi:Any'/>
  </xsd:choice>
  ...
</xsd:complexType>
```

Abbildung 7.5: Abbildung einer Vererbungsbeziehung durch Kopieren

Die Klasse **Component** erbt das Attribut **name** und die Refernz **whole** von der Klasse **ModelElement**. Das Attribut und die Referenz werden zusammen mit der lokal definierten Referenz **part** in einer Auswahl angeordnet. Diese Auswahl darf beliebig oft auftreten, so dass in den Elementdeklarationen keine weiteren Angaben über Multiplizitäten benötigt werden. Da das Schema alle Vererbungsbeziehungen durch Kopieren simuliert, müssen die Typen aller Attribute, deren Typ eine Klasse ist, und aller Referenzen auf **xmi:Any** abgebildet werden. Sonst könnte in den Instanzdokumenten keine abgeleitete Klasse anstelle einer Basisklasse verwendet werden.

Der in Abb. 7.5 dargestellte Aufbau einer Klassenrepräsentation in einem XMI-Schema bildet Multiplizitäten nicht ab. Um die Abbildung von Multiplizitäten zu erzwingen, muss eine der Konfigurationsvariablen **enforceMinimumMultiplicity** oder **enforceMaximumMultiplicity** mit dem Wert „true“ angegeben werden. In diesem Fall wird eine Reihung anstatt einer Auswahl erzeugt. Im Unterschied zur Auswahl wird die Reihung ohne Angabe einer Multiplizität erzeugt, so dass die aufgeführten Elemente nur einmal auftreten dürfen. Zudem ist die Reihenfolge des Auftretens der Elemente in den Instanzdokumenten ebenfalls festgelegt. Sie muss der im Schema angegeben Reihenfolge entsprechen. Für jedes der in der Reihung enthaltenen Elemente muss durch die Attribute **minOccurs** und **maxOccurs** die Multiplizität spezifiziert werden. Dies entspricht den Elementdeklarationen bei Verwendung des Vererbungsmechanismus' aus XML-Schema (siehe Abb. 7.6).

Wird **useSchemaExtension** definiert, wird der Vererbungsmechanismus aus XML-Schema verwendet. In diesem Fall darf das MOF-Modell keine Mehrfachvererbungen enthalten, weil diese von XML-Schema nicht unterstützt werden. In der MCL wird die Klasse **Component** durch einfache Vererbung von der Klasse **ModelElement** abgeleitet. In Abb. 7.6 wird dargestellt, wie die entsprechende Schemarepräsentation unter Verwendung des Mechanismus aus XML-Schema aussieht.

```
<xsd:complexType name='Component'>
  <xsd:complexContent>
    <xsd:extension base='ModelElement'>
      <xsd:element name='part' type='ModelElement'/>
                   minOccurs='0' maxOccurs= 'unbounded'/>
      ...
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Abbildung 7.6: Vererbung durch XML-Schema-Mechanismus

Alle lokal definierten Attribute werden unterhalb eines **xsd:extension**-Knotens aufgeführt. Da hier die Informationen über die Vererbungsbeziehungen explizit zur Verfügung stehen, können die Typen von Attributen, deren Typ eine Klasse ist, und Referenzen direkt abgebildet werden. Außerdem müssen in den Elementdeklarationen Angaben über die Multiplizität gemacht werden. Diese wird analog zur Darstellung der Eigenschaften in einer Reihung durch die Attribute **minOccurs** und **maxOccurs** für jedes Element angegeben.

Eine weitere Möglichkeit zur Beeinflussung der Repräsentation einer Klasse ist die Konfigurationsvariable **contentType**. Diese Variable kann auf einen der Werte „mixed“, „complex“, „any“ oder „empty“ gesetzt werden und wirkt sich auf den Inhalt der Auswahl, Reihung oder Erweiterung aus. Wenn **contentType** den Wert „complex“ oder „mixed“ besitzt, werden die Modellelemente, die innerhalb des Namensraumes der Klasse definiert sind, explizit durch XML-Elemente repräsentiert. Dies entspricht den Darstellungen in den Abb. 7.6 und 7.5. Der Unterschied zwischen „complex“ und „mixed“ ist, dass bei der Angabe von „mixed“ die XML-Elemente mit Textelementen gemischt auftreten dürfen, und dies durch das Attribut „mixed“ des

complexType-Knotens gekennzeichnet wird. Ist die Variable **contentType** als „any“ definiert, wird lediglich ein **xsd:any**-Knoten erzeugt, so dass beliebige Elemente in beliebiger Reihenfolge innerhalb einer Instanz der Klasse stehen dürfen. Hat **contentType** den Wert „empty“, dürfen keine Elemente für die Eigenschaften einer Klasse erzeugt werden. Dies ist die Grundeinstellung.

Die Konfigurationsvariable **isNillable** wirkt sich ausschließlich auf die Erzeugung der Attributrepräsentationen aus. Wenn **isNillable** den Wert „true“ besitzt, wird in den XML-Elementen das Attribut **nillable** erzeugt und auf **true** gesetzt.

Wie in den Abb. 7.6 und 7.5 dargestellt ist, wird der Typ einer Eigenschaft durch das Attribut **type** angegeben. Dieses wird mit **xsd:string** für alle einfachen Datentypen belegt. Für alle Eigenschaften, deren Typ ein Enumerationstyp ist, wird **type** mit dem Namen des Enumerationstyps belegt. Ist der Typ einer Eigenschaft dagegen eine Klasse, wird in der Grundeinstellung **xmi.any** und bei Verwendung des Vererbungsmechanismus' aus XML-Schema der Name der Klasse verwendet. Die Typangabe in einem XMI-Schema kann durch die Konfigurationsvariable **schemaType** beeinflusst werden. Wenn **schemaType** einen von „nil“ verschiedenen Wert hat, wird dieser anstelle der Typangabe aus dem MOF-Modell verwendet.

In der Grundeinstellung werden neben den Elementen zur Repräsentation der Eigenschaften einer Klasse auch XML-Attributdeklarationen für jede lokal definierte Eigenschaft der Klasse erzeugt. Für jede Eigenschaft, die keine Komposition ist, wird ein XML-Attribut deklariert, dessen **name**-Attribut den Namen der Eigenschaft erhält. Die weiteren Attribute der Deklaration sind abhängig von Typ und Multiplizität der Eigenschaft. Wenn der Typ einer Eigenschaft eine Klasse ist, ist zudem der verwendete Vererbungsmechanismus von Bedeutung. Wird der Vererbungsmechanismus von XML-Schema verwendet, wird das Attribut **type** mit dem Namen der Klasse belegt. Andernfalls werden die allgemeinen Referenzen aus XML-Schema verwendet. Abb. 7.7 zeigt die Darstellung der Klasse **Component**, die ein XML-Schema-Generator in der Grundeinstellung erzeugt.

```
<xsd:complexType name='Component'>
  <xsd:choice minOccurs='0' maxOccurs='unbounded'>
    <xsd:element name='name' type='xsd:string'/>
    <xsd:element name='component' type='xmi:Any'/>
    <xsd:element name='whole' type='xmi:Any'/>
    <xsd:element name='part'type='xmi:Any'/>
    <xsd:element ref='xmi:extension'/>
  </xsd:choice>
  <xsd:attribute ref='xmi:id' use='optional'/>
  <xsd:attributeGroup ref='xsd:ObjectAttribs'/>
  <xsd:attribute name='name' type='xsd:string'/>
  <xsd:attribute name='whole' type='xsd:IDREFS'/>
</xsd:complexType>
```

Abbildung 7.7: Repräsentation der Klasse **Component** in der Grundeinstellung

Neben den Attributdefinitionen für die Eigenschaften aus dem MOF-Modell werden die Attribute zur Identifikation und Referenzierung erzeugt. Die Instanzen der Klasse können in den Instanzdokumenten anhand des Attributes **xmi:id** unterschieden werden, dessen Typ **xsd:ID** ist. Für das Attribut **name** aus der Basisklasse **ModelElement** wird ein Attribut erzeugt, dessen Typ **xsd:string** ist. Um die Referenz *whole* zu repräsentieren, wird ein Attribut des Typs **xsd:IDREFS** erzeugt. Die Aufführung dieses Attributes ist optional, weil nicht jede Komponente ein Teil einer anderen Komponente ist. Die weiteren Referenzen aus der Klasse **Component** werden nicht durch Attributdeklarationen ausgedrückt, weil sie mehrfach auftreten können.

Wenn der Vererbungsmechanismus aus XML-Schema verwendet wird, werden nur die lokalen Attributdeklarationen erzeugt. In diesem Fall wird die Klasse **Component** durch den in Abb. 7.8 dargestellten Typ repräsentiert.

```
<xsd:complexType name='Component'>
  <xsd:complexContent>
    <xsd:extension base='ModelElement'>
      <xsd:element name='part' type='Component'/>
                   minOccurs='0' maxOccurs='unbounded'/>
      <xsd:element ref='xmi.extension'/>
      <xsd:attribute ref='xmi:id' use='optional'/>
    </xsd:extension>
  </xsd:complexContent>                                  />
</xsd:complexType>
```

Abbildung 7.8: Vollständiger Typ der Klasse **Component** mit XML-Schema-Vererbung

Sowohl bei Verwendung des eingebauten Vererbungsmechanismus' als auch bei der Realisierung von Vererbungsbeziehungen durch Kopieren werden Eigenschaften einer Klasse gleichzeitig durch XML-Elemente und -Attribute repräsentiert. Beispiele dafür sind die Eigenschaften **name** und **whole** der Klasse **Component** in Abb. 7.7. Um die Erzeugung überflüssiger XML-Elemente oder -Attribute zu vermeiden, können die Konfigurationsvariablen **element** und **attribute** verwendet werden. Die Angabe von **element** erzwingt die Repräsentation eines Modellelementes durch ein XML-Element. Analog erzwingt die Angabe von **attribute** die Darstellung durch ein XML-Attribut. Insbesondere die Variable **attribute** unterliegt diversen Einschränkungen, um zu vermeiden, dass die Darstellung eines Modellelementes durch ein XML-Attribut erzwungen wird, wenn dies nicht möglich ist.

7.1.1.3 Assoziationen

Die Definition einer Assoziation in einem XMI-Schema erfolgt durch ein XML-Element, dessen Attribut **name** mit dem Namen der Assoziation belegt wird. Die Struktur des Elementes wird durch einen anonymen, komplexen Typ bestimmt. Dieser Typ enthält die Definitionen der Assoziationsenden und ein „Erweiterungselement“. Für jedes Assoziationsende wird ein XML-Element generiert. Da MOF bislang nur binäre Assoziationen erlaubt, sind immer genau zwei XML-Elemente enthalten. Die Definitionen der Assoziationsenden werden durch einen komplexen Typ bestimmt, der ausschließlich konstante Attributdefinitionen zur Identifikation und Referenzierung enthält. Dieselben Informationen sind auch innerhalb der Definition des Assoziationselementes vorhanden. In Abb. 7.9 ist die Assoziation zwischen den Klassen **ExportInterface** und **Connection** aus Abb. 4.8 dargestellt.

```
<xsd:element name='export_connection'>
  <xsd:complexType>
    <xsd:choice minOccurs='0' maxOccurs='unbounded'>
      <xsd:element name='export'/>
        <xsd:complexType>
          <xsd:attribute name='xmi:id' use='optional'/>
          <xsd:attributeGroup ref='xmi:ObjectAttribs'/>
        </xsd:complexType>
       </xsd:element>
       <xsd:element name='connection'/>
         <xsd:complexType>
           <xsd:attribute name='xmi:id' use='optional'/>
           <xsd:attributeGroup ref='xmi:ObjectAttribs'/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element ref='xmi:extension'/>
    </xsd:choice>
  </complexType>
</xsd:element>
```

Abbildung 7.9: Definition einer Assoziation

Die Assoziation enthält die Assoziationsenden *export* und *connection*. Für jedes Assoziationsende wird ein XML-Element erzeugt, das mit dem Namen des Assoziationsendes benannt wird. Der Typ eines Attributes enthält die in XMI festgelegten Attribute zur Identifikation und Referenzierung von Objekten, die weiter oben bereits beschrieben wurden. Die Typdefinition der hier betrachteten Assoziation besteht daher aus den beiden Elementen, die die Assoziationsenden *export* und *connection* repräsentieren, und dem fest definierten XMI Erweiterungselement.

7.1.1.4 Datentypen

MOF enthält einfache Datentypen, Aufzählungstypen und strukturierte Datentypen. Alle einfachen Datentypen werden in XMI-Schemata auf den vordefinierten Datentyp **xsd:string** abgebildet. Für die Konvertierung der Werte zwischen dem tatsächlichen Datentyp und der XMI-Darstellung ist die Schnittstelle zwischen Modell und XMI-Repräsentation zuständig. Die Darstellung strukturierter Datentypen erfolgt analog zur Repräsentation von Klassen. Für jedes Feld wird ein Attribut erzeugt, das den Typ des entsprechenden Feldes besitzt.

Für jeden Aufzählungstyp wird ein einfacher Typ (**xsd:simpleType**) generiert, der durch Einschränkung vom vordefinierten Datentyp **xsd:string** abgeleitet wird. Für jeden möglichen Wert des Datentyps wird ein Aufzählungswert (**xsd:enumeration**) erzeugt, dessen Attribut **value** durch das entsprechende Literal belegt wird. In Abb. 7.10 ist die Schemarepräsentation des Aufzählungstyps **PackageMergeKind** aus Abb. 5.12 dargestellt.

```
<xsd:simpleType name='PackageMergeKind'>
  <xsd:enumeration value='extend'/>
  <xsd:enumeration value='define'/>
</xsd:simpleType>
```

Abbildung 7.10: Definition des Aufzählungstyps **PackageMergeKind**

Der Aufzählungstyp **PackageMergeKind** enthält die Literale **extend** und **define**, so dass ein abgeleiteter einfacher Typ generiert wird, der den vordefinierten Datentyp **xsd:string** auf genau diese beiden Zeichenketten einschränkt.

7.2 Dokumente

Ein XMI-Dokument repräsentiert ein Modell, das aus MOF-konformen Modellelementen zusammengesetzt ist. Aufbau und Bedeutung der Modellelemente werden durch ein Metamodell beschrieben, das eine Instanz der MOF ist. Ein solches Metamodell enthält daher nur die in der MOF-Spezifikation festgelegten Modellelemente. In den folgenden Abschnitten wird zunächst der Aufbau eines XMI-Dokumentes beschrieben (Abschnitt 7.2.1). Anschließend wird dargestellt, wie ein XMI-Dokument für ein gegebenes Modell durch die in der XMI-Spezifikation angegebenen Produktionsregeln erzeugt werden kann (Abschnitt 7.2.2).

7.2.1 Aufbau

Ein XMI-Dokument besteht aus festgelegten Abschnitten, die für alle XMI-Dokumente gleich aufgebaut sind, und modellspezifischen Abschnitten. Alle Abschnitte sind unterhalb des Wurzelknotens **XMI** angeordnet oder werden durch Attribute des Wurzelknotens ausgedrückt. Der Aufbau der Elemente in den modellspezifischen Abschnitten wird durch das MOF-Modell bestimmt, das die verwendbaren Modellierungselemente beschreibt. Die Abbildung der Elemente eines Modells auf XML wird in Abschnitt 7.2.2 beschrieben. Die festgelegten Abschnitte

eines XMI-Dokumentes repräsentieren XMI-spezifische Informationen. Diese umfassen eine Versionsangabe, eine Beschreibung des enthaltenen Modells, Erweiterungen und Veränderungen gegenüber einem anderen XMI-Dokument. In Abb. 7.11 ist der prinzipielle Aufbau eines XMI-Dokumentes dargestellt.

```
<xmi:XMI version='2.1'
         xmlns:xmi = 'http://www.omg.org/XMI' >

   <xmi:Documentation>
     Dokumentation des Modells
   </xmi:Documentation>

   <xmi:Difference>
     Änderungen gegenüber einem anderen Modell
   </xmi:Difference>

   <xmi:Extension>
     Werkzeugabhängige Erweiterungen
   </xmi:Extension>

   Repräsentation des Modells

</xmi:XMI>
```

Abbildung 7.11: Aufbau eines XMI-Dokumentes

Der **XMI**-Knoten markiert die Wurzel des XMI-Dokumentes. Dies ist immer dann der Fall, wenn das repräsentierte Modell mehr als ein Element auf der obersten Ebene enthält, und das XMI-Dokument nicht in ein anderes XML-Dokument eingebettet ist. Innerhalb des **XMI**-Knotens wird der Namensraum **xmi** deklariert, um die Bestandteile eines XMI-Dokumentes verwenden zu können. Außerdem erfolgt die Angabe der Version der XMI-Spezifikation, die zur Erzeugung des Dokumentes verwendet wurde. Unterhalb des **XMI**-Knotens kann eine Beschreibung des Dokumentes erfolgen, die übliche Angaben wie Autor, Werkzeug, Erstellungsdatum usw. enthält.

XMI-Dokumente müssen keine vollständigen Modelle repräsentieren, sondern können auch lediglich Differenzen zu anderen XMI-Dokumenten enthalten. Diese Differenzen spiegeln Veränderungen des repräsentierten Modells gegenüber einer Vorgängerversion wieder. In Abb. 7.11 ist der Knoten **xmi:Difference** kursiv dargestellt, weil er nicht in dieser Form in einem XMI-Dokument vorkommt. Stattdessen können XMI-Dokumente **xmi:Add**-, **xmi:Delete**- und **xmi:Replace**-Knoten enthalten, die Ergänzungen, Löschungen und Ersetzungen in dem Basisdokument ausdrücken.

Da XMI zum werkzeugübergreifenden Datenaustausch verwendet werden soll, müssen werkzeugspezifische Informationen in einem XMI-Dokument abgelegt werden können. Jedes Werkzeug wertet nur seine eigenen Erweiterungen aus und lässt die von anderen Werkzeugen eingefügten Erweiterungen unberührt. Eine Erweiterung wird durch einen **xmi:Extension**-Knoten angegeben und kann beliebigen Inhalt enthalten.

7.2.2 Produktion

Der wichtigste Teil eines XMI-Dokumentes ist die Repräsentation der Elemente eines Modells. Die XMI-Spezifikation [Obj03b] spezifiziert eine Menge von Produktionsregeln, die die Erzeugung eines XMI-Dokumentes für ein Modell beschreiben. Die Produktionsregeln sind in zwei Abschnitte aufgeteilt. Der erste Abschnitt beschreibt die generelle Struktur eines XMI-konformen Dokumentes. Der zweite Abschnitt beschreibt den Aufbau der Objekte eines XMI-Dokumentes, d. h. die Erzeugung der modellspezifischen Anteile. Im Folgenden wird für dieses Beispiel durch Anwendung der Produktionsregeln aus Kapitel 4 der XMI-Spezifikation ein XMI-Dokument erzeugt.

Ein XML-Dokument muss genau ein Wurzelelement besitzen, um gemäß der XML-Spezifikation [XML00] *wohlgeformt* zu sein. Aus diesem Grund beginnt ein XMI-Dokument entweder mit einem speziellen Wurzelelement, oder es enthält nur genau ein Objekt. Wenn ein spezielles Wurzelelement verwendet wird, dann setzt sich dessen Name aus einem optionalen und frei definierbaren Präfix und der Zeichenkette „XMI“ zusammen. Das Startelement des Wurzelknotens enthält Angaben über die Version des verwendeten XMI-Standards und die innerhalb des Dokumentes verwendeten Namensräume. Bezogen auf das MCL-Beispiel bedeutet dies, dass der Wurzelknoten des generierten XMI-Dokumentes den Namen „xmi:XMI“ erhält, die Version mit „2.1“ angegeben wird, und der Namensraum „mcl“ deklariert wird, um die verwendeten Objekte eindeutig kennzeichnen zu können. Abb. 7.12 zeigt den entsprechenden Wurzelknoten. Die Objekte des Modells werden durch Kindobjekte des Wurzelknotens repräsentiert. Dies ist durch „...“ angedeutet.

```
<xmi:XMI version='2.1'
         xmlns:xmi = 'http://www.omg.org/XMI'
         xmlns:xmi = 'http://ist.unibwm.de/mcl' >

    ...

</xmi:XMI>
```

Abbildung 7.12: Wurzelknoten des Beispieldokumentes

Alle Elemente eines Modells werden in XMI durch XML-Elemente und -Attribute dargestellt. Einer der wichtigsten Mechanismen ist die Vergabe von Namen für die XMI-Elemente. Ein *XMI-Name* setzt sich aus mehreren Bestandteilen zusammen, die abhängig von der Art des darzustellenden Objektes kombiniert werden. Handelt es sich bei dem darzustellenden Objekt um ein Objekt der obersten Ebene, ist der XMI-Name in der Voreinstellung der Namensraum, in dem das Objekt definiert worden ist, gefolgt von einem Doppelpunkt und dem Namen des Objekttyps.

In Abb. 4.9 ist die Komponente **Waschmaschine** das einzige Objekt auf der obersten Ebene. Die XMI-Repräsentation dieses Objektes erfolgt durch ein XML-Element, dessen Name „mcl:Component“ ist. „mcl“ist der Namensraum, in dem die Klasse „Component“ definiert ist. Dieser wird mit dem Namen der Klasse durch einen Doppelpunkt verbunden. Für alle Objekte, die sich nicht auf der obersten Ebene befinden, wird der Namensraum ignoriert und nur der

Typname zur Bezeichnung des XML-Elementes verwendet. Für alle Objekte kann der XMI-Name durch die Konfigurationsvariable **xmiName** verändert werden. Wenn diese Variable für ein Objekt gesetzt wird, wird ihr Wert anstelle des Typnamens zur Erzeugung des XMI-Namens verwendet.

Objekte werden in XMI-Dokumenten durch einen XML-Knoten dargestellt, der gemäß der XML-Spezifikation aufgebaut ist (siehe Kapitel 6). Die Eigenschaften eines Objektes werden durch XML-Attribute dargestellt. Die XMI-Spezifikation unterteilt die möglichen Attribute in vier Gruppen. Die erste Gruppe bilden die Attribute, die bereits im Wurzelknoten vorhanden sind. Diese geben XMI-Version und verwendete Namensräume an. Sie sind für Objekte optional und werden nur dann angegeben, wenn ein XMI-Dokument nur ein Objekt und keinen speziellen Wurzelknoten enthält. Die zweite Gruppe von Attributen enthält Attribute, die zur Identifikation von XML-Elementen verwendet werden können. Mitglieder dieser Gruppe sind „id“, „label“ und „uuid“ die von den verschiedenen Referenzierungsmechanismen für XML benötigt werden. Auch die Erzeugung von Attributen dieser Gruppe ist optional. Außerdem dürfen auch mehrere Identifikationsattribute für ein Objekt generiert werden.

Die dritte Gruppe von Attributen enthält lediglich das Attribut „type“, das zur eindeutigen Identifizierung des Typs eines Objektes dient. Dieses Attribut muss immer dann erzeugt werden, wenn der Typ eines Objektes nicht eindeutig aus dem Modell hergeleitet werden kann. Der Wert des „type“Attributes wird durch den ersten Teil der XML-Schema Spezifikation [XML01b] bestimmt und muss ein qualifizierter Name sein. Ein qualifizierter Name besteht aus dem Namensraum für den Typ des Objektes, falls dieser nicht der voreingestellte Namensraum ist, einem Doppelpunkt und dem Namen des Objekttyps. In der vierten Gruppe von Attributen befinden sich die Attribute, die Eigenschaften des Objektes repräsentieren. Jedes Attribut, dessen Datentyp ein Datentyp oder eine Aufzählung ist, und jede Referenz, deren Typ ein Objekt innerhalb desselben Dokumentes ist, können durch ein XML-Attribut dargestellt werden.

```
<mcl:Component xmi.id='1' name='Waschmaschine' whole='nil'>
  Inhalt
</mcl:Component>
```

Abbildung 7.13: Repräsentation von Eigenschaften durch XML-Attribute

Abb. 7.13 zeigt den Knoten innerhalb des XMI-Dokumentes für unser Beispiel, der die Komponente *Waschmaschine* darstellt. Es wird ein Attribut zur Identifikation des Knotens erzeugt, weil dieser der Container für alle anderen Modellelemente des Beispiels ist und diese ihn daher referenzieren können müssen. Außerdem wird das von der Klasse *ModelElement* geerbte Attribut *name* erzeugt und mit „Waschmaschine“ belegt. Gemäß der XMI-Schema-Definition für das Metamodellelement *Component* muss außerdem noch ein XML-Attribut für die Referenz *whole* erzeugt werden. Da die Komponente *Waschmaschine* auf oberster Ebene angeordnet ist, wird das entsprechende Attribut mit dem Wert „nil“ belegt. Durch Setzen der Konfigurationsvariable **includeNils** auf den Wert „false“ kann die Erzeugung von XML-Repräsentationen für „nil“-wertige Attribute unterdrückt werden.

Jede Eigenschaft eines Objektes wird durch ein XML-Element dargestellt. Auch für jedes Attribut eines Objektes wird ein XML-Element erzeugt, das den XMI-Namen des Attributes trägt und den Wert des Attributes enthält. Wenn ein Attribut einen mehrwertigen Typ besitzt, wird für jeden Wert ein XML-Element erzeugt. Referenzen können ebenfalls durch XML-Elemente ausgedrückt werden. Dazu wird ein XML-Element mit dem XMI-Namen der Referenz erzeugt, das ein optionales XML-Attribut für die Typangabe und ein weiteres XML-Attribut für die Verbindung zum referenzierten Objekt enthält. Die Typangabe entspricht in ihrem Aufbau der Typangabe bei Objekten. Objekte, die im selben Dokument definiert sind, können durch ihr Identifikationsattribut referenziert werden. Für Objekte, die in einem anderen Dokument definiert sind, muss einer der in der XLinks-Spezifikation [XLi01] angegebenen Referenzmechanismen verwendet werden.

```
<mcl:Component xmi.id='1' name='Waschmaschine' whole='nil' >
  <name>Waschmaschine</name>
  <whole nil='true'/>
  <part xmi.id='2' type='mcl:Component' name='Steuerung'>
    Inhalt
  </part>
  <part xmi.id='3' type='mcl:Component' name='Tuer'>
    Inhalt
  </part>
   // weitere Bestandteile
</mcl:Component>
```

Abbildung 7.14: Repräsentation von Eigenschaften durch XML-Elemente

In Abb. 7.14 wird eine vollständige Darstellung der Komponente *Waschmaschine* angegeben. Es sind alle Angaben enthalten, die bereits in Abb. 7.13 dargestellt sind. Zusätzlich werden für die Eigenschaften der Metaklasse *Component* XML-Elemente angegeben. Die XML-Elemente **name** und **whole** enthalten die identischen Informationen wie die gleichnamigen XML-Attribute. Dies ist die Grundeinstellung, die bei der Erzeugung von XMI-Dokumenten angewendet wird. Sowohl die Erzeugung der Attribute als auch die Erzeugung der Elemente kann durch eine Konfigurationsvariable abgeschaltet werden. Die Komponente *Waschmaschine* enthält weitere Komponenten, z. B. die *Steuerung* und die *Tuer*. Für jede enthaltene Komponente wird ein XML-Element **part** erzeugt. Jedes **part**-Element erhält eine eindeutige Kennung und eine Typangabe. Die Kennung wird benötigt, um die Elemente referenzieren zu können. Die Typangabe ist erforderlich, da aus der XML-Schema-Typangabe nicht ersichtlich ist, welchen Typ ein **part**-Element repräsentiert, weil bei Verwendung der Voreinstellung alle Typreferenzen auf **xsd:Any** abgebildet werden. In diesem Fall wäre die Typangabe allerdings auch bei Verwendung des Vererbungsmechanismus' aus XML-Schema erforderlich, weil das MCL-Metamodell mehrere gültige Typen für die Eigenschaft *part* definiert.

Analog zur XMI-Schemagenerierung kann auch bei der XMI-Dokumentgenerierung durch die Konfigurationsvariablen **attribute** und **element** verhindert werden, dass ein XML-Element und

ein XML-Attribut für ein Attribut oder eine Referenz erzeugt werden. Außerdem kann die Repräsentation von Eigenschaften eines Objektes auch vollständig unterdrückt werden, indem die Konfigurationsvariable **serialize** auf den Wert „false “ gesetzt wird. Dies kann bei abgeleiteten Eigenschaften eingesetzt werden, um die Dokumente so klein wie möglich zu halten.

7.3 Zusammenfassung

Der OMG-Standard *XML Metadata Interchange* (XMI) definiert Abbildungen zwischen dem OMG-Standard MOF (siehe Kapitel 5) und den W3C-Standards XML und XML-Schema (siehe Kapitel 6), um verlustfreien Datenaustausch zwischen Modellierungswerkzeugen zu ermöglichen. Ein MOF-Modell wird auf ein XML-Schema abgebildet, das den Aufbau von XML-Dokumenten beschreibt. Die XML-Dokumente repräsentieren Anwendermodelle, die durch die Modellierungselemente beschrieben werden, die durch das MOF-Modell spezifiziert sind.

XMI-Schemata definieren die Struktur der XML-Elemente und -Attribute, die zur Beschreibung eines MOF-Modells benötigt werden. In der Voreinstellung erlauben die XML-Schemata nur wenige Validierungsmöglichkeiten und enthalten viel redundante Informationen. Beispielsweise werden Vererbungsbeziehungen durch Kopieren der Repräsentationen der geerbten Eigenschaften realisiert, weil die in MOF erlaubte Mehrfachvererbung nicht abgebildet werden kann. Außerdem werden für viele Eigenschaften einer MOF-Klasse sowohl XML-Elemente als auch XML-Attribute zur Repräsentation erzeugt.

Durch Konfigurationsvariablen, die einzelnen MOF-Modellelementen „angeheftet“ werden können, kann die Generierung der XMI-Schemata konfiguriert werden. Dies ermöglicht die Erzeugung kompakterer und restriktiverer XML-Schemata. Durch die Konfigurierbarkeit entstehen sowohl Vor- als auch Nachteile. Der Vorteil ist, dass für jeden Anwendungsfall ein optimales XMI-Austauschformat definiert werden kann, so dass z. B. die Validierungsmöglichkeiten von XML-Schema auch beim Datenaustausch durch XMI-Dokumente angewendet werden. Dies erleichtert die Implementierung von XMI-Schnittstellen. Der Nachteil ist, dass es nicht *das* XMI-Schema gibt, sondern für ein MOF-Modell viele verschiedene XMI-konforme XML-Schemata erzeugt werden können. Gleiches gilt für die XMI-Dokumente, die Anwendermodelle repräsentieren.

Kapitel 8

Generierte Komponenten

Jeder Codegenerator, der mit Hilfe des MOmo-Baukastens erstellt wird, enthält eine Implementierung eines Metamodells. Das Metamodell definiert die Elemente der Modellierungssprache, die der Codegenerator verarbeiten soll. Die Implementierung komplexer Metamodelle ist sehr aufwändig, da jedes Element des Metamodells in ein oder mehrere Elemente der Implementierung umgesetzt werden muss. Ein Beispiel für ein komplexes Metamodell ist das Metamodell der Version 2.0 der UML, das eine erheblich größere Anzahl von Modellelementen enthält als seine Vorgängerversionen. In der hier beschriebenen Repräsentation werden zur Implementierung des Metamodells der UML 2.0 14000 Java-Klassen benötigt.

Werkzeuge, die eine UML-basierte Modellierungssprache unterstützen, müssen das Metamodell der entsprechenden Sprache implementieren. Die Möglichkeit, Implementierungen von Metamodellen aus der Modellbeschreibung generieren zu könnnen, führt daher zu einer erheblichen Reduzierung des Aufwandes für die Realisierung eines Codegenerators, die unabhängig von der Implementierung der weiteren Bestandteile erreicht werden kann. Obwohl die MOmo-Implementierung eines MOF-Metamodells hauptsächlich auf die Anforderungen eines MOmo-Codegenerators zugeschnitten ist, kann sie auch die Entwicklung anderer Werkzeuge erleichtern.

Für die Version 1.4 des MOF-Standards wurde durch einen sog. *Java Community Process* eine Abbildung der Bestandteile eines MOF-Modells auf die Programmiersprache Java™ [GJSB00] festgelegt. Das Resultat, das *Java Metadata Interface* (JMI), wird in [Dir02] beschrieben. Eine Anpassung von JMI an die Version 2.0 des MOF-Standards ist bislang leider nicht vorgenommen worden, so dass für die MOmo-Implementierung entsprechende Veränderungen und Erweiterungen von JMI erst definiert werden mussten. Außerdem definiert JMI ausschließlich die Syntax der zu implementierenden Schnittstellen und sagt wenig über deren Semantik aus. Das Verhalten einer Implementierung der JMI-Schnittstelle ist daher nicht in allen Punkten genau festgelegt, so dass für die MOmo-Implementierung eine geeignete Umsetzung gewählt werden musste. Dies gilt insbesondere für die Umsetzung der Modellierungselemente, die in MOF 2.0 neu eingeführt werden.

Im Folgenden wird zunächst ein Überblick über den Aufbau einer MOmo-Implementierung eines MOF 2.0-Metamodells gegeben (Abschnitt 8.1). Darauf folgen Erläuterungen der verschiedenen Bestandteile einer solchen Implementierung (Abschnitte 8.2 und 8.3). Insbesondere werden die Schnittstellen angegeben, die eine MOmo-Implementierung bereitstellt, um die Elemente eines MOF-Modells zu verarbeiten. Außerdem wird beschrieben, wie die Schnittstellen implementiert werden.

8.1 Aufbau

Die MOmo-Implementierung eines MOF-Modells implementiert im Wesentlichen die Schnittstellen und Klassen, die durch die JMI-Spezifikation gefordert werden. Abweichungen werden immer dann erforderlich, wenn Modellelemente abgebildet werden müssen, die in MOF 2.0 neu eingeführt werden oder deren Definitionen sich in MOF 2.0 und MOF 1.4 unterscheiden. In die erstgenannte Kategorie fallen z. B. referenzierte Assoziationsenden und Pakete; in die letztgenannte die Möglichkeiten zur Definition von Redefinitions- und Teilmengenbeziehungen zwischen Attributen einer Klasse.

Die JMI-Spezifikation beschreibt eine Reihe von Schnittstellen, die eine konforme Implementierung für ein MOF-Modell bereitstellen muss. Ein Teil dieser Schnittstellen bildet die Elemente eines MOF-Modells auf Java-Schnittstellen ab. In der Regel korrespondiert jedes Element eines MOF-Modells mit einer festgelegten Anzahl von Bestandteilen der JMI-Schnittstelle. Beispielsweise wird jedes Paket auf genau eine Java-Schnittstelle abgebildet, und für jedes Attribut einer Klasse werden Methoden zum Abfragen und Setzen erzeugt. An den Beispielen wird bereits deutlich, dass die Schachtelungsebenen eines MOF-Modells in einer JMI-Implementierung weitgehend erhalten bleiben. Die Hauptelemente eines MOF-Modells, d. h. Pakete, Klassen, Assoziationen und Datentypen, werden auf Java-Schnittstellen abgebildet. Die anderen Elemente werden dagegen durch Methoden der Schnittstellen der übergeordneten Elemente repräsentiert.

Neben den modellabhängigen Elementen enthält eine JMI-konforme Implementierung auch Komponenten, die allen Implementierungen gemeinsam sind. Diese Komponenten umfassen Schnittstellen und Klassen, die zur Implementierung grundlegender Mechanismen benötigt werden. Diese Mechanismen, z. B. Reflexion und XMI-Import/-Export, werden in jeder Implementierung benötigt. Neben den genannten modellunabhängigen Komponenten, die Eigenschaften des gesamten Modells realisieren, werden weitere Komponenten zur Implementierung bestimmter Eigenschaften der modellabhängigen Bestandteile erforderlich. Diese umfassen unter anderem Klassen zur konsistenzerhaltenden Implementierung von Attributen und Assoziationsenden.

Zwischen den beiden Arten modellunabhängiger Bestandteile einer JMI-Implementierung besteht ein grundsätzlicher Unterschied. Die Komponenten zur Implementierung von Reflexion und XMI-Schnittstelle werden von der JMI-Spezifikation zumindest indirekt vorgeschrieben. Reflexion ermöglicht die Abfrage des Aufbaus eines Modellelementes zur Laufzeit, so dass das Element ohne vorherige Kenntnis seiner Struktur verwendet werden kann. Die

XMI-Schnittstelle verwendet die reflexive Schnittstelle, um das Laden und Speichern von Modellen zu realisieren. Die weiteren Komponenten sind dagegen nur dann notwendig, wenn die JMI-Implementierung möglichst komfortabel für den Programmierer gestaltet werden soll. Alternativ könnte etwa die Sicherung der Konsistenz einfach an den Programmierer übertragen werden, so dass keine weiteren Bestandteile nötig wären.

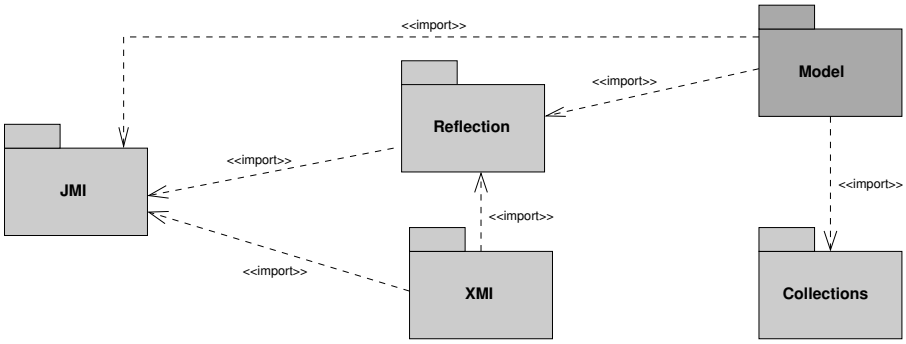


Abbildung 8.1: Bestandteile einer MOmo-Implementierung

In Abb. 8.1 ist die Architektur einer MOmo-Implementierung eines MOF-Metamodells dargestellt. Eine solche Implementierung besteht aus mindestens fünf Bestandteilen, die aufeinander aufbauen. Abb. 8.1 zeigt ein Paket für jeden Bestandteil. Das Paket **Model** enthält die modellabhängigen Bestandteile einer JMI-Implementierung, d. h. die Schnittstellen und Klassen, die die Elemente des MOF-Modells auf Java abbilden. Alle anderen Pakete sind in jeder MOmo-Implementierung eines MOF-Modells in gleicher Weise enthalten.

Das Paket **JMI** ist die Grundlage einer JMI-konformen Implementierung. Es enthält eine vollständige Implementierung des MOF-Metamodells, weil der Reflexionsmechanismus das Vorhandensein der MOF-Klassen erfordert. Außerdem enthält das **JMI**-Paket die Definitionen aller JMI-spezifischen Ausnahmen, die innerhalb einer JMI-Implementierung ausgelöst werden dürfen.

Die Pakete **Reflection** und **XMI** enthalten die Implementierungen der reflexiven Schnittstelle und der XMI-Schnittstelle, die von JMI gefordert werden. Das Paket **Collections** enthält JMI-spezifische Klassen zur Abbildung von Attributen und Assoziationsenden. In den folgenden Abschnitten werden die einzelnen Bestandteile einer JMI-konformen MOmo-Implementierung näher beschrieben.

8.2 Modellabhängige Bestandteile

Die modellabhängigen Bestandteile einer JMI-Implementierung bilden die Elemente eines MOF-Modells auf Elemente der Programmiersprache Java ab. Die JMI-Spezifikation legt fest, welche Schnittstellen für die vier Grundbausteine eines MOF-Modells, Pakete, Klassen, Assoziationen und Datentypen, erzeugt werden müssen. Neben den Schnittstellen werden in einer MOmo-Implementierung korrespondierende Klassen erzeugt, die die Schnittstellen implemen-

tieren. Diese Klassen sind *nicht* Teil der JMI-Spezifikation und mussten daher im Rahmen der Entwicklung des MOmo-Baukastens neu entwickelt werden.

Die existierende JMI-Implementierung Netbeans *Meta Data Repository (MDR)* [Mat03] verwendet eine Implementierung der MOF-Klassen und Reflexion zur Implementierung der JMI-Schnittstellen. Dies verkompliziert allerdings die Anwendung eines MDR-basierten Werkzeuges, weil das MOF-Repository als Voraussetzung benötigt wird (siehe Kapitel 3, Abschnitt 3.3.1). Die Verwendung der Reflexion bei jedem Zugriff einer Anwendung auf das Repository führt zudem zu einem schlechteren Laufzeitverhalten. In der derzeit verfügbaren Version kann MDR nur MOF 1.4-Modelle verabeiten. Um auch die Mechanismen der MOF-Version 2.0 unterstützen zu können, mussten im Rahmen dieser Arbeit Erweiterungen und Anpassungen des JMI-Standards entwickelt werden.

8.2.1 Pakete

Sowohl MOF als auch Java enthalten einen Paketmechanismus zur Stukturierung von Modellen bzw. Programmen. MOF-Pakete werden daher in JMI zunächst auf Java-Pakete abgebildet. Außerdem wird für jedes MOF-Paket eine Java-Schnittstelle erzeugt. Die Motivation ist, dass JMI genaugenommen keine Schnittstelle zu Paketen, sondern zu Paketinstanzen anbietet. Eine Paketinstanz definiert eine logische Domäne für Instanzen der M1-Ebene (siehe Kapitel 5, Seite 66). Diese logischen Domänen werden in den Versionen 1.x der MOF-Spezifikation als *Extent* bezeichnet und bilden die Grundlage einiger Regeln, die erlaubte und nicht erlaubte Beziehungen zwischen Instanzen auf der M1-Ebene beschreiben. In der Version 2.0 sollen diese Aspekte in einer gesonderten Spezifikation (MOF 2 Facility and Object Lifecycle) behandelt werden, die zur Zeit noch nicht vorliegt.

MOmo-Codegeneratoren arbeiten in genau einer logischen Domäne, so dass Regeln für erlaubte bzw. nicht erlaubte Beziehungen zwischen Objekten verschiedener Domänen hier keine Rolle spielen. Trotzdem erzeugt die hier beschriebene JMI-Implementierung Schnittstellen und Klassen, die Instanzen der Pakete eines MOF-Modells repräsentieren. Der Hauptgrund dafür ist, dass ein MOF-Modell eine geschachtelte Struktur aufweist, deren oberstes Element immer ein Paket ist. Eine Instanz dieses äußersten Elementes kann daher gut zur Erzeugung von Instanzen aller untergeordneten Modellelemente verwendet werden.

Dementsprechend ermöglichen die Schnittstellen zu den Paketinstanzen den Zugriff auf enthaltene Pakete, Klassen und Datentypen. Zugriff bedeutet hier, dass eine Paketinstanz für jedes enthaltene Paket, jede enthaltene Klasse und jeden enthaltenen Datentyp einen Verweis auf ein Objekt liefern kann, das das jeweilige Modellelement, also Paket, Klasse oder Datentyp, repräsentiert. Im Unterschied zu JMI 1.0 werden auch strukturierte Datentypen durch ein Objekt repräsentiert, um eine einheitliche Vorgehensweise bei der Abbildung der MOF-Modellelemente zu gewährleisten.

Ein weiterer Grund für die Abbildung von MOF-Paketen auf Klassen ist die größere Flexibilität bez. der Abbildung von Beziehungen zwischen Paketen. Die möglichen Beziehungen zwischen Paketen wurden in MOF 2.0 gegenüber MOF 1.x stark verändert. Während enge Kopplungen von Paketen in MOF 1.x durch Vererbungsbeziehungen und sog. *Clustering* ausgedrückt

wurden, enthält MOF 2.0 zu diesem Zweck einen Mechanismus zur *Vereinigung* von Paketen (siehe Kapitel 5, Seite 86). In MOF 1.x werden Beziehungen zwischen Paketen direkt über entsprechende Verbindungen zwischen den Repräsentationen ausgedrückt. In der JMI-Schnittstelle spiegelt sich dies z. B. durch Vererbungsbeziehungen zwischen den Schnittstellen wider, die für die Pakete bereitgestellt werden müssen. Durch Java-Pakete lassen sich diese Beziehungen nicht darstellen.

MOF 2.0 ersetzt Vereinigungsbeziehungen zwischen Paketen durch Vererbungs- und Redefinitionsbeziehungen zwischen den Modellelementen, die in den Paketen enthalten sind. Gemäß der MOF-Spezifikation können Vereinigungsbeziehungen zwischen Paketen in einem XMI-Dokument auf zweierlei Weise repräsentiert werden. Entweder werden die Vereinigungen explizit repräsentiert, oder es werden alle durch eine Vereinigungsbeziehung ausgedrückten Beziehungen zwischen den Objekten, die in den beteiligten Paketen enthalten sind, dargestellt. Die letztgenannte Darstellungsform ist in jedem Fall die Voraussetzung, um Code für ein MOF-Modell erzeugen zu können. Aus diesem Grund werden alle in einem Eingabedokument explizit repräsentierten Vereinigungsbeziehungen vor der eigentlichen Codegenerierung aufgelöst. Der MOmo-Baukasten enthält zu diesem Zweck ein Modul, das die „Ausfaltung“ von Vereinigungsbeziehungen in einem MOF-Modell durchführt. Zur Laufzeit bestehen keine Beziehungen zwischen den verschiedenen Paketinstanzen eines Modells.

Aufgrund des Wegfalls von Vererbungs- und Clustering-Beziehungen werden einige Elemente der JMI-Abbildung eines Paketes zur Abbildung von MOF 2.0 Modellen nicht länger benötigt. Die Schnittstelle für ein Paket verändert sich entsprechend. Es werden nur noch Methoden zum Zugriff auf die enthaltenen Pakete, Klassen und Datentypen benötigt. In Abb. 8.2 ist die Schablone abgebildet, die in der MOmo-Implementierung zur Erzeugung von Schnittstellen für Pakete verwendet wird.

```
public interface <packageName>Package extends RefPackage {

  // für jedes enthaltene Paket
  public <NestedPackageName>Package get<NestedPackageName>();

  // für jede enthaltene Klasse
  public <ClassName>Class get<ClassName>();

  // für jede enthaltene Assoziation
  public <AssociationName> get<AssociationName>();

  // für jeden enthaltenen strukturierten Datentyp
  public <StructTypeName>DataType get<StructTypeName>()

  // für jeden enthaltenen Auszählungstyp
  public <EnumerationTypeName> get<EnumerationTypeName>()
}
```

Abbildung 8.2: Erzeugung von Schnittstellen für Pakete

Die Hauptaufgabe der Schablone zur Erzeugung von Schnittstellen für Paketinstanzen ist demnach, eine Schnittstelle für das *Factory*-Muster [GHJV94] zu implementieren. Für jedes enthaltene Paket wird eine Methode **get**<*NestedPackageName*> erzeugt. Die Methode liefert einen Verweis auf ein Objekt zurück, das eine Instanz des enthaltenen Paketes repräsentiert. Die Schnittstelle des zurückgelieferten Objektes wird ebenfalls durch Anwendung der Schablone aus Abb. 8.2 erzeugt.

Für jede enthaltene Klasse wird eine Methode **get**<*ClassName*> erzeugt, die einen Verweis auf ein Objekt zurückliefert, das die enthaltene Klasse repräsentiert. Der Aufbau der Schnittstelle eines solchen Objektes wird in Abschnitt 8.2.2 beschrieben. Außerdem wird für jeden enthaltenen strukturierten Datentyp eine Methode **get**<*StructTypeName*> und für jeden enthaltenen Aufzählungtyp eine Methode **get***EnumerationTypeName* erzeugt. Diese liefern eine Referenz auf ein Objekt, das den jeweiligen Datentyp darstellt. Der Aufbau der Schnittstellen für strukturierte Datentypen und Auszählungtypen wird in den Abschnitten 8.2.4.2 bzw. 8.2.4.3 beschrieben.

Neben der angegebenen Schnittstelle wird in einer MOmo-Implementierung eines MOF-Modells zudem eine Klasse erzeugt, die die Schnittstelle implementiert. Es bestehen verschiedene Möglichkeiten, die entsprechenden Methoden zu realisieren. Entweder wird das *Singleton*-Muster verwendet, um sicherzustellen, dass innerhalb jeder Paketinstanz nur genau eine Instanz der enthaltenen Elemente existiert, oder jeder Aufruf der Methoden erzeugt eine neue Instanz des enthaltenen Elementes. Die JMI-Spezifikation macht keinerlei Angaben über das Verhalten einer konformen Implementierung. Um jegliche Einschränkung der Flexibilität zu vermeiden, realisiert die MOmo-Implementierung das letztgenannte Verhalten, so dass der Programmierer dafür verantwortlich ist, keine Elemente unerwünscht mehrfach zu erzeugen.

8.2.2 Klassen

MOF-Modelle beschreiben im Wesentlichen strukturelle Eigenschaften. Klassen und Assoziationen sind die Grundbausteine, die verwendet werden können, um die strukturellen Eigenschaften auszudrücken. Entsprechend wichtig ist die Repräsentation der Eigenschaften von Klassen und Assoziationen in Abbildungen der MOF auf Programmiersprachen.

Der JMI-Standard definiert zwei Schnittstellen, die eine konforme Implementierung für jede Klasse bereitstellen muss. Eine Schnittstelle wird mit dem Klassennamen und einem nachgestellten **Class** bezeichnet und bildet die Eigenschaften der Klasse ab. Die wichtigste Funktion dieser Schnittstelle ist die Erzeugung von Instanzen, die durch zwei **create**-Methoden ermöglicht wird. Weitere **create**-Methoden werden für alle strukturierten Datentypen bereitgestellt, die innerhalb der Klasse definiert sind.

JMI sieht zudem vor, alle auf Ebene der Klassen definierten Attribute und Operationen in dieser Klasse abzubilden. Diese Attribute und Operationen entsprechen in ihrer Bedeutung weitgehend den **static**-Attributen und -Operationen aus C++. In MOF 2.0 ist die Spezifikation von Attributen und Operationen auf Klassenebene nicht mehr möglich, so dass die MOmo-Implementierung auch keine Abbildungen enthält. Die resultierende Schablone zur Erzeugung von Schnittstellen zur Repräsentation von Klassen ist in Abb. 8.3 dargestellt.

```
public interface <ClassName>Class {

  public <ClassName> create<ClassName>()
    throws javax.jmi.reflect.JmiException;

  public <ClassName> create<ClassName>(
    // fuer jedes nicht-abgeleitete Attribut
    <AttributeType> <AttributeName>
  ) throws javax.jmi.reflect.JmiException;

  // fuer jeden enthaltenen strukturierten Datentyp
  public <StructTypeName>DataType get<StructTypeName>()
    throws javax.jmi.reflect.JmiException;
}
```

Abbildung 8.3: Erzeugung von Schnittstellen für Klassen

Aufgrund der in MOF 2.0 fehlenden Attribute und Operationen auf Klassenebene entfällt ein wichtiger Grund für die Erzeugung von Schnittstellen zur Repräsentation von Klassen. Die zwingend benötigte Funktionalität reduziert sich auf das Erzeugen von Instanzen, d. h. die Implementierung des *Factory*-Musters. Gemäß MOF-Spezifikation befindet sich ohnehin jede Klasse in einem übergeordneten Paket oder einer übergeordneten Klasse, so dass die Methoden zur Erzeugung von Instanzen ebenso gut in den Repräsentationen der jeweils übergeordneten Modellelemente angeordnet werden könnten. Daraus resultiert eine erheblich schlankere Schnittstelle zu einem MOF-Modell, die zudem leichter zu verwenden ist, da Instanzen ohne zusätzlichen Zwischenschritt erzeugt werden können. Eine entsprechende Abbildung wird von Fraunhofer u. a. [FIT04] vorgeschlagen, die eine Abbildung von MOF 2.0 auf CORBA-IDL definieren.

In MOmo werden trotz der genannten Vorteile Schnittstellen und Implementierungen zur Repräsentation von Klassen erzeugt. Die Hauptgründe dafür sind Kompatibilität zu JMI 1.0 und die größere Flexibilität, die durch die Erzeugung der Klassenrepräsentationen erreicht wird. Die Flexibilität ist von Vorteil, wenn weitere Operationen auf Klassenebene, z. B. das Abfragen aller Instanzen einer Klasse, in die Implementierung integriert werden müssen. Entsprechende Operationen werden in metamodellbasierten Anwendungen häufig benötigt.

Die zweite Schnittstelle, die durch die JMI-Spezifikation festgelegt wird, beschreibt die Instanzen einer Klasse. Alle Attribute und Operationen einer Klasse müssen in dieser Schnittstelle auf Java™-Methoden abgebildet werden. Die JMI-Spezifikation definiert, wie die Methoden zur Abbildung von Attributen aus Name, Typ und Multiplizität des Attributes hergeleitet werden. Abb. 8.4 zeigt eine Schablone zur Erzeugung einer Schnittstelle für die Instanzen einer MOF-Klasse.

Wenn eine Klasse keine Superklassen besitzt, wird die Schnittstelle **RefObject** aus dem Paket **JMI** geerbt, die die reflexive Schnittstelle bereitstellt. Andernfalls werden die Schnittstellen aller Superklassen geerbt, die durch Anwendung der Schablone aus Abb. 8.4 für die Superklassen erzeugt werden. Auf Ebene der Schnittstellen unterstützt Java Mehrfachvererbung, so dass die

```
public interface <ClassName> extends
  // falls die Klasse keine Superklassen besitzt
  javax.jmi.reflect.RefObject
  // andernfalls, fuer jede Superklasse
  <SuperClassName>, ...
{
// fuer jedes Attribut
  // falls obere Grenze der Multiplizitaet gleich 1
  public <AttributeType> <AccessorName> ()
    throws javax.jmi.reflect.JmiException;
    // falls nicht schreibgeschuetzt
    public void <MutuatorName> (<AttributeType> newValue)
      throws javax.jmi.reflect.JmiException;

  // falls obere Grenze der Multiplizitaet groesser 1 und ungeordnet
  public Collection get<AttributeName> ()
    throws javax.jmi.reflect.JmiException;

  // falls obere Grenze der Multiplizitaet groesser 1 und geordnet
  public List get<AttributeName> ()
    throws javax.jmi.reflext.JmiException;

// fuer jede Operation
  // falls die Operation keinen Rueckgabewert besitzt
  public void <OperationName> (
  // sonst public <ReturnType>
  <OperationName> (
    // fuer jeden Parameter
    <ParameterType> <ParameterName>
  ) throws
    // fuer jede Ausnahme, die von der Operation ausgeloest wird
    <ExceptionName>, .., javax.jmi.reflect.JmiException;
}
```

Abbildung 8.4: Erzeugung von Schnittstellen für Objekte

in MOF möglichen Mehrfachvererbungen zwischen Klassen hier problemlos abgebildet werden können.

Jedes Attribut einer Klasse wird auf eine Lesemethode abgebildet, deren Rückgabewert von Typ und Multiplizität abhängt. Für beschreibbare, optionale oder einwertige Attribute werden zusätzlich Methoden zum Schreiben des Attributes erzeugt. Der Name der Leseoperation ist in der Regel **get** gefolgt vom Attributnamen. Eine Ausnahme bilden bool'sche Attribute, deren Leseoperation durch **is** gefolgt vom Attributnamen benannt ist, wenn der Attributname nicht bereits mit **is** beginnt. Andernfalls wird direkt der Name des Attributes verwendet. Bei den zusammengesetzten Namen wird der erste Buchstabe des Attributnamens immer groß geschrieben. Analog wird die Schreiboperation eines optionalen oder einwertigen Attributes aus **set** und dem Attributnamen zusammengesetzt.

Für einwertige und optionale Attribute haben der Rückgabewert der Leseoperation und der Parameter der Schreiboperation den Typ des Attributes. Für mehrwertige Attribute ist der Typ des Rückgabewertes der Leseoperation eine Klasse aus dem *Java Development Kit (JDK)*. Für ungeordnete Attribute wird **java.util.Collection** und für geordnete Attribute **java.util.List** zurückgegeben. Alle Einfüge- und Löschvorgänge müssen durch Aufrufe der Methoden der JDK-Klassen durchgeführt werden. So können eine vergleichsweise schlanke Schnittstelle und ein großer Funktionsumfang gleichzeitig realisiert werden.

MOF-Operationen werden direkt auf Java-Methoden mit dem gleichen Namen abgebildet. Wenn eine MOF-Operation keinen Rückgabewert besitzt, erhält die Java-Methode den Rückgabewerttyp **void**, andernfalls erhält sie die Abbildung des Typs der MOF-Operation. Alle Parameter der Operation werden auf Parameter der Java-Methode abgebildet. Die Abbildung der Typen von Operation und Parametern erfolgt in identischer Weise zu der Abbildung von Attributtypen.

Während die Erzeugung der Schnittstellen durch die JMI-Spezifikation exakt vorgeschrieben ist, werden über die weiteren Eigenschaften einer JMI-konformen Implementierung keine Aussagen gemacht, so dass die Auswahl geeigneter Mechanismen dem Implementierer überlassen bleibt. Im Folgenden wird die Realisierung der zwei wichtigsten Eigenschaften beschrieben. Zunächst wird auf die Umsetzung der Mehrfachvererbung eingegangen, und anschließend wird die Abbildung von Attributen und Operationen beschrieben.

8.2.2.1 Vererbung

Vererbung ist eines der Kernkonzepte objektorientierter Modellierungs- und Programmiersprachen. Die verfügbaren Vererbungsmechanismen lassen sich einerseits in *Schnittstellenvererbung* und *Implementierungsvererbung* und andererseits in *Einfachvererbung* und *Mehrfachvererbung* unterscheiden. Objektorientierte Programmier- oder Modellierungssprachen enthalten eine oder mehrere der genannten Möglichkeiten, um Vererbungsbeziehungen zwischen Klassen oder Objekten auszudrücken. Eine Übersicht über die Mechanismen gängiger Sprachen enthält [Pru97].

MOF erlaubt Mehrfachvererbung zwischen Klassen, d. h. eine Klasse darf von mehreren direkten Superklassen abgeleitet werden. Dies bedeutet, dass die Mehrfachvererbung sowohl auf Schnittstellen- als auch auf Implementierungsebene eingesetzt werden kann. Anders ausgedrückt sagt eine Mehrfachvererbung in MOF sowohl aus, dass die abgeleitete Klasse typkompatibel zu allen ihren Superklassen ist, als auch, dass die abgeleitete Klasse die Strukturen aller Superklassen enthält.

Java erlaubt Mehrfachvererbung auf Schnittstellenebene, d. h. eine Schnittstelle kann „ist-ein“-Beziehungen zu mehreren anderen Schnittstellen besitzen. Auf Implementierungsebene erlaubt Java dagegen nur einfache Vererbung, so dass die Struktur einer Klasse immer nur die Struktur genau einer direkten Superklasse enthalten kann.

Um ein MOF-Modell in Java implementieren zu können, muss die Mehrfachvererbung auf die in Java vorhandenen Mechanismen abgebildet werden. Die JMI-Spezifikation beschreibt lediglich den Aufbau der Schnittstellen, die in einer konformen Implementierung für jede Klasse

bereitgestellt werden müssen. Wie diese Schnittstellen implementiert werden, wird nicht spezifiziert. Grundsätzlich müssen also für jede Klasse die JMI-konforme Schnittstelle und eine Klasse, die diese Schnittstelle implementiert, erzeugt werden. Ein Beispiel für die Übertragung einer Klassenhierachie von MOF nach Java zeigt Abb. 8.5.

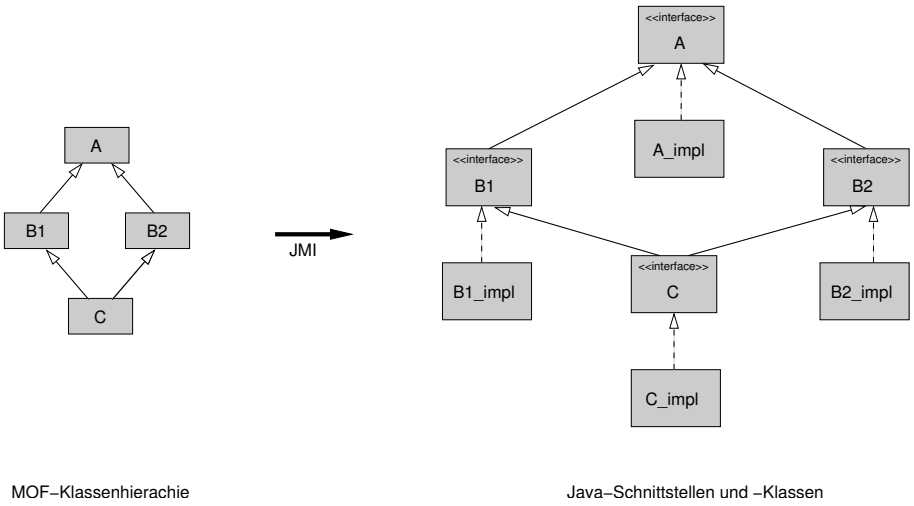


Abbildung 8.5: Abbildung einer MOF-Klassenhierarchie auf Java

Die Abbildung zeigt eine für den Einsatz von Mehrfachvererbungen typische Situation, die sog. Diamantenhierarchie. Die Klassen **B1** und **B2** erben von einer gemeinsamen Basisklasse **A**. Sowohl **B1** als auch **B2** werden wiederum von der Klasse **C** geerbt. In einer solchen Hierarchie stellt sich die Frage, wie häufig die Attribute und Operationen der Klasse **A** von der Klasse **C** geerbt werden. In den meisten Programmiersprachen, die mehrfache Vererbung unterstützen, bilden die geerbten Klassen eine Menge, so dass von jeder Klasse nur einmal geerbt werden kann. Dies wird in der Literatur auch als *virtuelle* Mehrfachvererbung bezeichnet. Eine Ausnahme von dieser Regel ist die Programmiersprache C++, die eine explizite Festlegung durch den Programmierer erfordert.

Zudem müssen Lösungen für die durch Mehrfachvererbung verursachten Schwierigkeiten, wie z. B. kollidierende Namen von Attributen und Operationen oder mehrfaches Erben von derselben Klasse festgelegt werden. Die Literatur unterscheidet im Wesentlichen drei Ansätze zur Abbildung von Mehrfachvererbungen auf Programmiersprachen, die nur einfache Vererbung unterstützen: Delegation, Kopieren und die Kombination der Einfachvererbung aus Java mit Delegation oder Kopieren.

Die technisch eleganteste Lösung ist die Delegation. Eine Klasse enthält eine Referenz für jede beerbte Klasse und leitet die Aufrufe aller beerbten Methoden an die entsprechenden Referenzen weiter. In Abb. 8.6 ist ein Beispiel für die Realisierung einer Mehrfachvererbungsbeziehung durch Delegation dargestellt.

Die Klasse **D** erbt von den Klassen **B** und **C**, die wiederum jeweils von der Klasse **A** erben. Die Umsetzung dieser Vererbungsbeziehungen erfolgt, indem jede einzelne Vererbungsbeziehung zunächst durch eine gerichtete Assoziation von der beerbenden zur vererbenden Klasse ersetzt

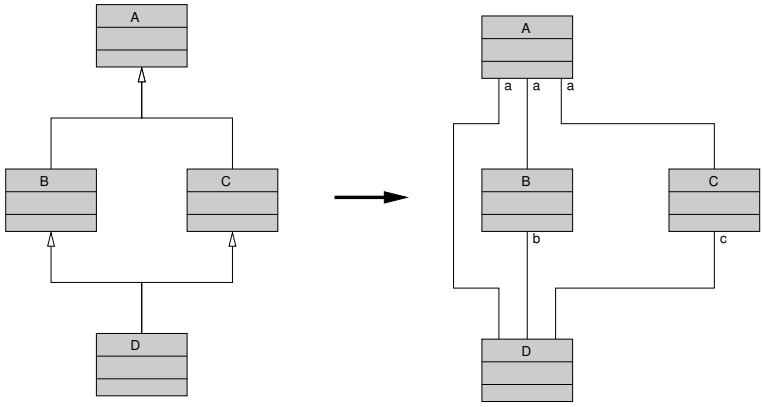


Abbildung 8.6: Implementierung einer Mehrfachvererbung durch Delegation

wird. Außerdem werden einer beerbenden Klasse alle Methoden der geerbten Klassen hinzugefügt. Dies muss transitiv geschehen, so dass auch eine Assoziation von der Klasse **D** zur Klasse **A** eingefügt werden muss. Zusätzlich müssen alle Methoden der vererbenden Klassen in die beerbende Klasse eingefügt werden. Die eingefügten Methoden reichen sämtliche Aufrufe an das Delegat der beerbten Klasse weiter.

Delegation ermöglicht die Abbildung von Mehrfachvererbungen auf Programmiersprachen und vermeidet weitgehend die Erzeugung zusätzlichen Codes zur Implementierung. Der Nachteil der Delegation ist, dass für jede beerbte Klasse ein Delegat erzeugt werden muss. Bei tiefen Vererbungshierarchien hat Delegation daher einen erheblichen Leistungsverlust zur Folge. Zudem werden mehrfach beerbte Klassen auch mehrfach erzeugt, was den Leistungsverlust weiter vergößert.

Die zweite Möglichkeit der Abbildung von Mehrfachvererbung auf Java-Klassen ist das Kopieren aller geerbten Attribute und Operationen von einer beerbten in die erbende Klasse. Im Allgemeinen ist das Kopieren von Code zu vermeiden, weil die Wartbarkeit deutlich verschlechtert wird. Wird der Code allerdings generiert, spielt dieser Nachteil kaum eine Rolle, weil das zu wartende Artefakt nicht der Code sondern die Code-Schablone ist. Ein Vorteil der Implementierung einer Mehrfachvererbung durch Kopieren des geerbten Codes ist, dass für jede Schnittstelle genau eine Implementierung erzeugt wird. Da keine zusätzlichen Objekte und Aufrufe benötigt werden, ergibt sich eine deutlich bessere Laufzeiteffizienz. Dieser Vorteil wiegt umso schwerer, je tiefer die Vererbungshierarchien sind. Bezüglich des Speicherverbrauchs ergibt sich ein ähnliches Bild. Einerseits führt der Delegationsansatz zur Erzeugung kompakteren Codes, andererseits aber zu höherem Speicherverbrauch der Objekte zur Laufzeit. Da der Speicherverbrauch des Codes gegenüber dem Speicherverbrauch zur Laufzeit weniger bedeutend ist, weist der Ansatz des Kopierens auch Vorteile in dieser Beziehung auf. In Abb. 8.7 ist ein Beispiel für die Abbildung einer Mehrfachvererbung durch Kopieren dargestellt.

Die Abbildung zeigt, dass alle Attribute und Operationen einer Superklasse in alle von ihr direkt oder indirekt abgeleiteten Klassen kopiert werden müssen. Im dargestellten Beispiel erben zunächst die Klassen **B** und **C** direkt von der Klasse **A**. Die Klasse **D** beerbt die Klassen **B** und **C** direkt und die Klasse **A** indirekt. Um diese Vererbungsbeziehungen durch Kopieren von At-

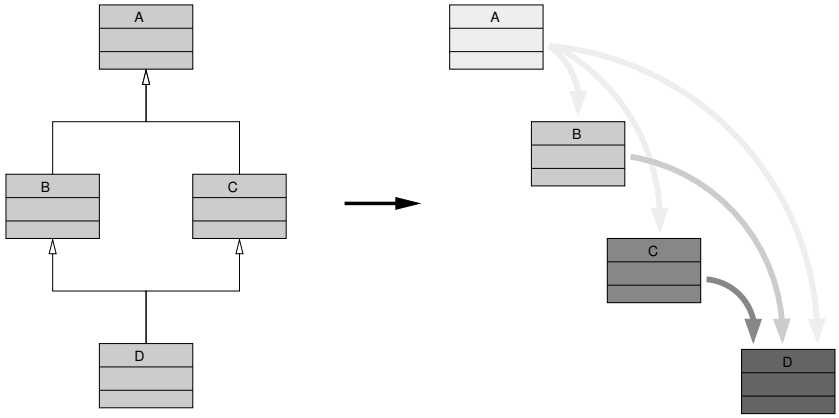


Abbildung 8.7: Implementierung einer Mehrfachvererbung durch Kopieren

tributen und Operationen zu implementieren, müssen alle Attribute und Operationen der Klasse **A** in die Klassen **B**, **C** und **D** kopiert werden. Außerdem müssen die Attribute und Operationen der Klassen **B** und **C** in die Implementierung der Klasse **D** kopiert werden. In Abb. 8.7 wird jeder Kopiervorgang durch einen Pfeil in der Farbe der jeweils kopierten Klasse dargestellt.

Beide Ansätze können mit dem in Java enthaltenen Mechanismus der Einfachvererbung auf Implementierungsebene kombiniert werden. In diesem Fall wird eine der direkten Superklassen einer Klasse mit Hilfe des Java-Mechanismus' geerbt und alle anderen durch Delegation oder Kopieren hinzugefügt. Die Bibliothek *nsuml* [Plo02], eine Implementierung des Metamodells der UML-Version 1.3, verwendet eine Kombination aus Java-Implementierungsvererbung und Kopieren der Eigenschaften der zusätzlich geerbten Eigenschaften. Das Hauptproblem dieses Ansatzes ist die Behandlung von Konflikten zwischen den Namen geerbter Attribute oder Operationen. Da ein Teil der Attribute oder Operationen durch den Java-Mechanismus geerbt wird, kann für diese kein Einfluss auf die Namensvergabe ausgeübt werden. Die daraus resultierenden Probleme werden in Abschnitt 8.2.2.2 aufgezeigt.

Für die Implementierung der MOmo-Objektrepräsentation werden alle Vererbungsbeziehungen auf Implementierungsebene durch Kopieren von Code realisiert. Die Nachteile bez. Wartbarkeit und Abbildbarkeit einiger Konstellationen können als vernachlässigbar gegenüber den Nachteilen der anderen Möglichkeiten bewertet werden.

In Tab. 8.1 werden die Vor- und Nachteile der verschiedenen Verfahren zur Implementierung von Mehrfachvererbung in Java zusammenfassend dargestellt. Es wurden sechs Kriterien untersucht, um einen der Ansätze auszuwählen. Die beiden Kombinationen zwischen Java-Implementierungsvererbung und Delegation bzw. Kopieren sind in der Tabelle gemeinsam dargestellt, weil die Vor- und Nachteile wesentlich durch die Verwendung des Java-Mechanismus' hervorgerufen werden.

Die Kombination zwischen Java-Implementierungsvererbung und Delegation oder Kopieren weist zwei wesentliche Nachteile gegenüber der reinen Anwendung einer der beiden Abbildungsmechanismen auf. Der Aufwand, der zur Implementierung einer Schablone für diesen Ansatz erforderlich ist, ist deutlich höher als bei den beiden anderen Ansätzen, weil permanent

	Delegation	**Kopieren**	**Kombination**
Aufwand	+	+	o
Codegröße	-	-	+
Durchgängigkeit	+	+	-
Flexibilität	+	+	-
Leistung	-	+	o
Wartbarkeit	+	-	o

Tabelle 8.1: Vor- und Nachteile von Abbildungsmöglichkeiten

zwischen den durch den Java-Mechanismen und den zusätzlich zu implementierenden Vererbungsbeziehungen unterschieden werden muss.

Ein technischer Nachteil ist die mangelnde Flexibilität, die zum Beispiel bei der Auflösung von Namenskonflikten hinderlich ist. Wird Java-Implementierungsvererbung verwendet, kann die Vergabe der Namen für Attribute und Operationen nicht beeinflusst werden. Im Zusammenhang mit der mangelnden Durchgängigkeit des Kombinationskonzeptes, die durch die gleichzeitige Verwendung mehrerer Vererbungsrepräsentationen verursacht wird, ergeben sich Abhängigkeiten des generierten Codes von der Reihenfolge, in der die Vererbungsbeziehungen für eine Klasse durchlaufen werden, oder von den Kriterien, die zur Auswahl der direkt geerbten Klasse verwendet werden. Ein weiterer technischer Nachteil ist, dass sich die Java-Implementierungsvererbung nicht mehr zur Verbindung des generierten Codes mit anderen Bestandteilen einer Applikation verwenden lässt. Aus diesen Gründen wurden die Kombinationsansätze verworfen.

Der Einsatz von Delegation ist ein vielversprechendes Konzept zur Abbildung von Mehrfachvererbungen auf die Programmiersprache Java. In der Literatur werden mehrere Lösungen vorgestellt, die diesen Ansatz verwenden. Es existieren auch Implementierungen, die Java um Mehrfachvererbung erweitern und diese dann automatisch auf Delegationen abbilden (z. B. Jamie [VTB98]). Bettini u. a. zeigen eine Schwäche dieser Ansätze, die zu einer unvollständigen Abbildung der Mehrfachvererbung führt [BLV03]. Das Problem, das sog. *Delegations-Idiom*, wird durch Aufrufe von Methoden innerhalb der Methodenrümpfe der Delegate ausgelöst. Es muss sichergestellt werden, dass solche Aufrufe wiederum an das delegierende Objekt gerichtet sind, um die Semantik der Vererbung zu bewahren. Die Lösung dieses Problems erfordert die Übergabe des **this**-Zeigers des delegierenden Objektes an das Delegat. Anschließend muss das Delegat alle Aufrufe an sich selbst an diese Referenz richten. Es zeigt sich, dass der Aufwand zur Sicherstellung der Vererbungssemantik vergleichsweise hoch ist, wenn Delegation zur Implementierung verwendet wird.

Ein zweiter Nachteil der Delegation ist die benötigte Laufzeit, die zur Erzeugung von Objekten und zur Abarbeitung eines Methodenaufrufs benötigt wird. Für jedes Objekt müssen alle Delegate erzeugt werden, so dass bei tiefen Vererbungshierarchien eine Vielzahl von Objekten erzeugt werden muss, um die Funktionalität eines einzigen Objektes bereitzustellen. Zudem hat jeder Aufruf einer geerbten Methode mindestens einen weiteren Aufruf der Methode im Delegat zur Folge, was die Laufzeiteffizienz deutlich verschlechtert.

Demgegenüber stehen Vorteile des Delegationsansatzes auf den Gebieten Flexibilität und Wartbarkeit. Delegation ist der einzige Ansatz, der ohne Probleme sowohl virtuelle als auch nicht-virtuelle Mehrfachvererbung abbilden kann. Dazu müssen lediglich die Zusammensetzungen der Klassen entsprechend angepasst werden. Für virtuelle Mehrfachvererbung wird für jedes Mitglied der Menge aller direkt und indirekt geerbten Klassen eine Delegation erzeugt. Im Fall nicht-virtueller Vererbung wird nur für jede direkt geerbte Superklasse eine Delegation erzeugt. Diesem Vorteil der Flexibilität steht wiederum ein Effizienznachteil gegenüber, da bei der Abbildung virtueller Mehrfachvererbung sämtliche in den Delegaten erzeugten Delegationen überflüssig sind.

Die Implementierung von Mehrfachvererbung durch Kopieren des Codes geerbter Klassen hat deutliche Vorteile im Bereich Laufzeiteffizienz gegenüber dem Delegationsansatz. Zudem ist Kopieren deutlich flexibler einsetzbar als die Kombinationen aus Java-Einfachvererbung und Delegation oder Kopieren, ohne die mit Delegation verbundenen Laufzeitnachteile in Kauf nehmen zu müssen.

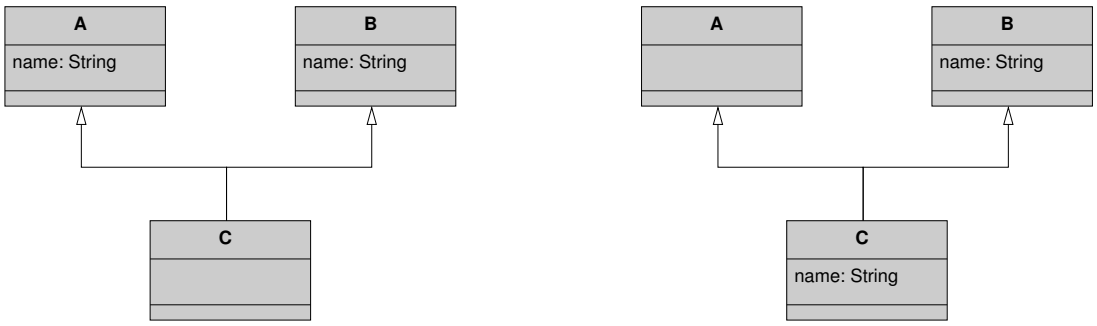
Da sich der Effizienznachteil bei der Abbildung von MOF-Modellen aufgrund der tiefen Vererbungshierarchien besonders stark auswirkt, wurde auch die Verwendung der Delegation zur Implementierung von MOF-Mehrfachvererbungsbeziehungen verworfen. Der Flexibilitätsvorteil gegenüber dem Kopieransatz hat kaum Auswirkungen, weil die Mehrfachvererbung in MOF-Modellen am ehesten mit der virtuellen Vererbung in C++ vergleichbar ist. Mehrfaches Vorkommen der Instanzen einer Basisklasse muss daher nicht abgebildet werden können.

8.2.2.2 Attribute und Operationen

Die Attribute einer MOF-Klasse werden durch einen Namen, einen Typ und eine Multiplizitätsangabe gekennzeichnet. Der Typ eines Attributes ist entweder eine Klasse oder ein Datentyp. Die Multiplizität eines Attributes hat maßgeblichen Einfluss auf die Abbildung des Attributes auf Java. Unabhängig von der Methode, die zur Abbildung einer Mehrfachvererbungsbeziehung auf Java gewählt wird, müssen einige Probleme gelöst werden. Ein häufig auftretendes Problem ist die Kollision zwischen den Namen von Eigenschaften, die aus verschiedenen Klassen geerbt werden. Abb. 8.8 zeigt die zwei prinzipiellen Möglichkeiten durch Vererbung auftretender Namenskonflikte.

In beiden dargestellten Klassenhierarchien erbt die Klasse **C** von den Klassen **A** und **B**. In Abb. 8.8(a) führt dies zu einem Konflikt zwischen den Attributen **name** aus **A** und **B**, weil durch die Vererbung beide Attribute in den Namensraum von **C** eingefügt werden. In manchen Programmiersprachen kann ein solcher Namenskonflikt durch Voranstellen des Namens der geerbten Klasse aufgelöst werden. In C++ wird der *„Scope Resolution Operator“* **::** verwendet, um die einzelnen Attribute eindeutig ansprechen zu können. Im dargestellten Beispiel wird das von der Klasse **A** geerbte Attribut **name** in einer Instanz der Klasse **C** durch **A::name** identifiziert.

In Java™existiert keine solche Möglichkeit, so dass Namenskonflikte nicht zur Übersetzungs-, sondern erst zur Laufzeit behandelt werden können. Dies wird weiter unten erläutert. Während



(a) horizontal (b) vertikal

Abbildung 8.8: Namenskonflikte zwischen Attributen

der Erzeugung des Implementierungscodes wird untersucht, ob ein Namenskonflikt vorliegt. Dazu müssen folgende Situationen unterschieden werden:

1. Der Name eines Attributes ist eindeutig innerhalb des Namensraumes, der durch die umgebende Klasse festgelegt wird. In diesem Fall kann das Attribut ohne weitere Untersuchungen erzeugt werden.

2. Der Name eines Attributes ist nicht eindeutig. In diesem Fall sind weitere Untersuchungen notwendig. Es muss festgestellt werden, ob ein Attribut die Wurzeleigenschaft besitzt. Wurzeleigenschaft bedeutet, dass alle anderen gleich benannten Attribute Redefinitionen des Attributes sind.

   (a) Wenn ein Attribut die Wurzeleigenschaft besitzt, kann es ebenfalls ohne Namenszusätze erzeugt werden.

   (b) Wenn ein Attribut weder eindeutig benannt ist noch die Wurzeleigenschaft besitzt, muss untersucht werden, ob es ein gleich benanntes Attribut redefiniert. In diesem Fall brauchen weder Attribut noch Zugriffsoperationen erzeugt werden. Andernfalls müssen Attribut und Zugriffsoperationen mit dem voll qualifizierten Namen des Attributes erzeugt werden.

In ähnlicher Weise müssen Konflikte zwischen gleich benannten Operationen aufgelöst werden. Entscheidender Unterschied zwischen Operationen und Attributen ist, dass Operationen nicht durch ihren Namen, sondern durch ihre Signatur identifiziert werden. Die Signatur einer Operation besteht aus ihrem Namen und den Typen ihrer Parameter. Im Unterschied zu MOF/UML gehört der Typ des Rückgabewertes einer Operation in den meisten Programmiersprachen nicht zur Signatur. Daraus resultieren sowohl Probleme bei der Abbildung von Operationen als auch von Attributen, weil für Attribute gemäß der JMI-Spezifikation Zugriffoperationen zum Abfragen und ggf. Verändern des Attributes erzeugt werden müssen.

Für Operationen kann das Problem nicht gelöst werden, so dass zwei Operationen, die sich nur durch ihren Rückgabewert unterscheiden, als Kollision behandelt werden müssen. Sie werden

analog zu den Attributen mit ihrem qualifizierten Namen erzeugt. Die Auflösung von Namenskonflikten zwischen Attributen und Operationen durch Erzeugung von Implementierungen mit voll qualifizierten Namen führt zu einem weiteren Problem. Auf Ebene der Schnittstellen wird der Mehrfachvererbungsmechanismus von Java verwendet, so dass die generierte Schnittstelle sowohl den unqualifizierten als auch die qualifizierten Namen des mehrfach definierten Attributes enthält. Wenn die Implementierung die Namenskonflikte lediglich auflöst, fehlt aber die Implementierung des unqualifizierten Attributes. Aus diesem Grund muss zusätzlich eine Implementierung der unqualifizierten Schnittstelle erfolgen. Da aufgrund des Konfliktes nicht sinnvoll festgelegt werden kann, auf welche der kollidierenden Attribute die Operationen für das unqualifizierte Attribut abgebildet werden sollen, führt der Aufruf einer solchen Operation zum Auslösen eines Laufzeitfehlers. Dies entspricht einer Übertragung des Verhaltens von Programmiersprachen mit Mehrfachvererbung mit dem Unterschied, dass die Identifikation des Problems erst zur Laufzeit erfolgt.

Ein weiteres Problem besteht in der Kollision zwischen definierten Operationen und durch die JMI-Schnittstelle definierten Operationen zur Abbildung von Attributen. Auf diese Art hervorgerufene Namenskonflikte werden in der MOmo-Implementierung nicht behandelt und müssen daher bereits bei der Modellierung berücksichtigt werden. Implizite Redefinitionsbeziehungen sollten nach Möglichkeit vermieden werden, weil sie ein Modell unübersichtlicher machen und die Verständlichkeit des Modells beeinträchtigen.

Von den oben genannten Eigenschaften eines Attributes fehlt nun noch die Abbildung der Multiplizitätsangabe. In der JMI-Spezifikation werden die Multiplizitäten in drei Kategorien unterteilt, die zur Erzeugung verschiedener Schnittstellen führen. Für ein Attribut, dessen Multiplizitätsangabe eine obere Grenze von Eins enthält, werden Methoden zum Abfragen und Setzen des Attributes erzeugt. Sowohl der Rückgabewert der Abfrage- als auch der Parameter der Setze-Methode besitzen den Typ des Attributes. Für Attribute, deren Multiplizitätsangabe eine obere Grenze größer als eins enthält, wird nur eine Abfragemethode generiert, deren Rückgabewert jeweils aus einer Referenz auf eine Klasse aus dem *Java Development Kit (JDK)* besteht. Ungeordnete Attribute werden durch eine Instanz der JDK-Klasse **Collection** repräsentiert, während geordnete Attribute durch eine Instanz der JDK-Klasse **List** dargestellt werden. In beiden Fällen erfolgen alle Operationen zum Einfügen, Löschen oder Verändern über die Schnittstellen der JDK-Klassen. Auffällig ist, dass die JMI-Spezifikation lediglich Aussagen über die Auswirkungen der Multiplizität zur Übersetzungszeit macht. Die Wirkung einer Multiplizitätsangabe zur Laufzeit ist unspezifiziert.

Multiplizitäten können aber durchaus Auswirkungen haben, die von einer Implementierung berücksichtigt werden sollten, und bei direkter Interpretation zu Problemen während der Laufzeit führen. Grundsätzlich gibt es zwei Fälle, die zur Laufzeit Probleme hervorrufen. Der erste Fall sind Multiplizitäten mit einer unteren Grenze, die größer als Null ist. In diesem Fall wird theoretisch die Belegung eines Attributes mit der vorgegebenen Anzahl von Instanzen erzwungen, so dass die Belegung genau genommen direkt bei der Erzeugung einer Instanz der umgebenden Klasse erfolgen muss. Außerdem können Löschoperationen zur Laufzeit aufgrund des Unterschreitens der unteren Grenze fehlschlagen. Der zweite Fall ist ein mehrwertiges Attribut mit einer endlichen oberen Grenze. In diesem Fall würden Einfügeoperationen zur Laufzeit bei

Überschreiten der oberen Grenze fehlschlagen. Eine Kombination beider Fälle führt somit unweigerlich zu einer Nur-Lesbarkeit eines Attributes. Wenn nämlich die untere und die obere Grenze übereinstimmen und größer als Null sind, darf gemäß der genannten Semantik nach der Initialisierung weder eingefügt noch gelöscht werden, weil in beiden Fällen eine der beiden Grenzen der Multiplizität über- bzw. unterschritten wird. Eine direkte Interpretation der Multiplizitäten ist aus den genannten Gründen nicht praktikabel.

Génova u.a schlagen vor, Multiplizitäten bei Einfüge- und Löschoperationen nicht zu berücksichtigen [GdCL03]. Stattdessen wird erst beim Abfragen eines Attributes überprüft, ob das Attribut in einem konsistenten Zustand ist. Da dieser Ansatz eine benutzbare Schnittstelle für jede beliebige Multiplizität garantiert, wurde er auch für die MOmo-Implementierung ausgewählt.

Die Implementierung der Attribute einer Klasse muss für „echte“ Attribute ebenso funktionieren wie für navigierbare Assoziationsenden. Navigierbare Assoziationsenden müssen ihre übergeordnete Assoziation über alle Operationen unterrichten, damit die Konsistenz der von einer Implementierung verwalteten Daten gewährleistet werden kann. Zudem entspricht die Semantik der JMI-Operationen zum Einfügen und Löschen von Elementen nicht exakt der Semantik, die durch die JDK-Klassen implementiert wird. Aus diesen Gründen können die Collection-Klassen des JDK lediglich als Basis für die Implementierung JMI-konformer Schnittstellen verwendet werden. In einer MOmo-Implementierung wird daher eine eigene Klassenhierarchie zur Implementierung von Attributen verwendet. In Abb. 8.9 sind die in dieser Hierarchie enthaltenen Klassen und ihre Beziehungen zu den JDK-Klassen **Collection** und **List** dargestellt.

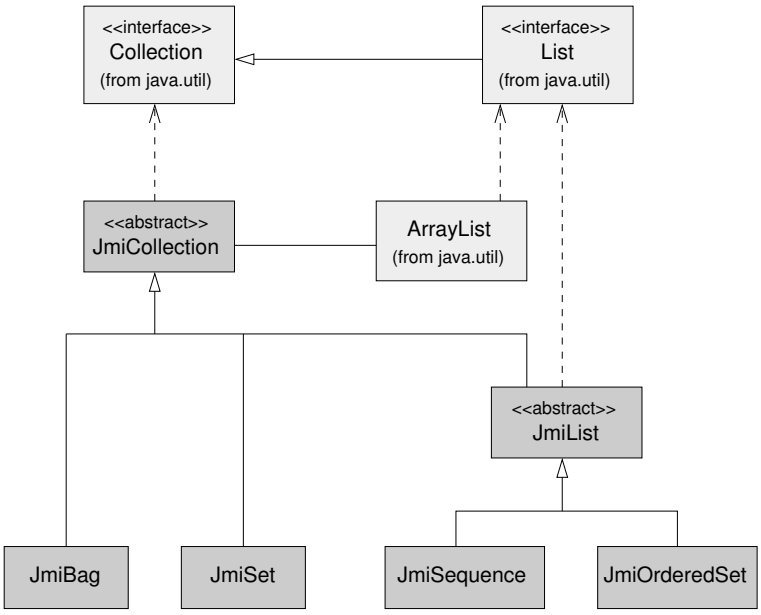


Abbildung 8.9: Klassen zur Implementierung von Attributen in MOmo

Die Klassenhierarchie besteht aus zwei abstrakten und vier konkreten Klassen. Die abstrakten Klassen **JmiCollection** und **JmiList** übernehmen die Implementierung der Schnittstellen **Col-**

lection bzw. **List** aus dem JDK. Die konkreten Klassen **JmiBag**, **JmiSet**, **JmiSequence** und **JmiOrderedSet** erben die Eigenschaften der abstrakten Basisklassen und bilden jeweils eine Kombination der Attribute *isOrdered* und *isUnique* ab. Die Benennung der konkreten Klassen entspricht dem Vorschlag aus der Spezifikation der UML-Infrastrukturbibliothek. In Tab. 8.2 wird dargestellt, welche JMI-Collection-Klasse für welche Kombination der Attribute verwendet wird.

isOrdered	isUnique	Klasse
false	true	JmiSet
true	true	JmiOrderedSet
false	false	JmiBag
true	false	JmiSequence

Tabelle 8.2: Abbildung der Attributarten (nach [UP03a])

Die Klasse **JmiSet** dient zur Abbildung eines Attributes, dessen Instanzen keine Werte mehrfach enthalten, und die Menge der enthaltenen Werte zudem nicht geordnet ist. Instanzen der Klasse **JmiOrderedSet** repräsentieren Attribute, deren Instanzen geordnete Wertemengen ohne mehrfach auftretende Werte enthalten. Entsprechend stellen die Klassen **JmiBag** und **JmiSequence** Attribute mit ungeordneten bzw. geordneten Mengen von Werten dar, erlauben aber im Unterschied zu **JmiSet** und **JmiOrderedSet** mehrfach auftretende Werte.

Die Implementierung eines Attributes in einer MOmo-Metadaten-Implementierung erfolgt unabhängig von der Multiplizität immer durch Instanziierung der JMI-Collection-Klasse, um die in diesen Klassen implementierten Mechanismen zur Abbildung von Redefinitions-, Teilmengen- und Vereinigungsbeziehungen nutzen zu können. Es wird jeweils die JMI-Collection-Klasse ausgewählt, die der Belegung der Meta-Attribute *isUnique* und *isOrdered* entspricht. Dies bedeutet, dass auch Attribute mit skalaren Typen durch Objekte repräsentiert werden müssen, weil JDK-Collection-Klassen nur Objekte aufnehmen können. Für einwertige skalare Attribute kann dies durch entsprechende Typumwandlungen in den Zugriffsmethoden verdeckt werden. Für mehrwertige Attribute ist dies nicht möglich, weil die Schnittstellen der JDK-Collection-Klassen zum Lesen und Schreiben von Attributen verwendet werden und diese keine Möglichkeiten zur Verarbeitung skalarer Typen enthalten. Der resultierende Verlust an Komfort bei der Verwendung der MOmo-Schnittstelle wurde in Kauf genommen, um eine möglichst einheitliche Implementierung aller Arten von Attributen zu ermöglichen. In der Version 5.0 des JDK wird der Komfortverlust durch das sog. *Autoboxing*, das die notwendigen Typumwandlungen automatisch vornimmt, ohnehin vermieden.

Die Implementierung der einzelnen Methoden aus den Schnittstellen **Collection** und **List** erfolgt durch Delegation an die in jeder Instanz einer JMI-Collection-Klasse enthaltene Instanz der JDK-Klasse **ArrayList**. Die Methoden zum Einfügen und Löschen von Objekten müssen zuvor einige Überprüfungen vornehmen. Die Einfügeoperation muss zunächst überprüfen, ob das Attribut schreibende Zugriffe erlaubt. Wenn ja, muss anschließend sichergestellt werden, dass der Typ des einzufügenden Objektes konform zu dem Typ des Attributes ist. In späteren Versionen der Schnittstelle, die auf der Java-Version 1.5 aufsetzen, sollten anstelle der Typ-

überprüfung zur Laufzeit die Möglichkeit einer Typüberprüfung zur Übersetzungszeit durch Verwendung der Java-Generics genutzt werden. In der vorliegenden Version wird auf diese Möglichkeit verzichtet, weil die Java-Version 5.0 bisher nur in Betaversionen für wenige Architekturen verfügbar ist.

Instanzen der Klassen **JmiSet** und **JmiOrderedSet** müssen nach einer erfolgreichen Typüberprüfung untersuchen, ob das einzufügende Objekt bereits in der Menge der enthaltenen Objekte enthalten ist. Ist dies der Fall, wird eine Ausnahme ausgelöst und das Einfügen damit verweigert. Instanzen der Klassen **JmiBag** und **JmiSequence** können auf diese Überprüfung verzichten, da mehrfach auftretende Objekte in diesem Fall erlaubt sind. Anschließend kann der eigentliche Einfügevorgang beginnen, der abhängig von der Realisierung der Redefinitions- und Teilmengenbeziehungen eines Attributes ist.

Redefinitionen Eine Redefinition definiert Gleichheit zwischen der redefinierenden und der redefinierten Menge von Instanzen. Im Standardfall, dass der Kontext des redefinierenden Assoziationsendes vom Kontext des redefinierten Assoziationsendes abgeleitet ist, definiert die Beziehung zudem eine Einschränkung des Typs der Instanzen, die dem redefinierten Ende hinzugefügt werden können. Dies ist in Abb. 8.10 dargestellt.

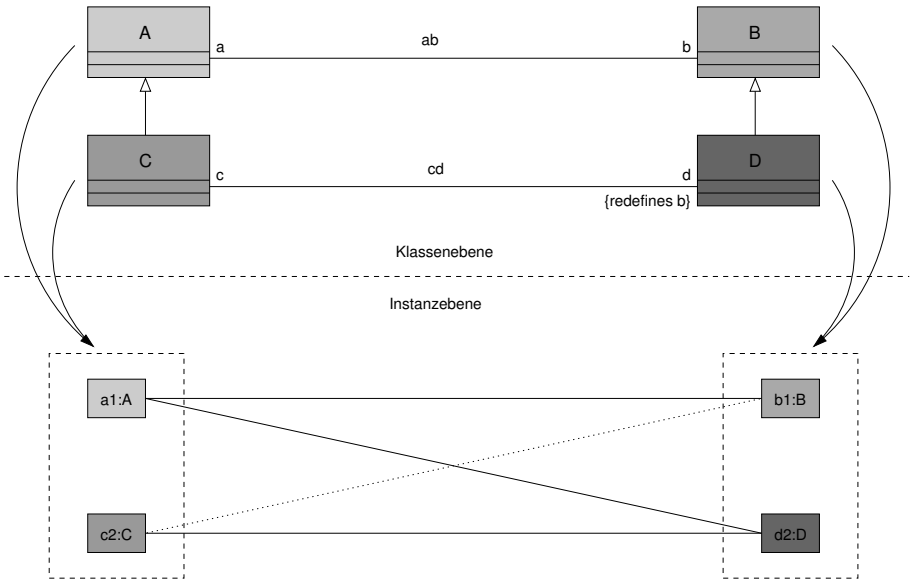


Abbildung 8.10: Auswirkung einer Redefinition auf die Instanzebene

Das Assoziationsende **d** der Assoziation **CD** redefiniert das Assoziationsende **b** der Assoziation **AB**. Somit müssen die Mengen von Instanzen, die in den Assoziationsenden **b** und **d** für eine Instanz der Klasse **C** enthalten sind, identisch sein. Gleichzeitig bedeutet die Redefinition, dass keine Verbindungen zwischen Objekten der Klassen **C** und **B** mehr erzeugt werden können, weil jede Instanz in **b** aufgrund der Redefinition typkonform zur Klasse **D** sein muss.

Redefinitionsbeziehungen können auf verschiedene Arten umgesetzt werden. Eine Möglichkeit ist die Verschmelzung von redefiniertem und redefinierendem Attribut während der Codegene-

ration, so dass lediglich eine Repräsentation für beide Attribute erzeugt wird, die durch mehrere Zugriffsoperationen zugänglich gemacht wird. Vorteile dieser Methode sind die Vermeidung von Redundanz und die höhere Effizienz zur Laufzeit. Ihr Nachteil ist der höhere Analyseaufwand, der während der Codeerzeugung erforderlich ist, weil für jedes zu generierende Attribut etwaige Redefinitionsbeziehungen untersucht werden müssen.

Die weiteren Möglichkeiten repräsentieren jedes Attribut explizit und implementieren die Redefinitionsbeziehung durch Propagation. Es werden entweder alle Einfüge- und Löschvorgänge oder alle Abfragen an alle redefinierenden und redefinierten Attribute propagiert. Im ersten Fall entsteht eine explizite Repräsentation der Redefinitionen durch gleich belegte Attribute. Vorteile dieser Methode sind ihre leichte Realisierbarkeit und die Nähe zu der in der Spezifikation angegebenen Definition. Zudem lassen sich die vergleichsweise häufig genutzten Abfrageoperationen effizient realisieren. Die letzte Möglichkeit führt alle Einfüge- und Löschoperationen nur auf dem jeweils direkt angesprochenen Attribut durch. Die Redefinition wird implementiert, indem die Abfragen eines Attributes zusätzlich die Objekte in allen redefinierten und allen redefinierenden Attributen zurückliefern. Dieser Ansatz vermeidet einen Teil der Redundanz und ist trotzdem ähnlich leicht zu realisieren wie der zuvor genannte Ansatz. Allerdings treten bei allen Operationen Laufzeitnachteile gegenüber der direkten Verschmelzung der Attribute auf, weil die Einfüge- und Löschoperationen alle redefinierten und redefinierenden Attribute „befragen“ müssen, bevor eine Operation durchgeführt werden kann, und die Abfrageoperation den Iterator redefinieren muss, damit dieser auch die redefinierenden und redefinierten Attribute einbezieht.

Es wird deutlich, dass dem Vorteil der leichteren Realisierbarkeit erhebliche Nachteile des generierten Codes gegenüberstehen, so dass die Möglichkeit der Verschmelzung durch Redefinitionsbeziehungen verbundener Attribute zur Realisierung ausgewählt wurde. Außerdem ist anzumerken, dass dieser Ansatz nur bei der Realisierung der Implementierungsvererbung durch Kopieren möglich wird, da redefinierendes und redefiniertes Attribut in der Regel nicht innerhalb derselben Klasse definiert werden.

Teilmengen Die Implementierung von Teilmengenbeziehungen zwischen Attributen erfordert ähnliche Mechanismen wie die Implementierung von Redefinitionsbeziehungen. Allerdings müssen die an einer Teilmengenbeziehung beteiligten Attribute immer explizit repräsentiert werden, so dass der favorisierte Ansatz zur Implementierung von Redefinitionen hier nicht angewendet werden kann. Die Auswirkungen einer Teilmengenbeziehung zwischen zwei Attributen ist in Abb. 8.11 dargestellt.

Abb. 8.11 zeigt ein Assoziationsende **d**, das als Teilmenge eines Assoziationsendes **b** definiert ist. Auf der Instanzebene existieren vier Instanzen der Assoziationen **AB** und **CD**. **a1b1** verbindet die Objekte **a1** und **b1** und ist eine Instanz von **AB**. Die anderen drei Assoziationsinstanzen verbinden das Objekt **c2** mit den Objekten **b2**, **d3** und **d4**. Die gestrichelt dargestellten Boxen umfassen jeweils die Objekte, die bei der Abfrage aller entgegengesetzten Enden zu einem Objekt zurückgegeben werden.

Um komfortable Implementierungen von in Teilmengenbeziehung stehenden Attributen und Assoziationsenden bereitstellen zu können, muss eine Implementierung entweder die Einfüge-

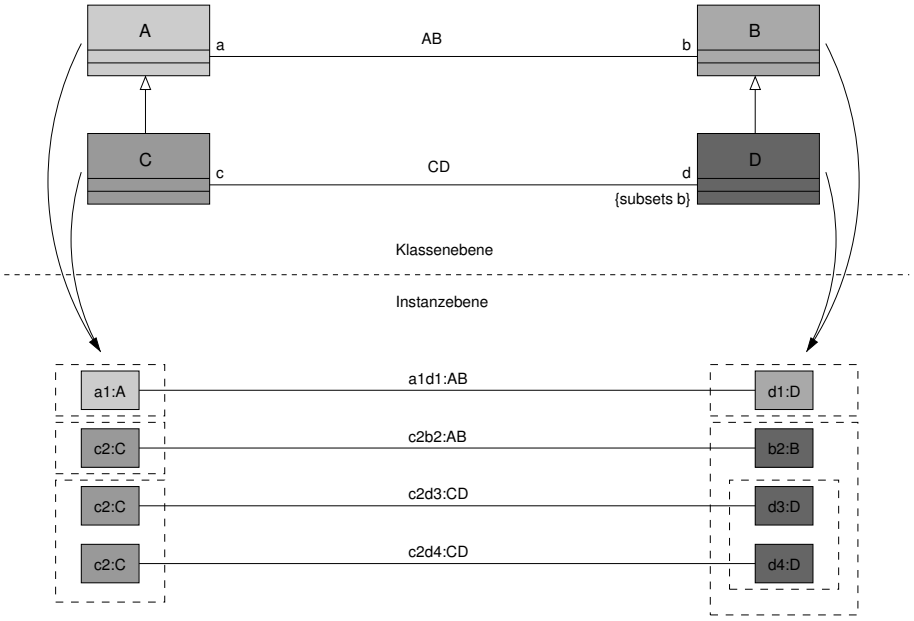


Abbildung 8.11: Auswirkung einer Teilmengenbeziehung auf die Instanzebene

und Löschoperationen propagieren, oder die Abfrageoperationen müssen alle als Teilmenge definierten Attribute einbeziehen. Die Vor- und Nachteile dieser beiden Ansätze wurden bereits in der Beschreibung der Implementierung von Redefinitionsbeziehungen genannt. Um eine redundanzfreie Implementierung zu erreichen, wurde für die MOmo-Implementierung die Propagation von Abfrageoperationen gewählt. Jede Einfüge- und Löschoperation muss sicherstellen, dass durch ihre Durchführung keine Bedingungen verletzt werden. Die Abfrageoperationen müssen Zugriff auf alle in einem Attribut abgelegten Objekte bieten. Daher müssen insbesondere die Iteratoren einer JMI-Collection-Klasse neben den direkt gehaltenen Objekten auch die in Teilmengen eines Attributes gehaltenen Objekte durchlaufen.

8.2.3 Assoziationen

Die Abbildung von Assoziationen auf Code wird in der Literatur häufig diskutiert (z. B. in [Rum87], [Nob96], [HS98], [HBR00] und [Ste01]). Allerdings werden dabei zumeist Aspekte diskutiert, die durch die JMI-Spezifikation bereits festgelegt worden sind; beispielsweise ob Assoziationen auf Objekte oder auf Methoden abgebildet werden sollen. In einem MOF-Modell können Assoziationen sowohl implizit als auch explizit dargestellt werden. Unter einer impliziten Darstellung versteht man sich wechselseitig referenzierende Attribute. Eine explizite Darstellung einer Assoziation repräsentiert jede Assoziation durch (mindestens) ein eigenes Objekt. MOF ermöglicht die Verwendung beider Darstellungen, so dass eine Abbildung von MOF auf eine Programmiersprache ebenfalls beide Darstellungen umfassen muss.

Das Hauptproblem der Abbildung von MOF-Assoziationen tritt dagegen an einer Stelle auf, die nicht diskutiert wird. Im Unterschied zu UML 1.x erlaubt MOF mehrfache Instanzen einer

Assoziation, die dieselben Objekte miteinander verbinden. Diese Möglichkeit besteht bereits in MOF 1.4; sie wird in der JMI-Spezifikation allerdings nur erwähnt und nicht abgebildet.

Um eine vollständige Abbildung der Mechanismen aus MOF nach Java zu ermöglichen, muss die Abbildung von Assoziationen gegenüber der JMI-Spezifikation verändert werden. Eine einheitliche Abbildung aller möglichen Assoziationsarten ist nur möglich, wenn die Instanzen einer Assoziation explizit repräsentiert werden. Leider hat dies zur Folge, dass die Verwendung der Assoziationen weniger komfortabel ist. In Abb. 8.12 ist die Schablone zur Erzeugung von Schnittstellen für Assoziationen dargestellt, die in einer MOmo-Implementierung verwendet wird.

```
public interface Link {
  public RefObject getFirstEnd();
  public RefObject getSecondEnd();
}

public interface <AssociationName>
  extends javax.jmi.reflect.RefAssociation {

  public boolean exists(Link link) throws javax.jmi.reflext.JmiException;

  // obere Grenze der Multiplizitaet des ersten Assoziationsendes gleich 1
  public <End1ClassName> get<End1Name>(<End2Type> opposite)
    throws javax.jmi.reflect.JmiException;

  // obere Grenze der Multiplizitaet des ersten Assoziationsendes
  // groesser als 1 und ungeordnet
  public Collection get<End1Name>(<End2Type> opposite)
    throws javax.jmi.reflect.JmiException;

  // obere Grenze der Multiplizitaet des ersten Assoziationsendes
  // groesser als 1 und ungeordnet
  public List get<End1Name>(<End2Type> opposite)
    throws javax.jmi.reflect.JmiException;

  // analog fuer zweites Assoziationsende
  ...

  public Link add(<End1Type> end1, <End2Type> end2)
    throws javax.jmi.reflext.JmiException;

  public boolean remove(Link link)
    throws javax.jmi.reflect.JmiException;
}
```

Abbildung 8.12: Erzeugung von Assoziationen

Jede Instanz einer Assoziation wird durch ein Objekt repräsentiert, das die Schnittstelle **Link** implementiert. Alle Methoden, die eine eindeutige Identifikation einer Verbindung zwischen zwei Objekten benötigen (**exists** und **remove**), verwenden diese *Link*-Objekte anstelle der Paare

zu verbindender Objekte. Die Methode **add** instanziiert die Assoziation für ein Paar von Objekten und liefert einen Verweis auf ein Link-Objekt zurück. Es ist Aufgabe des Programmierers, die Verweise auf Link-Objekte in einer Anwendung geeignet zu verwalten.

Um den Verlust an Komfort zu mildern, der durch die vorgeschlagene Realisierung von Assoziationen hervorgerufen wird, könnte eine Implementierung für Assoziationen mit eindeutigen Verbindungen zwischen Objekten die Link-Objekte ignorieren. In diesem Fall ergibt sich eine Semantik, die der in der JMI-Spezifikation vorgeschlagenen Schnittstelle entspricht. Der einzige Unterschied ist, dass der Programmierer die Link-Objekte erzeugen muss. Alternativ könnte man auf die Einheitlichkeit der Abbildung von Assoziationen verzichten und lediglich die Repräsentationen der Assoziationen, die mehrfache Verbindungen zwischen denselben Objekten erlauben, durch Anwendung der Schablone aus Abb. 8.12 erzeugen. Für alle anderen Assoziationen erhalten die Methoden **exists** und **remove** Paare von Objekten anstelle der Link-Objekte als Parameter übergeben, und die Methode **add** liefert einen bool'schen Wert anstelle des Link-Objektes zurück.

8.2.4 Datentypen

MOF unterstützt einfache und strukturierte Datentypen sowie Aufzählungstypen. Die einfachen Datentypen werden auf elementare Java-Datentypen abgebildet und bilden dadurch die Grundlage der Java-Implementierungen von MOF-Modellen. Strukturierte Datentypen enthalten mindestens ein Attribut, so dass die Definition zusammengesetzter Daten möglich ist. Der wesentliche Unterschied zwischen strukturierten Datentypen und Klassen ist, dass die Instanzen strukturierter Datentypen keine Identität besitzen. Jede Kombination von Attributbelegungen tritt also genau einmal auf. Aufzählungstypen sind Datentypen, deren Wertemenge aus einem Satz von Symbolen besteht.

8.2.4.1 Einfache Datentypen

Einfache Datentypen werden in JMI auf Klassen abgebildet, die allerdings lediglich als Platzhalter für einen Datentyp der Zielsprache dienen. MOmo erlaubt die Abbildung beliebiger Datentypen auf die MOF-Datentypen durch Einsatz des *TypeMappingModule*. Die Abbildung der einfachen MOF-Datentypen aus dem Paket **PrimitiveTypes** wird durch den JMI-Standard festgelegt und ist in Tab. 8.3 dargestellt.

Wann immer möglich, werden skalare Java-Datentypen zur Abbildung der einfachen MOF-Datentypen verwendet, weil diese einen komfortableren Umgang bei der Implementierung JMI-basierter Anwendungen ermöglichen.

8.2.4.2 Strukturierte Datentypen

Ein strukturierter MOF-Datentyp wird in einer MOmo-Implementierung auf zwei Schnittstellen abgebildet, deren Erzeugung in den Abb. 8.13 und 8.14 dargestellt wird. Für jedes Attribut des Datentyps werden Methoden zum Lesen und Schreiben erzeugt. Eine MOmo-Implementierung

MOF 2.0	Java
Boolean	boolean
Integer	int
Long	long
Float	float
Double	double
String	java.lang.String

Tabelle 8.3: Abbildung einfacher Datentypen

enthält für jeden strukturierten Datentyp zusätzlich eine Klasse, die die Schnittstelle implementiert.

```
public interface <StructTypeName>DataType {
  public <StructTypeName> get<StructTypeName>(
    // fuer jedes Attribut des Datentyps
    <AttributeName> <AccessorName>
    ) throws javax.jmi.reflect.JmiException;
}
```

Abbildung 8.13: Schnittstellen für strukturierte Datentypen

Abb. 8.13 zeigt die Schablone, die zur Erzeugung einer Java-Schnittstelle für einen strukturierten Datentyp verwendet wird. Die Funktion dieser Schnittstelle entspricht der Funktion der Schnittstellen für Klassen, die oben beschrieben wurde. Die Kapselung der Erzeugung strukturierter Datentypen durch diese Schnittstelle ermöglicht einer MOmo-Implementierung, die Instanzen des Datentyps zu verwalten. So lässt sich die doppelte Erzeugung von Instanzen mit denselben Werten vermeiden.

```
public interface <StructTypeName> extends javax.jmi.reflect.RefStruct {
  // fuer jedes enthaltene Attribut
  public <AttributeName> <AccessorName> ()
    throws javax.jmi.reflect.JmiException;
}
```

Abbildung 8.14: Schnittstellen für Instanzen strukturierter Datentypen

Abb. 8.14 zeigt die Schnittstelle, die den Zugriff auf die Instanzen eines strukturierten Datentyps ermöglicht. Die Schnittstelle enthält eine Abfragemethode für jedes Attribut des Datentyps.

8.2.4.3 Aufzählungstypen

Die aktuelle Java-Version 1.4 enthält keine Aufzählungstypen, so dass diese in JMI durch geeignete Schnittstellen und Klassen simuliert werden müssen. In der Java-Version 1.5 sind Aufzählungstypen enthalten, die in zukünftigen Versionen der JMI-Spezifikation zur Abbildung

der Aufzählungstypen eines MOF-Modells verwendet werden können. Zurzeit erfolgt die Repräsentation eines Aufzählungstyps in JMI durch die in Abb. 8.15 angegebenen Schablonen.

Im Unterschied zu den meisten anderen Elementen einer JMI-Implementierung legt die JMI-Spezifikation für Aufzählungstypen sowohl die Schnittstelle als auch die Implementierung fest.

8.3 Modellunabhängige Bestandteile

Die modellunabhängigen Bestandteile sind die gemeinsame Basis aller MOmo-Metamodellimplementierungen. Sie realisieren die grundlegenden Mechanismen, die zum Umgang mit Metamodellen benötigt werden. Die reflexive Schnittstelle ermöglicht es, Applikationen auf der Basis eines gemeinsamen Meta-Metamodells für eine Klasse von Metamodellen zu entwickeln. Beispiele für solche Applikationen sind der XML-Generator des MOmo-Baukastens (siehe Kapitel 9, Abschnitt 9.3.2) und die Komponenten der XMI-Schnittstelle, die jede MOmo-Metamodellimplementierung enthält (siehe Abschnitt 8.3.2).

8.3.1 Reflexion

Die JMI-Spezifikation definiert eine Schnittstelle, die Mechanismen zum Abfragen der Eigenschaften von Objekten ermöglicht. Aufbauend auf dieser Schnittstelle können Systemkomponenten unabhängig von der genauen Struktur eines Objektmodells erstellt werden. Dies ermöglicht beispielsweise die Programmierung einer gemeinsamen Schnittstelle aller durch MOF-Modellelemente aufgebauten Modelle zu XML, die im folgenden Abschnitt 8.3.2 beschrieben wird.

Der Reflexionsmechanismus, der in der JMI-Spezifikation beschrieben wird, ist eine Umsetzung des entsprechenden Teils der MOF 1.x Spezifikation. Die Version 2.0 der MOF-Spezifikation enthält eine deutlich veränderte Definition dieser Schnittstelle. Reflexion wird in den Paketen *Reflection* und *CMOFReflection* definiert und durch Vereinigungsbeziehungen in die Pakete *EMOF* und *CMOF* integriert. Durch diese Art der Definition ist die reflexive Schnittstelle in MOF 2.0 weitaus enger mit der Definition der sonstigen Elemente verbunden als in MOF 1.x, da die einzelnen Attribute und Methoden in die Klassen des Metamodells eingefügt werden, und nicht ausschließlich in Basisklassen gehalten werden, die von den Klassen des Metamodells beerbt werden. Zur Realisierung der reflexiven MOF-Schnittstelle wird keine Unterstützung durch die gewählte Programmiersprache benötigt. Der MOmo-MOF-Codegenerator erzeugt die reflexive Schnittstelle, ohne auf den Reflexionsmechanismus aus Java zurückzugreifen.

Die Reflexionsschnittstelle ist die gemeinsame Basis aller MOF-Modelle. Entsprechend bildet die Umsetzung der Reflexionsschnittstelle die Basis aller durch eine JMI-Abbildung implementierten MOF-Modelle. Sie wird auch zur Umsetzung der JMI-Abbildung selbst verwendet, z. B. bei der Abbildung explizit repräsentierter Assoziationen, die eine Synchronisation der im

```
public interface <EnumerationName> extend javax.jmi.reflect.RefEnum {}

public final class <EnumerationName>Enum implements <EnumerationName> {
  // fuer jedes Element der Auszaehlung
  public static final <EnumerationName>Enum <LiteralIdentifier>
    = new <EnumerationName>Enum("<LiteralName>");

  private static final List typeName;
  private final String literalName;

  static {
    java.util.ArrayList temp = new ArrayList();
    // für jeden Teil des voll qualifizierten Namens
    temp.add("<FullyQualifiedNamePart>");

    typeName = java.util.Collections.unmodifiableList(temp);
  }

  private <EnumerationName>Enum(String literalName) {
    this.literalName = literalName;
  }

  public static <EnumerationName> forName(String value) {
    // für jedes Element des Aufzaehlungstyps
    if (value.equals("<LiteralName>")) return <LiteralIdentifier>;

    throw new IllegalArgumentException("<Message>");
  }

  public String toString() { return literalName; }
  public List refTypeName() { return typeName;  }
  public int hashCode() { return literalName.hashCode(); }

  public boolean equals(Object o) {
    if (o instanceof <EnumerationName>Enum) return (o==this);
    else if (o instanceof <EnumerationName>)
      return (o.toString().equals(literalName));
    else return ((o instanceof javax.jmi.reflect.RefEnum) &&
       ((javax.jmi.reflect.RefEnum)o).refTypeName().equals(typeName) &&
       o.toString().equals(literalName));
  }

  protected Object readResolve() throws java.io.ObjectStreamException {
    try {
      return forName(literalName);
    } catch (IllegalArgumentException iae) {
      throw new java.io.InvalidObjectException(iae.getMessage());
    }
  }
}
```

Abbildung 8.15: Aufzählungstypen in JMI

Assoziationsobjekt enthaltenen Daten mit den in den Assoziationsenden enthaltenen Daten erfordern.

Die Programmierung mit Hilfe der reflexiven Schnittstelle erfolgt in der Regel in zwei Schritten. Im ersten Schritt werden die Eigenschaften eines Objektes erfragt, und im zweiten Schritt werden eine oder mehrere Eigenschaften verwendet. Um die Eigenschaften eines Objektes abfragen zu können, muss auf die Klasse zugegriffen werden, die die Eigenschaften des Objektes festlegt. Zu diesem Zweck enthält jedes Objekt einen Verweis auf seine (Meta-)Klasse, der über einen Methodenaufruf abgefragt werden kann. Anschließend lassen sich alle oder bestimmte Eigenschaften über die entsprechenden Methoden der Metaklasse abfragen und verwenden. In Abb. 8.16 sind die Basisschnittstellen der reflexiven Schnittstelle dargestellt.

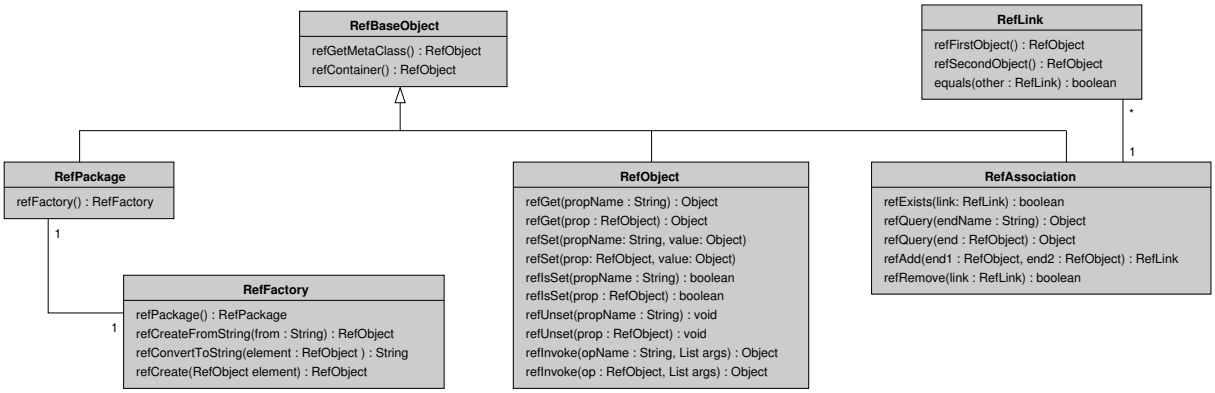


Abbildung 8.16: Reflexive Basisschnittstellen

Aus der engen Integration der reflexiven Schnittstelle resultiert ein Problem bei der Erzeugung des Codes der reflexiven Schnittstelle. Wann immer eine MOF-Klasse eine Operation enthält, die zur reflexiven Schnittstelle gehört, kollidiert die Abbildung dieser Operation mit der zusätzlich für jede Klasse erzeugten Methode der reflexiven Schnittstelle. Die Klasse **Object** des MOF-Metamodells enthält z. B. die Operation **getMetaClass**. Die Abbildung der Klasse **Object** enthält daher eine Methode **getMetaClass**. Eine gleichnamige Methode würde für jede Klasse eines MOF-Modells erzeugt, um die Implementierung der reflexiven Schnittstelle bereitzustellen. Eine Implementierung der Klasse **Object** enthielte dann zwei gleichnamige Methoden. Um dies zu vermeiden, erhalten alle Methoden der reflexiven Schnittstelle das Präfix **ref**, und die Verwendung dieses Präfixes bei Operationen eines MOF-Modells wird verboten.

Die Grundlage der Schnittstelle ist die Klasse **RefBaseObject**, die Mechanismen zum Abfragen der Klasse und des umgebenden Elementes ermöglicht. Auf **RefBaseObject** bauen die Klassen **RefPackage**, **RefObject** und **RefAssociation** auf, die die Grundelemente eines MOF-Modells abbilden. Diese Klassen enthalten reflexive Varianten der Methoden, die in den Schablonen zur Erzeugung der modellabhängigen Anteile einer MOmo-Implementierung enthalten sind. Die Klasse **RefFactory** ermöglicht die Erzeugung neuer Instanzen und enthält außerdem Methoden zur Umwandlung eines Objektes in eine Zeichenkette sowie zur Umwandlung einer Zeichenkette in ein Objekt.

In Abb. 8.16 fehlen die Schnittstellen zur Repräsentation von Datentypen. Wie in den oben angegebenen Schablonen stützen sich die modellabhängigen Schnittstellen für Datentypen ebenfalls auf Elemente der reflexiven Schnittstelle. Die Schnittstelle **RefStruct** repräsentiert strukturierte Datentypen und erbt von **RefBaseObject**. Die Schnittstelle enthält reflexive Varianten der in der Schablone angegebenen Methoden sowie eine Methode zum Abfragen der enthaltenen Attribute. Aufzählungstypen werden durch Objekte repräsentiert, die die Schnittstelle **RefEnum** implementieren. Diese Schnittstelle erbt *nicht* von **RefBaseObject** und definiert Methoden zum Vergleichen von Werten des Aufzählungstyps.

8.3.2 XMI

Um sinnvoll einsetzbar zu sein, müssen Modellierungswerkzeuge Modelle persistent speichern können. Aus diesem Grund wird in der JMI-Spezifikation eine Schnittstelle festgelegt, die das Laden und Speichern von XMI-Dokumenten (siehe Kapitel 7) ermöglicht. XMI definiert ein Austauschformat für alle Modelle, deren Meta-Metamodell MOF ist. Daher bietet sich die Verwendung der reflexiven Schnittstelle einer MOmo-Implementierung zur Realisierung der XMI-Schnittstelle an.

```
public interface XmiReader {
  public java.util.Collection read(String URI,
                                   javax.jmi.reflect.RefPackage extent)
    throws java.io.IOException, javax.jmi.xmi.MalformedXMIException;

  public java.util.Collection read(java.io.InputStream stream,
                                   String URI,
                                   javax.jmi.reflect.RefPackage extent)
    throws java.io.IOException, javax.jmi.xmi.MalformedXMIException;
}


public interface XmiWriter {
  public void write(java.io.OutputStream stream,
                    javax.jmi.reflect.RefPackage extent,
                    String xmiVersion) throws java.io.IOException;

  public void write(java.io.OutputStream stream,
                    java.util.Collection objects,
                    String xmiVersion) throws java.io.IOException;
}
```

Abbildung 8.17: JMI-Schnittstellen zu XMI

Abb. 8.17 zeigt die XMI-Schnittstelle, die eine JMI-konforme Implementierung bereitstellen muss. Die Schnittstelle besteht aus zwei Java-Schnittstellen **XmiReader** und **XmiWriter**. Der **XmiReader** enthält zwei Methoden, die das Einlesen eines XMI-Dokumentes ermöglicht. Nach dem Einlesen kann auf die im XMI-Dokument enthaltenen Elemente über die

JMI-Schnittstellen zugegriffen werden. Der **XmiWriter** ermöglicht die Serialisierung einer Repräsentation eines MOF-Modells, so dass unter anderem XMI-Dokumente erzeugt werden können.

Für jede der beiden Schnittstellen wird in einer MOmo-Implementierung eine Klasse erzeugt, die die Schnittstelle implementiert. Die Klassen implementieren im Wesentlichen die Regeln zur XMI-Dokumentproduktion aus der Abbildung von MOF 2.0 auf XMI [Obj03b] bzw. deren Umkehrung. Zur Implementierung der Klassen werden die Reflexionsmechanismen verwendet. So lassen sich die Klassen zum Lesen und Schreiben aller Modelle verwenden, deren Meta-Metamodell MOF ist.

8.4 Zusammenfassung

In diesem Kapitel wurde die Erzeugung einer Java-Implementierung eines MOF 2.0 Modells beschrieben. Eine solche Implementierung ist Teil eines jeden MOmo-Generators und kann außerdem als Grundlage für viele Anwendungen insbesondere im Bereich Werkzeugbau verwendet werden. MOmo-Implementierungen JMI-konformer Schnittstellen orientieren sich weitgehend an der JMI-Spezifikation. Da diese aber noch nicht für MOF 2.0 aktualisiert wurde, mussten einige Anpassungen und Erweiterungen vorgenommen werden, um verändert definierte und neue Modellelemente aus MOF 2.0 abbilden zu können. Die Veränderungen umfassen insbesondere die Abbildungen der Attribute und Assoziationsenden, für deren Spezifikation MOF 2.0 erweiterte Mechanismen zur Verfügung stellt. Die gewählte Abbildung basiert auf der Java-Version 1.4 und führt daher Typüberprüfungen zur Laufzeit durch. Beim Übergang auf die Java Version 5.0 können diese Typüberprüfungen durch Verwendung der neuen Möglichkeiten zur generischen Programmierung ersetzt werden.

Jede MOmo-Implementierung enthält modellabhängige und modellunabhängige Bestandteile. Die modellabhängigen Bestandteile unterscheiden sich je nach Modell, d. h. eine JMI-Schnittstelle für UML unterscheidet sich von der JMI-Schnittstelle für MOF. Die modellabhängigen Bestandteile bilden die Modellelemente eines MOF-Modells auf Java-Schnittstellen und -Klassen ab. Modellunabhängige Bestandteile sind dagegen für alle Implementierungen einer JMI-Schnittstelle gleich. Sie bilden die Basis der Implementierung und ermöglichen die persistente Speicherung von Modellen (XMI-Schnittstelle) sowie die Verarbeitung unbekannter Modellelemente durch Reflexion.

Kapitel 9

Generische Komponenten

Das Ziel des MOmo-Baukastens ist, die Entwicklung von Codegeneratoren für MOF-basierte Modellierungssprachen zu vereinfachen. Diese Aufgabe kann teilweise durch den im Baukasten enthaltenen MOF-Codegenerator erfüllt werden, der für jedes MOF-Modell eine Java-Implementierung mit JMI-konformen Schnittstellen erzeugen kann. Der Aufbau der Metadatenimplementierung, die den Parser und das Metamodell für einen MOmo-Codegenerator bereitstellen, wurde in Kapitel 8 beschrieben.

Um eine hohe Flexibilität der Codegeneratoren bei gleichzeitiger Reduzierung des Entwicklungsaufwandes zu erreichen, wird die Codegenerierung durch Anwendung von Schablonen realisiert. Neben einer JMI-Implementierung eines MOF-Modells enthält ein MOmo-Generator daher generische Komponenten, die eine Ablaufumgebung für Schablonen implementieren. Konzeption und Implementierung dieser Komponenten werden in diesem Kapitel erläutert.

In Abschnitt 9.1 wird der grundlegende Ansatz beschrieben, der für die Realisierung der Ablaufumgebung innerhalb eines MOmo-Generators ausgewählt wurde. Die nachfolgenden Abschnitte stellen die einzelnen Komponenten im Detail dar. Abschnitt 9.2 erläutert die Arbeitsweise der Steuerkomponente, die alle MOmo-Codegeneratoren verwenden, und Abschnitt 9.3 beschreibt die Realisierung der Schablonenablaufumgebung.

9.1 Ansatz

Die Aufgabe der Codegeneratoren, die mit Hilfe des MOmo-Baukastens entwickelt werden sollen, besteht in der Erzeugung von Artefakten für ein Modell, das durch die Elemente einer MOF-basierten Modellierungssprache beschrieben wird. Auf die einzelnen Elemente des Modells kann über die Schnittstellen zugegriffen werden, die in Kapitel 8 vorgestellt wurden. Die Umwandlung der Metaobjekte, die durch den Reader erzeugt werden, kann entweder direkt programmiert oder durch eine Ablaufumgebung für Schablonen realisiert werden.

Verglichen mit direkt programmierten Lösungen können Schablonen leichter und schneller verändert und an bestimmte Anwendungsfälle angepasst werden. Um die gewünschte hohe Flexibilität bei der Codegenerierung zu erreichen, soll der Code durch Anwendung von Schablonen erzeugt werden. Dies ermöglicht eine weitergehende Reduzierung des Entwicklungsaufwandes gegenüber direkt programmierten Lösungen. Das Ziel ist, dass sich der Entwickler eines Codegenerators vollständig auf die Umsetzung von Modellelementen auf die zu erzeugenden Artefakte konzentrieren kann. Der MOmo-Baukasten muss daher neben einem MOF-Codegenerator auch eine Ablaufumgebung für Schablonen bereitstellen, um dieses Ziel zu erreichen.

Ein MOmo-Generator enthält dazu neben der JMI-Implementierung für die zu übersetzende Sprache zwei weitere Komponenten. Diese Komponenten sind generisch, d. h. alle MOmo-Generatoren enthalten die gleichen Komponenten. Die generischen Komponenten stellen die Ablaufumgebung für die Schablonen bereit. Eine Komponente wandelt die Metaobjekte in eine Menge sog. *Kontexte* um, die die im Modell enthaltenen Informationen zur Auswertung durch die Schablonen aufbereiten. Ein Kontext ist eine Repräsentation der zu übersetzenden Modelle oder Modellausschnitte, die auf die Auswertung durch eine Schablone abgestimmt ist. Die Ausprägung eines Kontextes ist abhängig vom verwendeten Schablonenmechanismus. Beispielsweise ist eine Menge von „Schlüssel-Wert“-Paaren ein geeigneter Kontext, falls der Schablonenmechanismus *Velocity* eingesetzt wird. Wird dagegen *XSLT* als Schablonenmechanismus verwendet, ist ein XML-Dokument ein geeigneter Kontext. Eine weitere Komponente wendet die Schablonen auf die Kontexte an und erzeugt auf diesem Wege die Artefakte. Abb. 9.1 zeigt die minimale Architektur eines MOmo-Generators, d. h. alle zwingend erforderlichen Bestandteile.

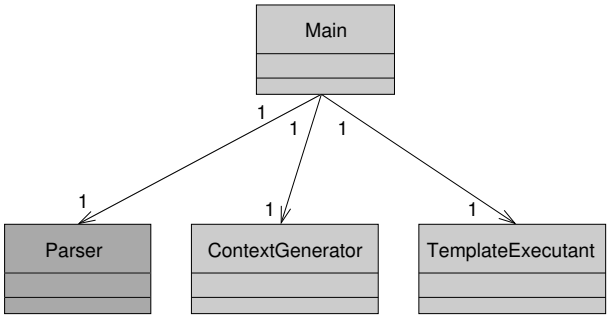


Abbildung 9.1: Minimale Architektur eines MOmo-Generators

Jeder MOmo-Generator enthält eine *Steuerkomponente*, die durch die Klasse **Main** implementiert wird und den Generierungsprozess steuert. Die Steuerkomponente ruft zunächst einen XMI-Reader auf, um eine Darstellung des zu übersetzenden Modells aufzubauen, die über JMI-konforme Schnittstellen zugreifbar ist. Im (optionalen) zweiten Schritt wird eine beliebige Anzahl von MOmo-Modulen auf Menge der vom Reader erzeugten Objekte angewendet, um Anpassungen an bestimmte Anwendungsfälle vorzunehmen. Im dritten Schritt werden die Elemente der JMI-konformen Darstellung durch den *Kontextgenerator*, der durch die Klasse **ContextGenerator** implementiert ist, in einen Kontext eingefügt, der den Schablonen alle Informationen des Modells in einer geeigneten Form bereitstellt. Im dritten Schritt wendet der

Schablonenausführer die Schablonen auf die Kontexte an, um die gewünschten Artfakte zu erzeugen. Dieser Schritt wird von der Klasse **TemplateExecutant** durchgeführt, die für jede Schablone die Menge der auszuwertenden Kontexte bestimmt und die Schablonen darauf ausführt. Nach dem vierten Schritt ist die Erzeugung der Artefakte abgeschlossen, so dass ein Generator im produktiven Einsatz häufig nur diese vier Schritte ausführen wird. Insbesondere während der Entwicklungsphase einer Schablone kann es sinnvoll sein, das erzeugte Artefakt durch einen Formatierer in eine besser lesbare Form zu bringen, weil dies die Fehlererkennung erleichtert. In einem optionalen fünften Schritt werden daher Formatierer auf die erzeugten Artefakte angewendet.

In den folgenden Abschnitten werden die Steuerkomponente und die Schablonenablaufumgebung genauer beschrieben. Zunächst erläutert Abschnitt 9.2 die Funktion der Steuerkomponente. Anschließend wird in Abschnitt 9.3 die Schablonenablaufumgebung erläutert.

9.2 Steuerkomponente

Der oben beschriebene Ablauf wird von allen MOmo-Codegeneratoren durchgeführt und ist vollkommen unabhängig von der Eingabesprache, die ein Generator verarbeiten soll. Aus diesem Grund ist auch die Steuerkomponente eines MOmo-Codegenerators unabhängig von der Eingabesprache und kann für alle Generatoren unverändert verwendet werden. Im Einzelnen besteht die Aufgabe der Steuerkomponente aus dem Auswerten der Konfigurationsvariablen, die die Eingabedatei sowie alle weiteren Bestandteile des Generators festlegen. Es werden die Konfigurationsvariablen **momoc.parser**, **momoc.module.<*name*>**, **momoc.contextgenerator**, **momoc.templateexecutant** und **momoc.formatter.<*name*>** verarbeitet.

Die Variablen **momoc.parser**, **momoc.contextgenerator** und **momoc.templateexecutant** müssen in jeder Konfiguration mit den vollständigen Namen der Java™-Klassen belegt werden, die zum Einlesen des XMI-Dokumentes, zum Aufbauen der Kontexte und zum Ausführen der Schablonen verwendet werden sollen. Zusätzlich können beliebig viele Module durch **momoc.module.<*name*>** und Formatierer durch **momoc.formatter.<*name*>** angegeben werden, die auf die interne Objektrepräsentation des eingelesenen Modells bzw. die generierten Artefakte angewendet werden. Die angegebenen Variablen müssen sich im kursiv dargestellten Anteil unterscheiden und werden ebenfalls durch vollständige Namen der zu verwendenden Klassen belegt.

Die Klassen, die mit Hilfe der Steuerkomponente zu einem MOmo-Codegenerator verbunden werden können, müssen bestimmte Schnittstellen erfüllen. Diese Schnittstellen werden in den folgenden Abschnitten kurz dargestellt. Jeder der oben beschriebenen Konfigurationvariablen ist genau eine Schnittstelle zugeordnet, die eine Klasse implementieren muss, um als gültige Belegung einsetzbar zu sein.

9.2.1 Die Reader-Schnittstelle

Ein MOmo-konformer Reader muss die Schnittstelle **Reader** implementieren, die in Abb. 9.2 dargestellt ist. Die Schnittstelle enthält die Methoden **setInputPaths** und **read**. Die Methode **setInputPaths** ermöglicht die Übergabe einer Menge von Eingabedateien, die jeweils ein Fragment eines MOF-Modells enthalten. Die Methode **read** kombiniert die Elemente aus den Eingabedokumenten und liefert eine Objektrepräsentation zurück, die aus Instanzen der in Kapitel 8 beschriebenen Klassen besteht.

```
package de.unibw_muenchen.momoc.reader;

public interface Reader {
    public void setInputPaths(java.util.List pathList);
    public java.util.Collection read();
} // Reader
```

Abbildung 9.2: MOmo-konforme Reader-Schnittstelle

9.2.2 Modul-Schnittstelle

Jedes benutzerdefinierte Modul, das innerhalb eines Generierungsprozesses verwendet werden soll, muss die Schnittstelle **Module** implementieren, die in Abb. 9.3 dargestellt ist. Die Steuerkomponente übergibt die Menge der vom Reader erzeugten Metaobjekte durch Aufruf der Methode **init** an ein Modul. Anschließend ruft die Steuerkomponente die Methode **perform** auf, um die Manipulation der Objektrepräsentation durchzuführen.

```
package de.unibw_muenchen.momoc.modules;

public interface Module {
    public void init(java.util.Collection objects);
    public void perform();
} // Module
```

Abbildung 9.3: MOmo-konforme Modul-Schnittstelle

9.2.3 Kontextgenerator-Schnittstelle

Kontextgeneratoren, die mit der MOmo-Steuerkomponente verwendet werden sollen, müssen die Schnittstelle **ContextGenerator** implementieren. Diese ist in Abb. 9.4 gezeigt. Durch Aufruf der Methode **init** wird die Objektrepräsentation an einen Kontextgenerator übergeben. Zusätzlich kann ein Name für das Modell übergeben werden. Ein Aufruf der Methode **generate** erzeugt die Kontexte für die Objektrepräsentation, die durch die Methode **getContexts** abgefragt werden können.

```
package de.unibw_muenchen.momoc.generator;

public interface ContextGenerator {
    public void init(java.util.Collection metaObjects, String modelName);
    public void generate();
    public java.util.Map getContexts();
} // ContextGenerator
```

Abbildung 9.4: MOmo-konforme Kontextgenerator-Schnittstelle

9.2.4 Schablonenausführer-Schnittstelle

Abb. 9.5 zeigt die Schnittstelle, die eine Komponente zur Ausführung von Schablonen implementieren muss, um durch die MOmo-Steuerkomponente verwendet werden zu können. Die Methode **init** ermöglicht die Übergabe der Kontexte an den Schablonenausführer. Die Methode **generate** erzeugt die Menge der erzeugten Artefakte, die durch die Methode **getArtefacts** zurückgeliefert werden.

```
package de.unibw_muenchen.momoc.generator;

public interface TemplateExecutant {
    public void init(java.util.Map contexts);
    public void generate();
    public java.util.Collection getArtefacts();
} // TemplateExecutant
```

Abbildung 9.5: MOmo-konforme Schablonenausführer-Schnittstelle

9.2.5 Formatierer-Schnittstelle

In Abb. 9.6 ist die Schnittstelle dargestellt, die MOmo-konforme Implementierungen eines Artefakt-Formatierers implementieren müssen. Die Methoden **addFile** und **setFiles** ermöglichen die Übergabe einer Datei bzw. einer Menge von Dateien an eine Instanz eines Formatierers. Die Methode **format** formatiert alle übergebenen Dateien.

```
package de.unibw_muenchen.momoc.formatter;

public interface Formatter {
    public void addFile(java.io.File file);
    public void setFiles(java.util.Set files);
    public void format();
} // Formatter
```

Abbildung 9.6: MOmo-konforme Formatierer-Schnittstelle

9.3 Schablonenablaufumgebung

In diesem Abschnitt wird die generische Schablonenablaufumgebung dargestellt, die der MOmo-Baukasten bereitstellt. Es gibt eine Vielzahl von Möglichkeiten, eine schablonenbasierte Codeerzeugung auf der Basis existierender Implementierungstechnologien zu realisieren. In Abschnitt 9.3.1 werden zunächst die verschiedenen Möglichkeiten der technischen Realisierung eines Schablonenmechanismus' auf der Basis existierender Implementierungen beschrieben. Aufbauend auf den Beschreibungen wird die Auswahl begründet, die für die Realisierung im MOmo-Baukasten getroffen wurde. Die Abschnitte 9.3.2 und 9.3.3 enthalten Darstellungen der Komponenten der Schablonenablaufumgebung, die in jedem MOmo-Generator enthalten sind.

9.3.1 Schablonenmechanismen

Eine der Anforderungen für den MOmo-Baukasten ist, dass die Komponenten des Baukastens auf Standardkomponenten basieren sollen. Aufgrund dieser Anforderung wurde die Möglichkeit, einen eigenen, speziell an die Anforderungen eines Rahmenwerkes für Codegeneratoren angepassten Schablonenmechanismus zu verwenden, von vorherein verworfen. Stattdessen wurden die Möglichkeiten untersucht, erhältliche Mechanismen einzusetzen. Die betrachteten Mechanismen können in Ausführungsmaschinen für Schablonen für spezielle Anwendungsdomänen und allgemein verwendbare Skriptsprachen unterschieden werden. Im Folgenden werden die Schablonensprachen *Velocity* und *XSLT* sowie einige ausgewählte Skriptsprachen vorgestellt.

9.3.1.1 Velocity

Das Jakarta-Projekt [Jak] entwickelt *Velocity* [Vel], eine Java-Implementierung eines Schablonenmechanismus. Velocity kann sowohl alleinstehend zur Realisierung von Anwendungen in den Bereichen Reporterzeugung und Datentransformation verwendet werden als auch in Anwendungen integriert werden. Beispiele für Anwendungen im UML- und MDA-Umfeld, die Velocity einsetzen, sind ArgoUML [Arg] und AndroMDA [And]. Ein weiteres Anwendungsgebiet von Velocity ist die Programmierung von Servlets. Die Aufgabe besteht hier meist in der dynamischen Erzeugung eines kleinen sich verändernden Teils eines ansonsten statischen HTML- oder XML-Dokumentes.

Der Sprachumfang von Velocity ist im Verhältnis zu den anderen hier beschriebenen Sprachen sehr gering. Die Sprache enthält die Möglichkeit, Variablen einzuführen und mit einem Wert zu belegen (**#set**) sowie einfache Kontrollstrukturen, die die bedingte (**#if/#elsif/#else**) und wiederholte (**#foreach**) Ausführung des Codes einer Schablone ermöglichen. In den Bedingungen der Kontrollstrukturen sind die logischen Operatoren der Programmiersprache Java erlaubt (**==, !=, !, <, <=, >, >=, &&,** ||).

Des Weiteren enthält Velocity Mechanismen, die die Aufteilung einer Schablone in mehrere Dateien erlauben. Mit Hilfe dieser Mechanismen lassen sich Velocity-Schablonen strukturieren. Durch die Anweisung **include** kann ein beliebiges Textdokument in das Ergebnis der Abarbeitung einer Schablone eingefügt werden. Dieser Mechanismus ist gut geeignet, um im Rahmen der Codegenerierung fixe Bestandteile des Codes von sich ändernden zu trennen. Die Lesbarkeit der Dokumente lässt sich dadurch deutlich verbessern. Durch die Anweisung **parse** lässt sich der Code einer Schablone auf mehrere Dateien verteilen. Die Anweisungen, die in einer durch **parse** eingebundenen Schablone stehen, werden zur Laufzeit in den Code der aufrufenden Schablone eingefügt und ausgeführt.

Die Übergabe von Variablen aus einer Anwendung an eine Velocity-Schablone erfolgt über einen sog. *Kontext*. Innerhalb dieses Kontextes werden die Variablen über ihren Namen identifiziert. Der Aufruf **context.put("name", "MOF")** erzeugt im Kontext **context** die Variable **name** und belegt diese mit der Zeichenkette „MOF". In einer Velocity-Schablone kann durch **$name** auf die Variable zugegriffen werden.

```
import java.util.Collection;
import java.util.List;

public interface $name extends

#set ($last = $superclasses.size())
#if ($last == 0)
    javax.jmi.reflect.RefObject
#else
  #foreach( $class in $superclasses )
    #parse("GetFullName.vm")
    #if ($velocityCount < $last)
      ,
    #end
  #end
#end

{
#foreach ($attribute in $ownedAttributes)
  #parse("Attribute.vm")
#end

...

} // end of interface
```

Abbildung 9.7: Velocity-Schablone zur Erzeugung einer JMI-Schnittstelle

Abb. 9.7 zeigt einen Ausschnitt der Erzeugung einer JMI-Schnittstelle für eine Klasse. Der Ausschnitt greift zunächst auf die Variable **$name** zu, um den Namen der Schnittstelle festzulegen. Anschließend wird die Liste der Superklassen erzeugt, indem die Variable **$superclasses** aus-

gewertet wird. Hier zeigt sich die enge Integration von Velocity und der Programmiersprache Java, die es ermöglicht, Methoden aufzurufen und die Elemente aus Java-Collections auszulesen. Durch die Anweisung **#set** wird der Variablen **last** die Anzahl der Superklassen zugewiesen. Besitzt die Klasse keine Superklassen, erhält sie eine vorgegebene Superklasse, andernfalls wird die Liste der Superklassen erzeugt.

Um die Liste der Superklassen zu erzeugen, wird eine **#foreach**-Anweisung eingesetzt. Die **#foreach**-Anweisung iteriert über die Elemente der Variable **$superclasses** und weist in jedem Schleifendurchlauf das an der aktuellen Position enthaltene Element der Variable **$class** zu. Voraussetzung ist, dass die Variable **$superclasses** mit der Instanz einer Klasse des Java-Collection-API belegt ist.

Alle Superklassen müssen mit ihrem vollständigen Namen angegeben werden, um eindeutig identifizierbar zu sein. Die entsprechende Funktionalität wird durch die Schablone „GetFullName.vm“ bereitgestellt, die durch eine **#parse**-Anweisung in die Schablone zur Erzeugung der Schnittstelle eingefügt wird. In gleicher Weise wird Code zum Erzeugen der JMI-konformen Methoden für die Attribute einer Klasse eingebunden.

Anhand des Beispiels lässt sich erkennen, dass die Stärke von Velocity in der Unterstützung variabler Codegenerierung für strukturell einfache Anwendungsfälle liegt. Die Syntax der Sprache erlaubt sehr kompakte Formulierungen für die am häufigsten verwendeten Konstrukte. Durch die enge Integration mit Java ist auch die Verarbeitung geschachtelter Strukturen möglich. Dazu ist es allerdings notwendig, Objektreferenzen innerhalb einer Schablone aufzulösen, weil der Kontext ausschließlich einfache *Schlüssel*=*Wert*-Paare enthält. Wann immer also geschachtelte Strukturen aufgelöst werden müssen, muss innerhalb der Schablonen Java-Programmcode eingefügt werden. In dem in Abb. 9.7 dargestellten Beispiel ist dies in jeder Schablone notwendig, die durch eine **#parse**-Anweisung eingebunden wird. Die Schablone „GetFullName“ muss die übergeordneten Modellelemente auswerten, d. h. z. B. die Java-Methode **getNamespace** der JMI-Schnittstelle aufrufen, und die Schablone „Attribute.vm“ benötigt Zugriff auf die Eigenschaften der enthaltenen Attribute.

9.3.1.2 XSLT

XSL Transformations (XSLT) definiert eine deklarative Sprache zur Transformation von XML-Dokumenten. Zusätzlich enthält XSLT Elemente funktionaler Programmiersprachen. Weitere Elemente funktionaler Sprachen sind zwar nicht Bestandteil des XSLT-Standards, können aber nachgebildet werden [Nov01]. Ein XML-Dokument wird durch Anwendung einer Transformation in ein beliebiges Artefakt umgeformt. Die XSLT-Spezifikation [Cla99b] definiert Syntax und Semantik der Elemente der Transformationssprache.

Die Transformationen werden in XSLT durch sog. *Stylesheets* beschrieben. Ein Stylesheet ist ein XML-Dokument, das eine oder mehrere Schablonen enthält. Jede Regel wird innerhalb eines Stylesheets durch eine Schablone ausgedrückt. Da Stylesheets selbst XML-Dokumente sind und daher durch Transformationen erzeugt werden können, kann XSLT für beliebig dynamische Anwendungen verwendet werden.

Eine Transformation in XSLT ist die Anwendung einer Menge von Regeln, die die Umformung eines Quelldokumentes in ein Zieldokument festlegen. Die Durchführung einer Transformation wird durch Muster gesteuert. Jede Umformungsregel wird durch eine Schablone repräsentiert, die einem bestimmten Muster zugeordnet ist. Während der Transformation wird im Quelldokument nach Elementen gesucht, die zu einem dieser Muster passen. Wird ein passendes Element gefunden, wird eine Schablone instantiiert, um den entsprechenden Teil des Zieldokumentes aufzubauen. Während einer Transformation sind beliebige Veränderungen des Quelldokumentes möglich, weil Quell- und Zieldokument vollständig voneinander getrennt sind.

Eine Schablone kann sowohl Elemente enthalten, die direkt in das Zieldokument kopiert werden, als auch Elemente, die weitere Instruktionen der Transformation ausdrücken. Wenn eine Schablone instantiiert wird, werden alle enthaltenen Instruktionen ausgeführt und durch das entstehende Teildokument ersetzt. Die Instruktionen innerhalb einer Schablone können alle Knoten eines XML-Dokumentes auswählen und/oder auswerten, die sich unterhalb des aktuellen Knotens befinden. Jeder Transformationsschritt, der auf einen untergeordneten Knoten angewendet wird, wird rekursiv auf dieselbe Art durchgeführt. Die Abarbeitung eines Stylesheets beginnt daher mit dem Suchen des Wurzelknotens des XML-Dokumentes und dem Finden und Anwenden passender Schablonen.

Während der Durchführung einer Transformation kann der Fall eintreten, dass mehrere Schablonen für einen Knoten anwendbar sind. In diesem Fall wird trotzdem nur genau eine Schablone angewendet. Die Auswahl der Schablone wird in Abschnitt 5.5 der XSLT-Spezifikation beschrieben. Zur Auflösung eines Konfliktes wird zunächst die Reihenfolge der Import-Beziehungen und anschließend die Priorisierung der Schablonen ausgewertet. Die Priorität einer Schablone kann explizit festgelegt werden. Für alle zur Auswahl stehenden Schablonen, die keine explizit festgelegte Priorität besitzen, wird die Priorität aus dem Muster ermittelt. Sollte auch nach Auswertung der Prioritäten keine eindeutige Auswahl möglich sein, muss der XSLT-Prozessor entweder die Verarbeitung mit einer Fehlermeldung abbrechen oder die zuletzt innerhalb des/der Stylesheets definierte Schablone verwenden.

Bereits durch Anwendung einzelner Schablonen können beliebig komplexe Dokumente oder Teildokumente erzeugt werden. Es lassen sich Strukturen beliebiger Komplexität aufbauen, Zeichenketten in die Zieldokumente einfügen oder anhand im Quelldokument auftretender Elemente wiederholt Abbildungen in das Zieldokument einfügen. Einfache Transformationen können in der Regel durch Anwendung einer einzigen Schablone durchgeführt werden. Dies gilt insbesondere für Transformationen, die die vollständige Struktur des Zieldokumentes unabhängig von der Struktur des Quelldokumentes ermitteln.

Jede Instantiierung einer Schablone wird relativ zu einem aktuellen Knoten und einer aktuellen Knotenmenge ausgeführt. Der *aktuelle* Knoten ist dabei immer ein Mitglied der aktuellen Knotenmenge. Die meisten Operationen in XSLT werden relativ zum aktuellen Knoten oder zur aktuellen Knotenmenge angewendet. Nur wenige Operationen verändern den aktuellen Knoten während ihrer Ausführung. Ein Beispiel für eine solche Operation ist die **for-each**-Schleife, die in jedem Schleifendurchlauf einen neuen aktuellen Knoten bestimmt. Nach Abarbeitung der Schleife ist der aktuelle Knoten wiederum derselbe wie vor der Schleife.

Die XSLT-Spezifikation beschreibt ausschließlich den Sprachumfang von XSLT. Der Mechanismus, der zur Anwendung der Stylesheets auf ein XML-Dokument von einem XSLT-Prozessor eingesetzt wird, ist dagegen unspezifiziert und bleibt den Implementierungen überlassen. Es wird allerdings ein Mechanismus empfohlen [Cla99a] und festgelegt, dass die simultane Anwendung mehrerer Stylesheets auf ein XML-Dokument dieselbe Wirkung haben muss wie die Anwendung eines Stylesheets, das diese importiert.

Abb. 9.8 zeigt ein Stylesheet, das eine JMI-Schnittstelle für eine MOF-Klasse erzeugt. Die Wirkung des Stylesheets entspricht der Wirkung der Velocity-Schablone aus Abb. 9.7. Alle Anweisungen des Stylesheets beginnen mit einem vorangestellten **xsl**, weil der XSLT-Namensraum das Präfix **xsl** zugeordnet bekommen hat.

Die Anweisungen des Stylesheets beginnen mit einer Reihe von **import**-Anweisungen, die einige in anderen Stylesheets enthaltene Schablonen bekannt machen, so dass diese aufgerufen werden können. Analog zu der Velocity-Schablone sind die Schablonen zum Erzeugen vollständiger Namen sowie zum Erzeugen der Methoden in eigene Schablonen ausgelagert, die innerhalb dieses Stylesheets durch die Anweisung **call-template** aufgerufen werden. Im Unterschied zu Velocity wird hier nicht der in anderen Schablonen enthaltene Code eingefügt, sondern ein Funktionsaufruf ausgeführt.

9.3.1.3 Skriptsprachen

Auf den ersten Blick erscheint es vielversprechend, eine moderne Skriptsprache wie Perl [WCO00], Python [vRD03] oder Ruby [TH00] zu verwenden. Diese Sprachen sind weit verbreitet und werden für ein Vielzahl von Aufgaben eingesetzt. Perl ist eng verwandt mit C, Python mit C++ und Ruby mit Objective-C. Diese Verwandtschaften zeigen, dass die aktuellen Skriptsprachen in ihrem Funktionsumfang weit über das hinausgehen, was für die Gestaltung einer Schablone zur Codegenerierung benötigt wird.

Die Realisierung der MOmo-Generator-Architektur erfordert die Kommunikation zwischen den generischen Bestandteilen eines Generators und den Schablonen. Zum einen muss ein Kontext bereitgestellt werden, der die Informationen enthält, die von einer Schablone zur Erzeugung eines Artefakts benötigt werden. Zum anderen muss die Ausführung der Schablonen durch den Generator gesteuert werden, d. h. die Ausführung einer Schablone in einem bestimmten Kontext muss veranlasst und das Ergebnis in eine bestimmte Datei weitergeleitet werden.

Um Daten zwischen einer Java-Applikation und einem Skript austauschen zu können, muss die Skriptsprache über eine Java-Anbindung verfügen. Die Integration von Perl und Java ist nur eingeschränkt möglich, was vermutlich durch die sehr ähnlichen Hauptanwendungsgebiete begründet ist. Python ist dagegen in einer Java-Implementierung (Jython, [PR02]) erhältlich, die eine vollständige Integration der Sprache in eine Java-Applikation ermöglicht. Für Ruby existiert mit JRuby [JRY04] ebenfalls eine Java-Implementierung, so dass auch Ruby-Skripten uneingeschränkter Zugriff auf Java-Objekte gewährt werden kann.

Die Schablonen eines MOmo-Generators erzeugen die gewünschten Artefakte, die in Dateien gespeichert werden. Bei der Verwendung von Velocity oder XSLT geben die Schablonen alle Bestandteile einer Schablone aus, die nicht zur Steuerung der Verarbeitung benötigt wer-

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

  <xsl:import href="GetComposedName.xsl"/>
  <xsl:import href="GetCopyright.xsl"/>
  <xsl:import href="GetReference.xsl"/>
  <xsl:import href="GetPackageDefinition.xsl"/>
  <xsl:import href="GetFullName.xsl"/>

  <xsl:import href="Property.xsl"/>
  <xsl:import href="Operation.xsl"/>

  <xsl:template match="class">
    <xsl:variable name="classNode" select="current()"/>
    <xsl:variable name="ifaceName" ...

    import java.util.Collection;
    import java.util.List;

    public interface <xsl:value-of select="$ifaceName"/> extends

    <xsl:choose>
      <xsl:when test="count(superclasses/classref[@isParent='true'])=0">
        javax.jmi.reflect.RefObject
      </xsl:when>
      <xsl:otherwise>
        <xsl:for-each select="superclasses/classref[@isParent='true']">
          <xsl:variable name="superclassNode"
                        select="document(@href)/class"/>
          <xsl:call-template name="getFullName">
            <xsl:with-param name="elementNode" select="$superclassNode"/>
          </xsl:call-template>
          <xsl:if test="not (position()=last())">,</xsl:if>
        </xsl:for-each>
      </xsl:otherwise>
    </xsl:choose>

    {
    <xsl:for-each select="ownedAttributes/propertyref">
      <xsl:variable name="propNode" select="document(@href)/property"/>
      <xsl:call-template name="genProperty">
        <xsl:with-param name="propNode" select="$propNode"/>
        <xsl:with-param name="leafClassNode" select="$classNode"/>
      </xsl:call-template>
    </xsl:for-each>
    ...
    } // end of interface
  </xsl:template>

</xsl:stylesheet>
```

Abbildung 9.8: XSLT-Schablone zur Erzeugung einer JMI-Schnittstelle

den. Eine Schablone lässt sich daher leicht erstellen, indem ein Muster eines Artefakts um die dynamischen Anteile ergänzt wird. Werden dagegen Skriptsprachen verwendet, müssen Ein- und Ausgabeoperationen explizit angegeben werden. Dies entspricht der Verwendung von Ein- und Ausgabeoperationen in Java. Da Schablonen in der Regel eine Vielzahl von Ausgaben enthalten, werden die Schablonen entsprechend viele Aufrufe von Ein- und Ausgabeoperationen enthalten.

Zusammenfassend lässt sich sagen, dass Skriptsprachen zur Realisierung des Schablonenmechanismus' eines MOmo-Generators verwendet werden können. Allerdings kommen die meisten Vorteile, insbesondere der Funktionsumfang und die Möglichkeit, Schablonen objektorientiert programmieren zu können, kaum zur Geltung, während sich die Nachteile deutlich auswirken. Dies gilt allerdings in erster Linie für die relativ einfachen Abbildungen, die im Rahmen dieser Arbeit durchgeführt wurden. Es ist nicht auszuschließen, dass komplexe Aufgaben auf dem Gebiet der Codegenerierung, z. B. die Erzeugung von Code für eingebettete Systeme, besser mit Hilfe von Skriptsprachen anstelle von Schablonensprachen durchgeführt werden können. Der Grund dafür sind die vielfältigen Optimierungen, die zur Erzeugung von besonders kompaktem oder performantem Code notwendig sind.

9.3.1.4 Auswahl

Um einen Schablonenmechanismus für den MOmo-Baukasten auszuwählen, werden die in Tab. 9.3.1.4 dargestellten Kriterien untersucht und der direkten Programmierung mit Java gegenübergestellt. Das Kriterium *Funktionsumfang* bewertet den Umfang und die Mächtigkeit der bereitgestellten Mechanismen, die zur Definition der Schablonen genutzt werden können. *Komplexität* umfasst die Erlernbarkeit und Verständlichkeit der Sprachen und ist daher eng verbunden mit den Kriterien *Lesbarkeit* und *Erstellungsaufwand*. Das Kriterium *Implementierung* gibt die Verfügbarkeit von Implementierungen an. Die Verfügbarkeit mehrerer konkurrierender Implementierungen wird dabei als vorteilhaft angesehen, weil dadurch die kontinuierlichere Weiterentwicklung und Verbesserung eines Mechanismus' als wahrscheinlicher angesehen wird. Die *Integrierbarkeit* bewertet, wie gut die Schablonenmechanismen in Java-Anwendungen eingebettet werden können. Das Kriterium *Ein-/Ausgabe* gibt an, wieviel Aufwand für die Durchführung der Ein-/Ausgabeoperationen in einer Schablone notwendig ist. Schließlich wurde noch das Kriterium *Dokumentation* betrachtet, das Umfang und Qualität der vorhandenen Literatur bewertet.

Für die Entwicklung eines flexiblen Baukastens sind Funktionsumfang, Erstellungsaufwand, Ein-/Ausgabe und die Verfügbarkeit von Implementierungen die wichtigsten Kriterien. Den größten Funktionsumfang bieten Skriptsprachen wie Perl, Python oder Ruby bzw. die direkte Programmierung mit Java. Allerdings wird ein Großteil des Funktionsumfangs dieser Sprachen bei der Anwendung als Schablonensprache nicht benötigt, da die entsprechenden Funktionen nur zur Anwendungsentwicklung benötigt werden. Dies gilt insbesondere für Schnittstellen zu Fenstersystemen oder anderen Bibliotheken. Für die Verwendung in MOmo-Codegeneratoren ist der Funktionsumfang von Velocity und XSLT ausreichend, wobei XSLT einige Funktionen

	Velocity	XSLT	Skriptsprachen	Java
Funktionsumfang	-	o	+	+
Komplexität	+	o	-	-
Flexibilität	+	+	o	-
Lesbarkeit	+	-	o	o
Erstellungsaufwand	+	+	-	-
Implementierung	-	+	-/o	o
Integrierbarkeit	+	+	o/+	+
Ein-/Ausgabe	+	+	-	-
Dokumentation	o	+	+	+

Tabelle 9.1: Bewertung der Schablonenmechanismen

enthält, die in Velocity bereits Rückgriffe auf das Java-API erfordern. Dies umfasst insbesondere Funktionen zum Umgang mit Zeichenketten, die zur Codegenerierung häufig genutzt werden.

Die Unterstützung von Ein-/Ausgabeoperationen ist ein weiteres wichtiges Kriterium für die Auswahl eines Schablonenmechanismus. Die Skriptsprachen und Java erfordern explizite Aufrufe der für Ein-/Ausgabeoperationen zuständigen Funktionen, während die Schablonensprachen Velocity und XSLT das Ergebnis der Transformation automatisch ausgeben. Letzteres ist der gewünschte Fall, da ein großer Anteil der zu erzeugenden Artefakte statisch ist. Dieser Anteil kann in einer Schablone genau so angegeben werden, wie er im Artefakt erscheinen soll, während er in einem Skript durch den Aufruf der Ausgabeoperation gekapselt wird.

Aus diesem Grund sind Skriptsprachen und Java im Allgemeinen kein geeigneter Mechanismus zur Realisierung der MOmo-Generatoren. In Einzelfällen, in denen häufig Informationen aus einem gesamten Modell gesammelt und ausgewertet werden müssen, kann dies allerdings anders sein. Velocity und XSLT unterscheiden sich in vielerlei Hinsicht, sind aber beide zur Realisierung der Architektur eines MOmo-Generators geeignet. Die Vorteile von Velocity gegenüber XSLT sind die kompaktere Syntax und die enge Verzahnung mit der Programmiersprache Java, die zur Implementierung des MOmo-Baukastens verwendet wird. Demgegenüber stehen die Vorteile von XSLT, wie Standardisierung, mehrere verfügbare Implementierungen und größerer Funktionsumfang.

Ein weiterer Vorteil von XSLT sind die größeren Strukturierungsmöglichkeiten des verwendeten Kontextes. Der Kontext besteht bei Verwendung von XSLT aus XML-Dokumenten, die beliebig tief geschachtelte Strukturen abbilden können. Ein Velocity-Kontext enthält dagegen nur einfache Variablendefinitionen. Geschachtelte Strukturen werden in Velocity durch Rückgriffe auf Java abgebildet. In diesem Fall enthält eine Schablone neben den Velocity-Ausdrücken auch Java-Code.

Für die prototypische Realisierung des MOmo-Baukastens im Rahmen dieser Arbeit wurden die Vorteile von XSLT, insbesondere die Möglichkeit, strukturierte Kontexte darzustellen, und die große Anzahl verfügbarer Implementierungen als schwerwiegender angesehen als die Nachteile gegenüber anderen Ansätzen. Daher wurden die Komponenten des MOmo-Baukastens

XSLT-basiert erstellt. Es ist allerdings ohne Veränderung des grundlegenden Konzeptes möglich, jeden anderen der hier genannten Schablonenmechanismen zu verwenden. In den folgenden Abschnitten werden die XML- und XSLT-basierten Realisierungen des Kontextgenerators und des Schablonenanwenders vorgestellt.

9.3.2 Kontextgenerator

Der Kontextgenerator erzeugt eine Menge von Kontexten, die einzelne Elemente eines Modells repräsentieren. Dazu setzt der Generator auf einer MOmo-Implementierung eines MOF-Modells auf, so dass alle Modelle transformiert werden können, die sich durch MOF-Elemente beschreiben lassen. Dies schließt insbesondere die UML und alle UML-Profile ein, da diese durch Metamodelle definiert werden, die in MOF modelliert sind. In der Realisierung des MOmo-Baukastens sind die Kontexte als XML-Dokumente implementiert.

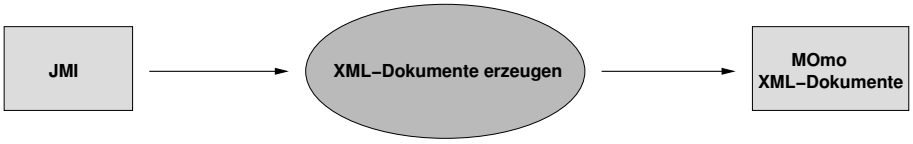


Abbildung 9.9: Die Aufgabe des XML-Generators

Im folgenden Abschnitt wird der Aufbau der XML-Dokumente beschrieben. Anschließend werden die Arbeitsschritte dargestellt, die zur Erzeugung der XML-Dokumente durchgeführt werden müssen.

9.3.2.1 Aufbau der XML-Dokumente

Der Kontextgenerator ist generisch implementiert, um unverändert für alle MOF-bassierten Modellierungssprachen verwendbar zu sein. Nach der Definition der Syntax einer Modellierungssprache durch ein MOF-Modell kann der Kontextgenerator die Kontexte für alle Modelle erzeugen, die durch die Modellierungssprache beschrieben werden. Daraus folgt, dass das Format der XML-Dokumente ebenfalls generisch sein muss, um alle Modelle abbilden zu können, deren Meta-Metamodell die MOF ist. Ein standardisiertes Dokumentformat, das diese Anforderung bereits erfüllt, ist XMI (siehe Kapitel 7).

XMI hat allerdings mehrere Eigenschaften, die die Eignung als Basis für die Anwendung von XSLT-Stylesheets deutlich einschränken. XMI-Dokumente können redundante Informationen enthalten, die durch verschiedenartige XML-Elemente dargestellt werden. Dies müsste bei der Implementierung von XSLT-Stylesheets beachtet werden, was die Komplexität der Stylesheets deutlich erhöhen würde. Zudem ist XMI vielfältig konfigurierbar. Die Konfiguration müsste ebenfalls in den Stylesheets ausgewertet werden, was die Komplexität der Stylesheets ebenfalls erhöhen würde. XMI ist zudem hauptsächlich zur Repräsentation vollständiger Modelle konzipiert worden. Im hier dargestellten Anwendungsfall ist aber eine Darstellungsform für einzelne

Modellelemente geeigneter, weil die Mehrzahl der Artefakte in der Regel für einzelne Modellelemente erzeugt werden muss.

Aufgrund der genannten Nachteile des XMI-Formates bei der Verwendung als Basis für die Anwendung von XSLT-Stylesheets wurde für den MOmo-Baukasten ein spezielles XML-Dokumentformat entworfen. Dieses Format ist wie XMI für alle MOF-basierten Modellierungssprachen einsetzbar. Im Unterschied zu XMI werden einzelne Modellelemente durch XML-Dokumente repräsentiert und ermöglicht so eine möglichst einfache Auswertung der enthaltenen Informationen. Zudem steigt die Skalierbarkeit gegenüber der direkten Verwendung der XMI-Eingabedokumente an, da nur die jeweils lokal benötigten Informationen verarbeitet werden müssen. Folgende Regeln beschreiben die Erzeugung entsprechender XML-Dokumente:

- Jedes eindeutig identifizierbare Java-Objekt wird auf ein XML-Dokument abgebildet. Der Wurzelknoten des Dokumentes enthält den lokalen Namen des Objektes und den Namensraum, der das Objekt umgibt.
- Alle Attribute eines Objektes werden durch Knoten des Dokumentes dargestellt, die unterhalb des Wurzelknotens angeordnet sind.
- Eigenschaften des Modells, deren Typ ein einfacher Datentyp ist, werden auf sog. Wertknoten abgebildet, die den Namen der Eigenschaft tragen. Wertknoten besitzen lediglich ein Attribut, das mit dem aktuellen Wert belegt wird.
- Objektreferenzen des Modells werden durch Verweise auf die entsprechenden Dokumente aufgelöst und durch Referenzknoten repräsentiert. Referenzknoten tragen den Namen der Referenz und verfügen über ein einziges Attribut, das mit der Referenz belegt wird. Zur Erzeugung von Referenzen lassen sich entweder die Typen der Objekte kombiniert mit den voll qualifizierten Namen verwenden oder eindeutige Bezeichner vergeben. Die erste Variante erleichtert die Fehlersuche bei der Entwicklung, die zweite ist effizienter bei der Verwendung eines Codegenerators.

Abb. 9.10 zeigt die Abbildung einer MOF-Klasse auf drei XML-Dokumente, die der Kontextgenerator erzeugt. Die linke Seite der Abb. zeigt die Klasse **NamedElement**, die ein Attribut **name** und eine Operation **getName** enthält. Der Typ des Attributes und der Typ des Rückgabewertes der Operation sind **String**. Auf der nächsthöheren Ebene der OMG-Metadatenarchitektur (siehe Kapitel 5) wird das dargestellte Modell durch Objekte für die Klasse, das Attribut und die Operation repräsentiert. Dies ist auch die Ebene der Zwischendarstellung, so dass drei XML-Dokumente erzeugt werden, die aufeinander verweisen. Diese Dokumente sind auf der rechten Seite der Abbildung dargestellt.

9.3.2.2 Arbeitsweise

Der Kontextgenerator erzeugt die im vorigen Abschnitt beschriebene Repräsentation eines MOF-Modells. Diese Repräsentation wird im nachfolgenden Arbeitsschritt, der durch den Codegenerator ausgeführt wird, zur Erzeugung der gewünschten Artefakte verwendet. Die Dokumente werden durch den in Abb. 9.11 dargestellten Ablauf erstellt.

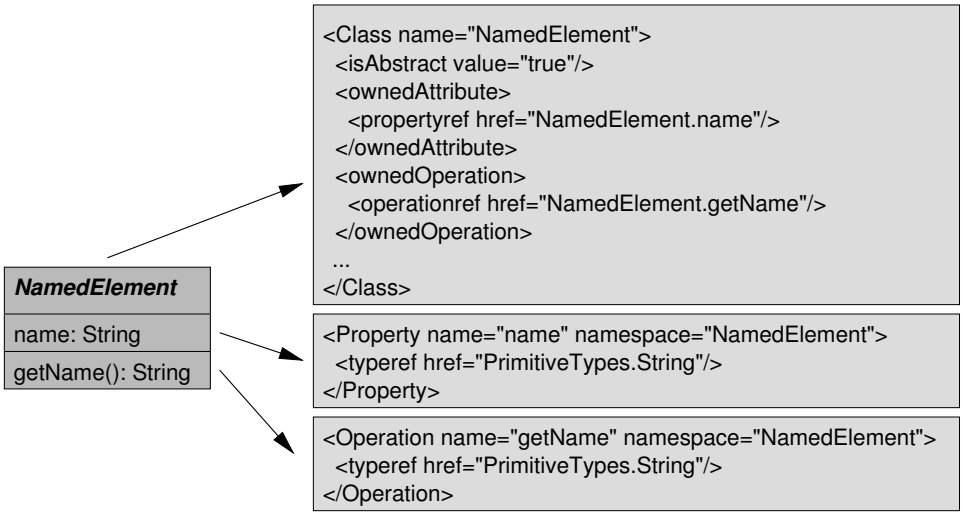


Abbildung 9.10: Kontexteerzeugung für eine JMI-Klasse

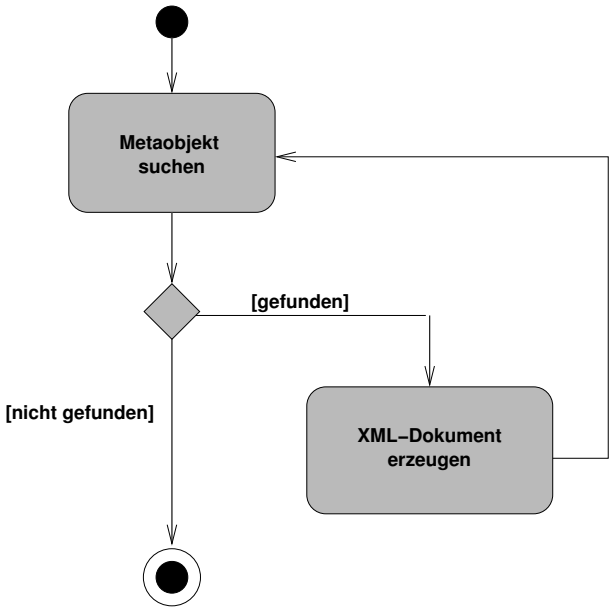


Abbildung 9.11: Erzeugung der XML-Repräsentation

Im ersten Verarbeitungsschritt sucht der Kontextgenerator ein Objekt, das er transformieren kann. Entsprechende Objekte müssen eine Schnittstelle implementieren, die den Zugriff auf die Bestandteile des Objektes ermöglicht. In der vorliegenden Implementierung wird dazu die Schnittstelle **RefObject** aus der JMI-Reflexions-Schnittstelle verwendet. Ein Objekt, das diese Schnittstelle implementiert, besitzt eine Verbindung zu der Klasse, die zu seiner Erzeugung instanziiert wurde. Durch Abfragen dieser Klasse können die Eigenschaften des Objektes ermittelt werden. Die reflexive Schnittstelle ermöglicht es, auf die Eigenschaften eines Objektes zuzugreifen.

Abb. 9.10 zeigt Beispiele für die Kontexte, die aus Metaobjekten erzeugt werden. Ein Beispiel für die Erzeugung eines Wertknotens ist die Übersetzung des Attributes **isAbstract**. Dieses Attribut ist auf den Wert **true** gesetzt, um zu kennzeichnen, dass die Klasse **NamedElement** abstrakt ist. Der entsprechende Wertknoten ist **`<isAbstract value="true">`**. Beispiele für Referenzknoten sind die **`propertyref`**- und **`operationref`**-Knoten in Abb. 9.10. Der Name eines Referenzknotens setzt sich immer aus dem Namen des Elementes mit nachgestelltem **`ref`** zusammen.

Die Ausgabe des Kontextgenerators ist eine Menge von XML-Dokumenten, die über Referenzen miteinander verbunden sind und dasselbe Modell darstellen wie das JMI-Modell, das als Eingabe dient. Alle weiteren Aufgaben der Codeerzeugung sind modellunabhängig und werden durch den Schablonenausführer durchgeführt, der in Abschnitt 9.3.3 beschrieben wird.

9.3.3 Schablonenausführer

Der Schablonenausführer ist die zweite Komponente, die alle MOmo-Generatoren gemeinsam haben. Der MOmo-Baukasten enthält eine generische Implementierung, die eine allgemeine Abbildung von Eingabedokumenten auf Artefakte durchführt, indem XSLT-Stylesheets auf die XML-Dokumente, die der Kontextgenerator für ein Modell erzeugt hat, angewendet werden. In Abb. 9.12 ist die Wirkung des Schablonenausführers zusammenfassend dargestellt.

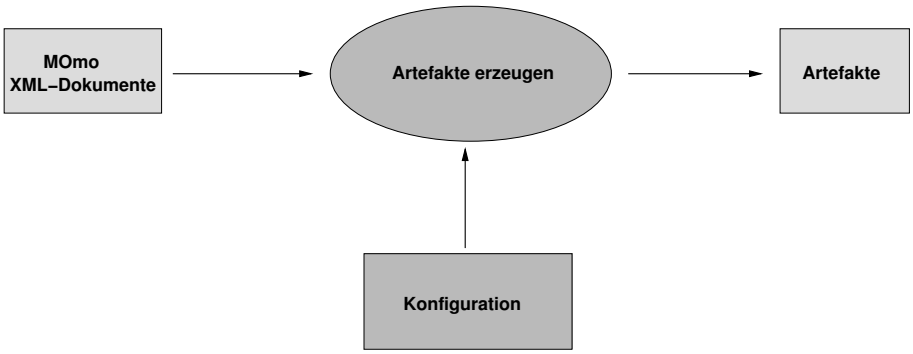


Abbildung 9.12: Der MOmo-Schablonenausführer

Die Steuerung des Abbildungsvorganges erfolgt durch eine Konfiguration. Die Konfiguration definiert die Ziele der Abbildung, indem bestimmten Eingabedokumenttypen bestimmte Stylesheets zugeordnet werden. Der Schablonenausführer wendet die Stylesheets auf die entsprechenden Dokumente an und erzeugt auf diese Weise die Artefakte.

Die Konfiguration eines MOmo-Generators legt fest, welche Artefakte für ein Modellelement erzeugt werden. Da jedes Modellelement in der XML-Darstellung durch ein XML-Dokument repräsentiert wird, müssen die für die Erzeugung eines Artefaktes zuständigen Stylesheets jeweils auf einen bestimmten Dokumenttyp angewendet werden. Ein Dokumenttyp repräsentiert immer einen Modellelementtyp, also z. B. eine Klasse, ein Paket oder einen strukturierten Datentyp. Für sehr komplexe Modellierungssprachen kann die Anzahl verschiedener Modellelementtypen sehr groß werden.

```
momoc.artefact.package.impl=stylesheets/jmi/PackageInterfaceImpl.xsl
momoc.artefact.package.impl.directory=impl
momoc.artefact.package.impl.prefix=Mof
momoc.artefact.package.impl.suffix=PackageImpl
momoc.artefact.package.impl.extension=java
```

Abbildung 9.13: Beispiel für einen Konfigurationsblock des Schablonenausführers

Der Konfigurationsblock für die Erzeugung eines Artefakts besteht aus einer Menge von Konfigurationsvariablen, die vom Schablonenausführer ausgewertet werden. Jede Konfigurationsvariable beginnt mit **momoc.artefact**, um zu kennzeichnen, dass diese Variable im Transformationsschritt verwendet wird. Auf **momoc.artefact** folgt ein individueller Bestandteil des Variablennamens, der in jeder Konfiguration jeweils nur einmal vorkommen darf. Dieser Bestandteil des Namens legt den „Namen“ des Artefaktes fest.

Konfigurationsblöcke können sowohl die Variablen enthalten, die zur Steuerung der Transformation benötigt werden, als auch Variablen, die erst während der Erzeugung eines Artefaktes ausgewertet werden. Die Variablen zur Steuerung der Transformation legen den Namen oder Teile des Namens der Ausgabedateien fest. Entweder werden die Variablen **momoc.artefact.*typ*.*artefakt*.prefix** und **momoc.artefact.*typ*.*artefakt*.suffix** verwendet, oder es wird die Variable **momoc.artefact.*typ*.*artefakt*.name** definiert. Die Belegungen der beiden erstgenannten Variablen werden mit dem Namen des Artefaktes zum Namen der Ausgabedatei verbunden. Der Name des Artefaktes wird aus dem Wurzelknoten des XML-Dokumentes entnommen, das das Artefakt repräsentiert. Entsprechend den Variablennamen werden entweder die Werte der beiden Variablen vor und hinter dem Namen angefügt und ergeben den vollständigen Artefaktnamen oder legen den Namen des Artefaktes direkt fest. Zusätzlich kann über die Variable **momoc.artefact.*typ*.*artefakt*.extension** eine Erweiterung des Namens festgelegt werden, z. B. *java* für die Bestandteile einer JMI-Implementierung.

Die weiteren Bestandteile eines Konfigurationsblockes für ein Artefakt sind abhängig von den Stylesheets, die zur Erzeugung des Artefaktes angewendet werden. Jede Variable, die in den Stylesheets verwendet werden soll, muss in der Konfiguration definiert werden. Die Auswertung dieser Variablen erfolgt während der Anwendung eines Stylesheets. Damit Konfigurationsvariablen von Stylesheets ausgewertet werden können, stellt der MOmo-Baukasten Zugriffsoperationen zur Verfügung, die aus XSLT-Stylesheets problemlos aufgerufen werden können.

Für jeden Konfigurationsblock wird vom Schablonenausführer das angegebene Stylesheet eingelesen und auf jedes „passende“ Dokument angewendet. Die Auswahl der passenden Dokumente erfolgt mit Hilfe der Typangabe innerhalb der Variablen. Der aktuelle Name des Typs muss mit dem lokalen Namen des Wurzelknotens eines Dokumentes übereinstimmen, damit das Dokument als passend angesehen wird. Die Aktionen für einen Konfigurationsblock, dessen Variablennamen mit **momoc.artefact.package** beginnen, werden folglich auf alle Dokumente angewendet, deren Wurzelknoten ein **<package>**-Knoten ist. In Abb. 9.13 ist ein Beispiel für einen Konfigurationsblock dargestellt.

Abb. 9.13 zeigt alle Konfigurationsvariablen, die zur Erzeugung der Implementierung einer Paketschnittstelle benötigt werden. Neben den bereits genannten Variablen **prefix**, **suffix** und **extension** enthält die dargestellte Konfiguration zusätzlich die Variable **directory**. Durch Setzen dieser Variable kann ein Verzeichnis unterhalb des Wurzelverzeichnises der Implementierung angegeben werden. Wenn beispielsweise eine JMI-Implementierung für das MOF-Metamodell unterhalb des Wurzelverzeichnisses *de/unibwm/ist* erzeugt wird, werden alle Implementierungen von Paketinstanzen jeweils im Verzeichnis *de/unibw/ist/impl/*<*container*> angeordnet. <*container*> ist das Verzeichnis, das das direkt übergeordnete Modellelement enthält.

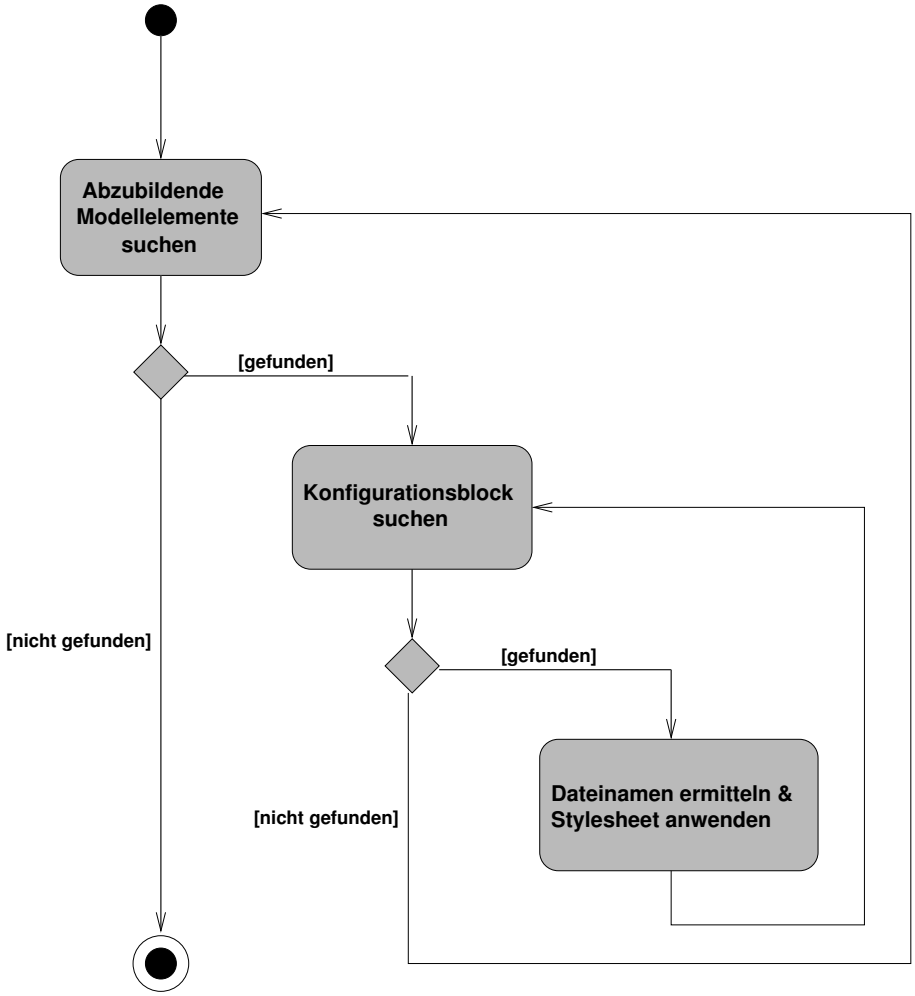


Abbildung 9.14: Erzeugung der Artefakte

In Abb. 9.14 ist die Arbeitsweise des Schablonenausführers zusammenfassend dargestellt. Es wird deutlich, dass der Schablonenausführer keinerlei Informationen über die Modellierungssprache benötigt, die gerade übersetzt wird. Diese Informationen sind ausschließlich in der Konfiguration und den Stylesheets enthalten, so dass sich die Aufgabe der Entwickler beim Einsatz des MOmo-Baukastens ausschließlich auf die Definition der Abbildung der Modellierungs- auf die Zielsprache konzentrieren kann.

Der aufwändige Teil der Implementierung eines Codegenerators mit Hilfe des MOmo-Baukastens ist das Erstellen der Stylesheets, die die XML-Dokumente der Zwischendarstellung in die Artefakte der Zieldarstellung übersetzen. Ein Hauptproblem dabei ist der mangelnde Sprachumfang von XSLT 1.0 in Bezug auf die Manipulation von Zeichenketten. Beispielsweise müssen zur Erzeugung einer JMI-konformen Schnittstelle für ein MOF-Modell häufig einzelne Buchstaben eines Namens explizit groß geschrieben werden, um die Namenskonventionen der JMI-Spezifikation einzuhalten.

Die Version 2.0 der XSLT-Spezifikation wird die notwendigen Funktionen zum Umgang mit Zeichenketten enthalten. Da XSLT 2.0 aber noch nicht verabschiedet ist, und mit einer Ausnahme auch noch keine entsprechenden Werkzeuge verfügbar sind, baut der MOmo-Baukasten auf der Version 1.0 des XSLT-Standards auf. Die fehlenden Funktionen müssen durch Rückgriffe auf das Java-API realisiert werden. Leider sind externe Funktionsaufrufe in XSLT 1.0 nicht standardisiert, so dass sich Syntax und Funktionsumfang der verschiedenen XSLT-Prozessoren unterscheiden. Innerhalb des MOmo-Baukastens werden daher die wichtigsten Textmanipulationsoperationen in einer Klasse definiert und durch vordefinierte Stylesheets gekapselt.

9.4 Zusammenfassung

In diesem Kapitel wurde der Aufbau der Schablonenablaufumgebung eines MOmo-Codegenerators beschrieben. Jeder Codegenerator enthält zwei Komponenten, die die Erzeugung der durch eine Konfiguration vorgegebenen Artefakte aus einer Repräsentation eines Modells durch Metaobjekte übernehmen. Aufgrund der geforderten Flexibilität wird die Erzeugung der Artefakte mit Hilfe von Schablonen durchgeführt, die auf Kontexte angewendet werden. Diese Aufteilung wirkt sich unmittelbar auf die in einem Generator enthaltenen Komponenten aus. Eine Komponente stellt die Kontexte bereit, die die zur Codeerzeugung notwendigen Informationen enthalten. Eine zweite Komponente wendet die Schablonen auf die Kontexte an und erzeugt so die gewünschten Artefakte.

Die beiden genannten Komponenten stellen eine Ablaufumgebung für Schablonen bereit. Ihre Implementierung hängt daher maßgeblich vom gewählten Schablonenmechanismus ab. Für die Implementierung innerhalb des MOmo-Baukastens wurde XSLT ausgewählt, um die schablonenbasierte Codeerzeugung zu realisieren. Die maßgeblichen Gründe für diese Auswahl sind:

- die Möglichkeit, Kontexte analog zu einem vorliegenden Modell zu strukturieren
- die Vielzahl und Qualität verfügbarer Implementierungen
- die Standardisierung, die eine kontinuierliche Weiterentwicklung der Sprache verspricht

Die konkrete Durchführung der Codeerzeugung beginnt mit einem Modell, das durch die in Kapitel 8 beschriebenen Schnittstellen angesprochen werden kann. Zunächst werden XML-Dokumente für die einzelnen Bestandteile des Modells erzeugt. Die XML-Dokumente sind

durch Referenzen miteinander verbunden und enthalten dieselben Informationen wie die JMI-konformen Objekte. Anschließend werden die XML-Dokumente durch XSLT-Stylesheets in die gewünschten Artefakte transformiert.

Welche Stylesheets auf welche Dokumente anzuwenden sind, wird durch die Konfiguration festgelegt. Die Konfiguration besteht aus einzelnen Blöcken, die jeweils eine Zuordnung eines Wurzelknotentyps zu einem Stylesheet enthalten. Außerdem enthalten Konfigurationsblöcke einen bestimmten Satz vordefinierter Variablen, die vom XML-Transformator ausgelesen werden, um Namen für die zu erzeugenden Artefakte festzulegen. Des Weiteren können beliebig viele Variablen angegeben werden, um Vorgänge innerhalb der Stylesheets zu beeinflussen.

Kapitel 10

Anwendungen

In den vorhergehenden Kapiteln wurden die Komponenten des MOmo-Baukastens dargestellt, die zur Entwicklung von Codegeneratoren für MOF-basierte Modellierungssprachen verwendet werden können. In diesem Kapitel soll zum einen die Anwendung des MOmo-Baukastens demonstriert und zum anderen das Laufzeitverhalten von MOmo-Codegeneratoren in Abhängigkeit vom zu übersetzenden Modell beschrieben werden.

Abschnitt 10.1 beschreibt die Anwendung des MOmo-Baukastens zur Implementierung eines Codegenerators für eine einfache Petri-Netz-Sprache sowie den Einsatz des resultierenden Codegenerators, um die Funktionsweise des MOmo-Baukastens zu verdeutlichen. Abschnitt 10.2 gibt einen Überblick über den Aufbau des MOmo-MOF-Codegenerators sowie dessen Laufzeitverhalten bei der Erzeugung von Metamodellimplementierungen für verschiedene MOF- und UML-Versionen. Anhand der Modellierungssprache MOF wird gezeigt, dass sich der MOmo-Baukasten zur Implementierung von Codegeneratoren für praktisch einsetzbare Sprachen eignet.

10.1 Petri-Netze

In diesem Abschnitt wird die Anwendung des MOmo-Baukastens an einem einfachen Beispiel erläutert. Es soll eine einfache Steuerung für eine Waschmaschine modelliert und implementiert werden. Zur Modellierung der Steuerung werden Petri-Netze verwendet. Die Umsetzung der Petri-Netze in Quelltexte der Programmiersprache Java erfolgt durch einen MOmo-Codegenerator, der für die Petri-Netz-Sprache entwickelt wird. In Abschnitt 10.1.1 werden die Elemente der hier verwendeten Petri-Netze vorgestellt. Abschnitt 10.1.2 beschreibt die Entwicklung eines MOmo-Codegenerators für Petri-Netze, der in Abschnitt 10.1.3 verwendet wird, um die Waschmaschinensteuerung zu implementieren.

10.1.1 Definition

Petri-Netze sind eine Sprache zur Beschreibung nebenläufiger Systeme. Sie wurden von C. A. Petri in seiner Dissertation [Pet62] eingeführt und werden in vielen verschiedenen Anwendungsbereichen verwendet. Aufgrund der vielfältigen Einsatzmöglichkeiten haben Petri-Netze ein breites Interesse hervorgerufen. Da für manche Anwendungsbereiche zusätzliche Beschreibungsmittel benötigt wurden, entstanden entsprechende Erweiterungen, wie z. B. *Zeit-Petri-Netze* [PZ87] zur Spezifikation des Zeitverhaltens eines Netzes und *Gefärbte Petri-Netze* [Jen97], die eine Typisierung der verarbeiteten Daten ermöglichen.

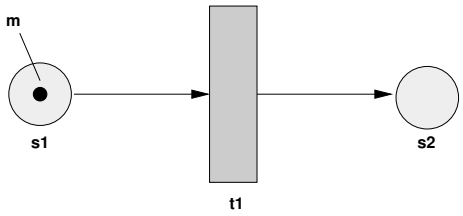


Abbildung 10.1: Elemente eines Stellen-Transitionen-Netzes

Die bekannteste Form der Petri-Netze sind die sog. *Stellen-Transitionen-Netze*. Ein Stellen-Transitionen-Netz ist ein Graph, dessen Knoten entweder eine *Stelle* oder eine *Transition* darstellen. In Abb. 10.1 sind die Stellen **s1** und **s2** als Kreise und die Transition **t1** als Rechteck dargestellt. Stellen und Transitionen werden durch die *Flussrelation* verbunden. Innerhalb des Graphs wird die Flussrelation durch gerichtete Kanten repräsentiert, die entweder von einer Stelle zu einer Transition oder von einer Transition zu einer Stelle führen. Die Flussrelation wird in Abb. 10.1 durch die Pfeile von der Stelle **s1** zur Transition **t1** und von der Transition **t1** zur Stelle **s2** repräsentiert.

Die Menge aller Stellen, die mit einer Transition über Kanten verbunden sind, wird in zwei Teilmengen unterschieden. Der *Vorbereich* ist die Menge der Stellen, die durch Kanten von einer Stelle zur Transition mit der Transition verbunden sind. Der *Nachbereich* enthält dagegen alle Stellen, die durch Kanten von der Transition zur Stelle mit der Transition verbunden sind. Der Vorbereich der Transition **t1** aus Abb. 10.1 enthält also die Stelle **s1**, während der Nachbereich derselben Transition die Stelle **s2** umfasst.

Neben Stellen und Transitionen enthält ein Stellen-Transitionen-Netz sog. *Marken*. Abb. 10.1 zeigt die Marke **m**, die sich in der Stelle **s1** befindet. Marken können als Aufenthalt an dieser Stelle aufgefasst werden. Dies ist die wesentliche Erweiterung der Stellen-Transitionen-Netze gegenüber den Automaten, da sich ein System parallel in mehreren oder auch mehrfach in derselben Stelle befinden kann. In letztgenanntem Fall muss eine Stelle in der Lage sein, mehrere Marken gleichzeitig aufzunehmen. Die Zuordnung einer Menge von Marken zu den Stellen eines Stellen-Transitionen-Netzes wird als *Markierung* des Netzes bezeichnet und kennzeichnet einen bestimmten Zustand des abgebildeten Systems. Der Ablauf eines Systems kann als Folge von Markierungen angesehen werden. Eine Markierung kann durch Schalten einer Transition in eine andere Markierung überführt werden.

Die *Schaltregel* legt fest, wann eine Transition schalten darf. Das *Schalten* einer Transition kann als *Verbrauchen von Marken* im Vorbereich der Transition und *Erzeugen von Marken* im Nachbereich der Transition aufgefasst werden. Verbrauchen und Erzeugen bedeutet, dass die Anzahl der Marken in einer Stelle erniedrigt bzw. erhöht wird. Die Anzahl der Marken, die durch Verbrauchen aus einer Stelle entfernt bzw. durch Erzeugen einer Stelle hinzugefügt werden, wird durch das sog. *Gewicht* der Kante bestimmt, die Stelle und Transition verbindet. Eine Kante des Gewichtes *g* verbraucht bzw. erzeugt *g* Marken, wenn die Transition schaltet. Dementsprechend setzt das Schalten einer Transition voraus, dass jede Stelle des Vorbereichs die entsprechende Anzahl von Marken enthält, und jede Stelle des Nachbereichs die entsprechende Anzahl Marken aufnehmen kann. Wird für eine Kante kein Gewicht angegeben, entspricht dies dem Gewicht Eins. Schaltet also die Transition **t1** in Abb. 10.1, wird die Marke **m** aus der Stelle **s1** entfernt und eine Marke in die Stelle **s2** eingefügt.

10.1.2 MOmo-Codegenerator

Ein MOmo-Codegenerator wird in drei Schritten entwickelt. Im ersten Schritt wird ein MOF-Modell der zu übersetzenden Modellierungssprache erstellt. Der zweite Schritt ist die Erzeugung einer Implementierung des MOF-Modells durch den MOmo-MOF-Codegenerator. Im dritten Schritt muss eine Abbildung der Elemente des MOF-Modells auf Quelltexte der Zielsprache definiert und durch Schablonen implementiert werden. Das Beispiel soll in der Programmiersprache Java realisiert werden, so dass eine Abbildung auf Java-Quelltexte erfolgen muss. Zusammengesetzt ergeben die generierte Metamodellimplementierung, die Komponenten der generischen Schablonenablaufumgebung aus dem MOmo-Baukasten und die Schablonen einen MOmo-Codegenerator.

10.1.2.1 Definition des Metamodells

Um einen MOmo-Codegenerator zu erstellen, wird ein MOF-Modell benötigt, das die Elemente der zu übersetzenden Modellierungssprache festlegt. In diesem Beispiel müssen die Bestandteile eines Stellen-Transitionen-Netzes durch MOF-Elemente modelliert werden.

Abb. 10.2 zeigt ein MOF-Modell, das eine einfache Stellen-Transitionen-Netz-basierte Modellierungssprache definiert. Die Klassen **Kante** und **Knoten** definieren die grundlegenden Bestandteile eines Graphen. Durch eine Assoziation zwischen den beiden Klassen wird festgelegt, dass jede Kante genau zwei Knoten miteinander verbindet, und dass jeder Knoten Anfangs- oder Endpunkt beliebig vieler Kanten sein darf. Die Ordnung auf dem Assoziationsende **knoten** wird verwendet, um den Eingangs- vom Ausgangsknoten zu unterscheiden.

Um auszudrücken, dass jede Kante in einem Stellen-Transitionen-Netz genau eine Stelle mit genau einer Transition verbindet, wird das Assoziationsende **knoten** der Assoziation zwischen den Klassen **Kante** und **Knoten** als **union** definiert. Damit ist dieses Assoziationsende auch schreibgeschützt, d. h. die Assoziation kann nicht instanziiert werden. Stattdessen muss eine Assoziation instanziiert werden, die ein Assoziationsende enthält, das in einer Teilmengenbeziehung zum Assoziationsende **knoten** steht.

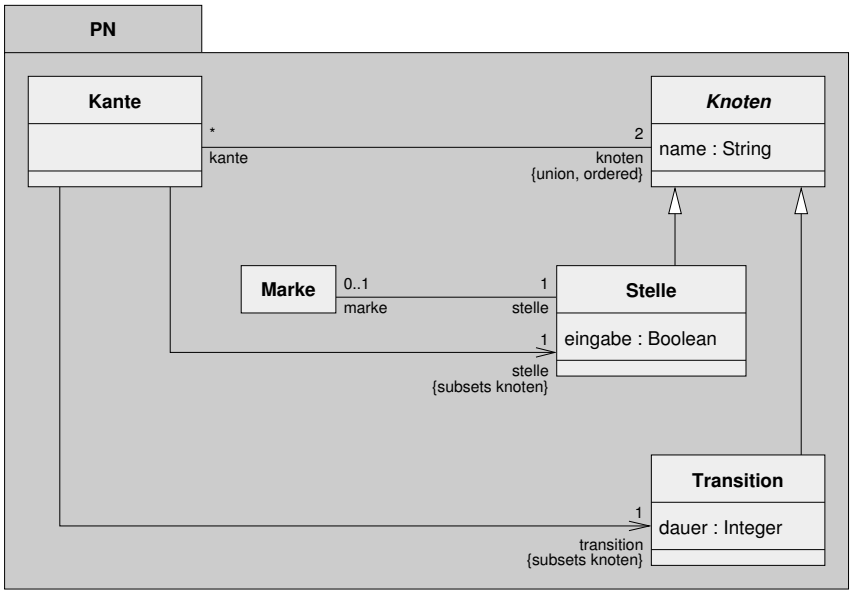


Abbildung 10.2: Definition von Stellen-Transitionen-Netzen

In Abb. 10.2 sind Assoziationen von der Klasse **Kante** zu den Klassen **Stelle** und **Transition** dargestellt, die jeweils ein Assoziationsende enthalten, das als Teilmenge des Assoziationsendes **knoten** definiert ist. Beide Teilmengen haben jeweils eine Multiplizität von Eins, so dass jeder Kante genau eine Stelle und eine Transition zugeordnet werden müssen. Das Gewicht der Kante wird immer als Eins angenommen und daher nicht modelliert.

Wie oben beschrieben, können Stellen sog. Marken enthalten. Marken werden durch Instanzen der Klasse **Marke** repräsentiert. Durch eine Assoziation zwischen den Klassen **Marke** und **Stelle** wird ausgedrückt, dass jede Marke sich in genau einer Stelle befindet, und jede Stelle maximal eine Marke enthalten darf. Des Weiteren enthält die Klasse **Stelle** das Boolesche Attribut **eingabe**. Mit Hilfe dieses Attributs können Stellen gekennzeichnet werden, die mit externen Quellen, z. B. Sensoren, verbunden werden sollen. Durch solche Verbindungen können Implementierungen der modellierten Steuerungen mit den zu steuernden Anwendungen oder Systemen gekoppelt werden.

Um die Steuerungen für Maschinen modellieren zu können, muss die Modellierungssprache die Modellierung des zeitlichen Verhaltens unterstützen. In der hier dargestellten Sprache können Zeitdauern für den Schaltvorgang einer Transition angegeben werden. Die Klasse **Transition** enthält das Attribut **dauer**, das die Spezifikation der Zeitdauer in Sekunden erlaubt.

10.1.2.2 Implementierung des Metamodells

Der im MOmo-Baukasten enthaltene MOF-Codegenerator verlangt eine XMI-Darstellung des MOF-Modells, das den Sprachumfang der Modellierungssprache festlegt. Abb. 10.3 zeigt die XMI-Darstellung des Modells der Modellierungssprache PN, die im Folgenden zur Modellierung einer einfachen Waschmaschinensteuerung verwendet werden soll. Wie in Kapitel 7 be-

schrieben wurde, kann es verschiedene XMI-Repräsentationen desselben Modells geben. Die hier dargestellte Repräsentation entspricht der Voreinstellung, die der XMI-Standard festlegt.

Die XMI-Repräsentation wird vom MOmo-MOF-Codegenerator verarbeitet, um eine einfache Implementierung der Sprache zu erzeugen. Abb. 10.4 zeigt die Bestandteile der generierten Implementierung des MOF-Modells der hier verwendeten Modellierungssprache. Um die Übersichtlichkeit zu erhöhen, wird auf die Darstellung der generierten Attribute und Methoden verzichtet. Die wichtigsten Methoden werden aber im Folgenden erläutert.

Der MOmo-MOF-Codegenerator erzeugt eine Repräsentation des MOF-Modells durch Java-Schnittstellen und -Klassen. Das Paket **PN** wird durch die Schnittstelle **PNPackage** und die Klasse **PNPackageImpl** implementiert. Die Schnittstelle **PNPackage** definiert Methoden zum Zugriff auf die Modellierungselemente, die im Paket **PN** enthalten sind. Die Klasse **PNPackageImpl** stellt eine Implementierung der Schnittstelle **PNPackage** bereit.

Die konkreten Klassen **Kante**, **Stelle**, **Transition** und **Marke** werden durch jeweils zwei Schnittstellen und zwei Klassen implementiert. Die Schnittstellen ***Klassenname*Class** und die Klassen ***Klassenname*ClassImpl** repräsentieren die Klassen des Modells, während die Schnittstellen ***Klassenname*** und Klassen ***KlassennameImpl*** die Instanzen der Klasse darstellen. Die Repräsentationen der Klassen werden verwendet, um neue Instanzen erzeugen zu können.

Die oben genannte Schnittstelle **PNPackage** ermöglicht den Zugriff auf die Klassen, die die Schnittstellen ***Klassenname*Class** implementieren, und bildet daher den Ausgangspunkt für die Erzeugung der Repräsentationen von PN-Modellen. Die abstrakte Klasse **Knoten** wird nur durch eine gleichnamige Schnittstelle repräsentiert, so dass keine Instanzen dieser Klasse erzeugt werden können. Das in der Klasse **Knoten** enthaltene Attribut **name** wird in jede Implementierung einer abgeleiteten Klasse, d. h. **StelleImpl** und **TransitionImpl**, kopiert. Diese ist nur möglich, weil Vererbung auf Ebene der Implementierungen durch Kopieren simuliert wird. In anderen Ansätzen müsste zumindest die Klasse **KlasseImpl** erzeugt werden, um die Implementierungen der Attribute und Operationen einer abstrakten Klasse bereitzustellen.

Jede Assoziation wird ebenfalls durch eine Schnittstelle und eine Klasse, die die Schnittstelle implementiert, repräsentiert. Da die Assoziationen des Modells der Sprache **PN** keine Namen besitzen, werden stattdessen die Namen der beteiligten Klassen verwendet. Es wird also beispielsweise eine Schnittstelle **Kante-StelleAssociation** generiert, die die Assoziation zwischen diesen beiden Klassen implementiert. Alle Instanzen der Assoziationen werden durch die generierten Klassen verwaltet.

```
<xmi:XMI xmi:version="2.1"
         xmlns:xmi="http://www.omg.org/XMI"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:cmof="http://schema.omg.org/spec/mof/2.0/cmof.xmi">

  <cmof:Package xmi:id="001" name="PN">
    <ownedType xsi:type="cmof:Class" xmi:id="002" name="Kante">
      <ownedAttribute xmi:id='003' name="knoten" type="006"
         lower="2" upper="2" isDerivedUnion="true" isOrdered="true"
         opposite="008" association="014"/>
      <ownedAttribute xmi:id='004' name="stelle" type="006"
         lower="1" upper="1" subsettedPropery="003" association="015"/>
      <ownedAttribute xmi:id='005' name="transition" type="006"
         lower="1" upper="1" subsettedProperty="003" association="017"/>
    </ownedType>
    <ownedType xsi:type="cmof:Class" xmi:id="006" name="Knoten">
      <ownedAttribute xmi:id="007" name="name" type="String"
         lower="1" upper="1"/>
      <ownedAttribute xmi:id="008" name="kante" type="002"
         lower="1" upper="1" association="014"/>
    </ownedType>
    <ownedType xsi:type="cmof:Class" xmi:id="009" name="Marke">
      <ownedAttribute xmi:id="010" name="stelle"
         lower="1" upper="1" association="019"/>
    </ownedType>
    <ownedType xsi:type="cmof:Class" xmi:id="011" name="Stelle"
      superClass="006">
      <ownedAttribute xmi:id="012" name="marke"
         lower="0" upper="unbounded" association="019"/>
    </ownedType>
    <ownedType xsi:type="cmof:Class" xmi:id="013" name="Transition"
       superClass="006"/>
    <ownedType xsi:type="cmof:Association" xmi:id="014"
       memberEnd="003 008"/>
    <ownedType xsi:type="cmof:Association" xmi:id="015"
      memberEnd="004 016">
      <ownedEnd xmi:id="016" type="002" lower="0"/>
    </ownedType>
    <ownedType xsi:type="cmof:Association" xmi:id="017"
      memberEnd="005 018">
      <ownedEnd xmi:id="018" type="002" lower="0"/>
    </ownedType>
    <ownedType xsi:type="cmof:Association" xmi:id="019"
      memberEnd="010 012"/>
  </cmof:Package>
</xmi:XMI>
```

Abbildung 10.3: XMI-Darstellung des Metamodells aus Abb. 10.2

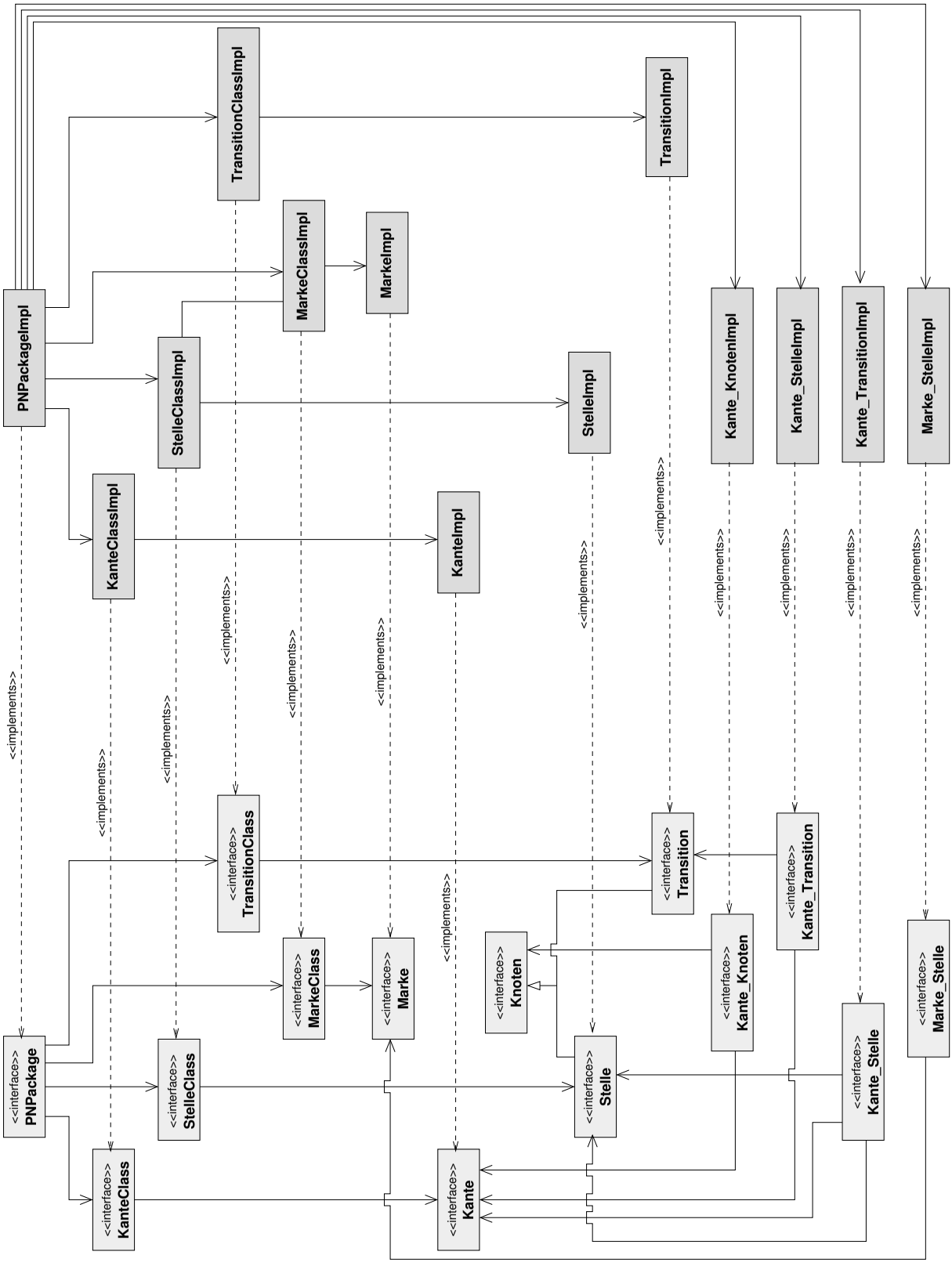


Abbildung 10.4: Generierte Implementierung des MOF-Modells

10.1.2.3 Definition einer Abbildung auf die Programmiersprache Java

Im dritten Schritt muss eine Abbildung der Elemente der Sprache **PN** festgelegt werden. Es soll hier eine Abbildung auf die Programmiersprache Java erfolgen. Jedes Netz besteht aus einer Menge von Stellen und Transitionen. Eine Stelle kann in dem durch Abb. 10.2 festgelegten Netztyp genau eine Marke speichern und besitzt einen Namen. Abb. 10.5 zeigt eine Java-Klasse, die diese Eigenschaften implementiert.

```
public class Place {
    public Place(String itsName) {
        this.itsName = itsName;
        this.hasToken = false;
        System.out.println("Stelle " + itsName + " erzeugt.");
    }

    public String getName() { return itsName; }
    public boolean hasToken() { return hasToken; }

    public void setToken(boolean token) {
        if (token)
            System.out.println("Marke in Stelle " + itsName + " erzeugt");
        else
            System.out.println("Marke in Stelle " + itsName + " verbraucht");
        this.hasToken = token;
    }

    private final String  itsName;
    private final boolean hasToken;
} // Place
```

Abbildung 10.5: Implementierung einer Stelle

Die Klasse **Place** besitzt zwei Attribute, um den Namen und die Marke zu speichern. Der Name einer Stelle kann bei der Erzeugung der Stelle gesetzt und durch die Methode **getName** abgefragt werden. Mit Hilfe der Methode **hasToken** kann abgefragt werden, ob die Stelle eine Marke enthält. Die Methode **setToken** kann aufgerufen werden, um eine Marke zu *verbrauchen* oder zu *erzeugen*. Zum Verbrauchen bzw. Erzeugen einer Marke wird die Methode **setToken** mit dem Booleschen Wert **false** bzw. **true** aufgerufen.

Der zweite wichtige Baustein von Stellen-Transitionen-Netzen sind die Transitionen. Diese werden durch Instanzen der Klasse **Transition** repräsentiert, deren Implementierung in Abb. 10.6 dargestellt ist.

```
import java.util.ArrayList;
import java.util.List;

public class Transition extends Thread {

    public Transition(String itsName, int itsDuraction) {
        this.itsName = itsName; this.itsDuration = itsDuration;
    }

    public void run() {
        try {
            while (!enabled()) {
                if (isReady) {
                    isReady = false; concurrent.setReady(true);
                }
                sleep(1);
            }
            consumeToken();
            sleep(itsDuration*1000);
            createToken();
            isReady = false;
            concurrent.setReady(true);
        } catch (InterruptedException ie) {
            System.err.println(ie.getMessage());
        }
    }

    public boolean enabled() {
        if (!isReady) return false;
        for (int i = 0; i < sources.size(); i++)
            if (!((Place)sources.get(i)).hasToken()) return false;
        for (int i = 0; i < targets.size(); i++)
            if (((Place)sources.get(i)).hasToken()) return false;
        return true;
    }

    public void consumeToken() {
        for (int i = 0; i < sources.size(); i++)
            ((Place)sources.get(i)).setToken(false);
    }

    public void createToken() {
        for (int i = 0; i < targets.size(); i++)
            ((Place)targets.get(i)).setToken(true);
    }
    public List getSources() { return sources; }
    public void addSource(Place p) { sources.add(p); }
    public List getTargets() { return targets; }
    public void addTarget(Place p) { targets.add(p); }
    public void setReady(boolean isReady) { this.isReady = isReady; }
```

```
    public void setConcurrent(Transition concurrent) {
        this.concurrent = concurrent;
    }

    private Transition concurrent = this;
    private List sources = new ArrayList();
    private List targets = new ArrayList();
    private boolean isReady = false;
    private String name;
    private int duration;
} // Transition
```

Abbildung 10.6: Implementierung einer Transition

Die Klasse **Transition** implementiert die dynamischen Anteile eines Stellen-Transitionen-Netzes. Jede Transition wird als eigener *Thread* implementiert. Prinzipiell sind alle Transitionen eines Netzes gleichzeitig aktiv und warten auf die Möglichkeit, weiterschalten zu können. Jede Transition führt dazu in einer Schleife die Methode **enabled** aus, die die folgenden drei Voraussetzungen zum Schalten der aktuellen Transition überprüft:

1. Die Transition muss bereit sein. Diese Voraussetzung spielt nur dann eine Rolle, wenn mehrere Transitionen um eine Marke im Vorbereich bzw. eine unbelegte Stelle im Nachbereich konkurrieren. In diesem Fall wird zu jedem Zeitpunkt genau eine der konkurrierenden Transitionen als bereit gekennzeichnet, um ein mehrfaches Verbrauchen bzw. Erzeugen einer Marke in einer Stelle zu verhindern.

2. Jede Stelle im Vorbereich der Transition muss eine Marke enthalten.

3. Keine Stelle des Nachbereichs darf eine Marke enthalten.

Sind alle Voraussetzungen erfüllt, werden zunächst die Marken des Vorbereichs durch Aufruf der Methode **consumeToken** verbraucht. Anschließend arbeitet die Transition für die spezifizierte Zeitdauer, bevor in allen Stellen des Nachbereichs durch die Methode **createToken** neue Marken erzeugt werden. Sowohl nach einem erfolglosen Test der Schaltvoraussetzungen als auch nach einem erfolgreichen Schaltvorgang gibt die Transition den Status „bereit“ an die nächste konkurrierende Transition ab. Konkurrierende Transitionen bilden dazu immer einen Ring, d. h. die letzte Stelle aus der Liste der um eine Stelle konkurrierenden Transitionen wird mit der ersten Transition aus der Liste verbunden und der „bereit“-Status zirkuliert über den Ring.

Auf der Basis der Klassen **Place** und **Transition** kann eine Klasse zur Steuerung eines Systems generiert werden. Dazu werden Schablonen auf sog. Kontexte angewendet. In der prototypischen Implementierung des MOmo-Baukastens sind Kontexte XML-Dokumente, die einzelne Metaobjekte repräsentieren. Bezogen auf die Sprache **PN** kann ein Kontext folglich Instanzen der Klassen **Kante**, **Stelle**, **Marke** und **Transition** darstellen.

Abb. 10.7 zeigt Beispiele für Kontexte, die durch den Kontextgenerator während der Codegenerierung für ein PN-Modell erzeugt werden. Der Wurzelknoten eines Dokumentes gibt den

```
<Stelle>
  <name value="heizen"/>
  <eingabe value="false"/>
</Stelle>

<Kante>
  <name value="kante7"/>
  <knoten>
    <TransitionRef href="heizung_an"/>
    <StelleRef href="heizen"/>
  </knoten>
  <stelle>
    <StelleRef href="heizen"/>
  </stelle>
  <transition>
    <TransitionRef href="heizung_an"/>
  </transition>
</Kante>

<Transition>
  <name value="heizung_an"/>
  <dauer value="0"/>
</Transition>
```

Abbildung 10.7: Beispiele für Kontexte

Typ des Metaobjektes an, das er repräsentiert. Alle Werte und Referenzen, die ein Metaobjekt enthält, werden durch eigene Knoten unterhalb des Wurzelknotens dargestellt. Auf Basis der Kontexte wird die Codegenerierung durch Anwenden von XSLT-Stylesheets durchgeführt.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
  <xsl:import href="connect.xsl"/>
  ...
  <xsl:template match="model">
    public class Control {

      public void do() {
        // verbinde Transitionen mit Stellen
       <xsl:for-each select="KanteRef">
         <xsl:call-template name="connect">
           <xsl:with-param name="edgeNode" select="document(@href)/Kante"/>
         </xsl:call-template>
       </xsl:for-each>

       // bilde Ringe
       List transitions = new ArrayList();
       <xsl:for-each select="KanteRef">
         <xsl:call-template name="addTransition"/>
           <xsl:with-param name="edgeNode" select="document(@href)/Kante"/>
```

```
      </xsl:call-template>
    </xsl:for-each>
    buildRings(transitions);

    // starte Transitionen
    <xsl:for-each select="TransitionRef">
      <xsl:call-template name="startTransition">
        <xsl:with-param name="transitionNode"
                        select="document(@href)/Transition"/>
      </xsl:call-template>
    </xsl:for-each>
  }

private void buildRings(List transitions) {

    List rings = new ArrayList();
    Iterator transitionIterator = transitions.iterator();
    while (transitionIterator.hasNext()) {
        Transition t = (Transition)transitionIterator.next();
        boolean found = false;
        Iterator ringIterator = rings.iterator();
        while (ringIterator.hasNext() && !found) {
            Ring r = (Ring)ringIterator.next();
            if (r.matches(t)) {
                r.add(t);
                found = true;
            }
        }

        if (!found) {
            Ring r = new Ring();
            r.add(t);
            rings.add(r);
        }
    }

    Iterator ringIterator = rings.iterator();
    while (ringIterator.hasNext())
        ((Ring)ringIterator.next()).connect();
}

private class Ring {

    private Set sources = new HashSet();
    private Set targets = new HashSet();
    private List concurrent = new ArrayList();

    public void add(Transition t) {
        concurrent.add(t);
        sources.addAll(t.getSources());
        targets.addAll(t.getTargets());
    }
```

```
        public void connect() {
            for (int i = 0; i < concurrent.size()-1; i++)
                ((Transition)concurrent.get(i)).setConcurrent(
                     (Transition)concurrent.get(i+1));
                ((Transition)concurrent.get(concurrent.size()-1))
                   .setConcurrent((Transition)concurrent.get(0));
        }

        public boolean matches(Transition t) {
            Iterator placeIterator = t.getSources().iterator();
            while (placeIterator.hasNext())
                if (sources.contains(placeIterator.next()))
                    return true;

            placeIterator = t.getTargets().iterator();
            while (placeIterator.hasNext())
                if (targets.contains(placeIterator.next()))
                    return true;

            return false;
        }
    }

      // Zugriffsmethoden für nach aussen verbundene Stellen
      <xsl:for-each select="StelleRef">
        <xsl:call-template name="accessPlace">
          <xsl:with-param name="placeNode"
                          select="document(@href)/Stelle"/>
        </xsl:call-template>
      </xsl:for-each>

      // erzeuge Stellen
      <xsl:for-each select="StelleRef">
        <xsl:call-template name="initPlace">
          <xsl:with-param name="placeNode"
                          select="document(@href)/Stelle"/>
        </xsl:call-template>
      </xsl:for-each>

      // erzeuge Transitionen
      <xsl:for-each select="TransitionRef">
        <xsl:call-template name="initTransition">
          <xsl:with-param name="transitionNode"
                          select="document(@href)/Transition"/>
        </xsl:call-template>
      </xsl:for-each>
    }
  </xsl:template>
</xsl:stylesheet>
```

Abbildung 10.8: Schablone zur Erzeugung einer Steuerung

Abb. 10.8 zeigt die Schablone, die die Klasse **Control** erzeugt. Die Schablone wird auf den Modellkontext angewendet, der Referenzen auf alle Elemente des Modells enthält.[1] Die Schablone erzeugt nur den Rahmen der Klasse und wendet weitere Schablonen auf die Kontexte für die einzelnen Kanten, Stellen und Transitionen an, um den Code für Steuerung der Waschmaschine zu erzeugen. Zunächst wird die Schablone **connect** auf alle Kontexte ausgeführt, die Kanten des Netzes repräsentieren, um die Verbindungen zwischen den Stellen und Transitionen zu erzeugen. Anschließend wird eine Liste der im Modell enthaltenen Transitionen erzeugt, um die oben beschriebenen Ringstrukturen konkurrierender Transitionen zu erzeugen. Der Aufbau der Ringstrukturen erfolgt durch einen Aufruf der Methode **buildRings**. Diese Methode wird ebenfalls durch die Schablone **control** erzeugt. Um Codesequenzen zum Starten der Threads zu generieren, wird die Schablone **startTransition** für jede Transition des Netzes aufgerufen. Die Schablone **accessPlace** erzeugt Methoden, die den Zugriff auf die Stellen ermöglichen, die zur Kopplung von Steuerung und zu steuerndem System vorgesehen sind. Abschließend erzeugen die Schablonen **initPlace** und **initTransition** Variablen für die Stellen und Transitionen des Netzes und initialisieren diese mit einem neu erzeugten Objekt.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
  <xsl:template name="connect">
    <xsl:param name="edgeNode"/>
    <xsl:variable name="from"
      select="document($edgeNode/knoten/*[0]/@href)/*"/>
    <xsl:variable name="to"
      select="document($edgeNode/knoten/*[1]/@href)/*"/>
    <xsl:choose>
      <xsl:when test="local-name($from)='Stelle'">
        <xsl:value-of select="concat($to/name/@value,
          '.addSource(', $from/name/@value, ');')"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="concat($from/name/@value,
          '.addTarget(', $to/name/@value, ');')"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

Abbildung 10.9: Schablone zur Erzeugung der Verbindungen einer Transition

[1] Im Allgemeinen enthält der Modellkontext nur Referenzen auf die Elemente der obersten Schachtelungsebene. Da die Sprache **PN** aber keine Möglichkeiten zur Schachtelung von Elementen enthält, sind hier Referenzen auf alle Elemente des Modells enthalten.

Abb. 10.9 zeigt die Schablone **connect**. Die Schablone erwartet den Wurzelknoten eines Kontextes, der eine Kante repräsentiert. Die Verarbeitung beginnt mit der Abfrage der Wurzelknoten der Kontexte, die die beiden verbundenen Knoten repräsentieren. Der erste **KnotenRef**-Knoten repräsentiert die Quelle, der zweite **KnotenRef**-Knoten das Ziel der Kante. Nachdem beide Wurzelknoten bekannt sind, wird der Knotentyp des Quellknotens untersucht. Handelt es sich um eine Stelle, wird dieser Knoten in den Vorbereich der Transition eingefügt, die durch den Zielknoten dargestellt ist. Andernfalls wird der Nachbereich der Transition, die durch den Quellknoten repräsentiert wird, um die Stelle erweitert, die durch den Zielknoten beschrieben wird. Die Schablone **connect** ist die komplizierteste der von **genControl** verwendeten Schablonen, da zwei Kontexte verarbeitet werden müssen. Alle anderen Schablonen benötigen lediglich die Daten eines Kontextes, um das gewünschte Codefragment zu erzeugen. Da diese Schablonen sich sehr ähnlich sind, soll hier nur eine weitere Schablone gezeigt werden. Abb. 10.10 zeigt die Schablone **initPlace**, die die Deklaration und Initialisierung einer Stelle des Netzes erzeugt.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
  <xsl:template name="initPlace">
    <xsl:param name="placeNode"/>
    private Place <xsl:value-of select="$placeNode/name/@value"/>
      = new Place("<xsl:value-of select="$placeNode/name/@value"/>");
  </xsl:template>
</xsl:stylesheet>
```

Abbildung 10.10: Schablone zur Deklaration und Initialisierung einer Stelle

10.1.3 Entwicklung einer Waschmaschinensteuerung

Mit Hilfe der beschriebenen Abbildung der Elemente von Stellen-Transitionen-Netzen auf Quelltexte der Programmiersprache Java soll jetzt eine Steuerung für folgenden Ablauf entwickelt werden:

- Die Waschmaschine wird mit Wasser befüllt. Wenn genug Wasser eingefüllt wurde, wird dies durch einen Sensor gemeldet.
- Das Wasser wird erwärmt, bis der Thermostat das Erreichen der gewünschten Temperatur meldet.
- Wenn genug Wasser eingefüllt wurde, und das Wasser die gewünschte Temperatur hat, wird zehn Minuten gewaschen.
- Nach dem Waschen wird das Wasser abgepumpt. Ein Sensor meldet, wenn die Maschine leer ist.
- Wenn die Maschine leer ist, wird die Wäsche drei Minuten geschleudert.

- Die Tür der Waschmaschine wird verriegelt, wenn der Waschvorgang gestartet wird. Der Waschvorgang beginnt mit dem Befüllen der Waschmaschine mit Wasser.
- Zwei Minuten nach Beendigung des Waschvorgangs wird die Verriegelung aufgehoben.

Der dargestellte Ablauf wird zunächst durch ein Stellen-Transitionen-Netz modelliert, das die in Abschnitt 10.1.2 beschriebenen Modellierungselemente verwendet. Es wird eine Notation verwendet, die auf der in Abschnitt 10.1.1 vorgestellen Notation für Stellen-Transitionen-Netze basiert. Zusätzlich wird für jede Stelle und jede Transition des Netzes ein Name angegeben. Da die Stellen nur durch einen kleinen Kreis symbolisiert werden, wird der Name neben der entsprechenden Stelle notiert. Die Möglichkeit, die Dauer einer Transition zu beschreiben, wird durch eine Erweiterung des Transitionssymbols notiert. Die Erweiterung besteht aus zwei Linien, die die obere und untere Kante des Rechtecks verlängern, sowie der Zeitangabe, die zwischen den Linien angegeben wird. Abb. 10.11 zeigt eine Darstellung des oben beschriebenen Waschvorgangs.

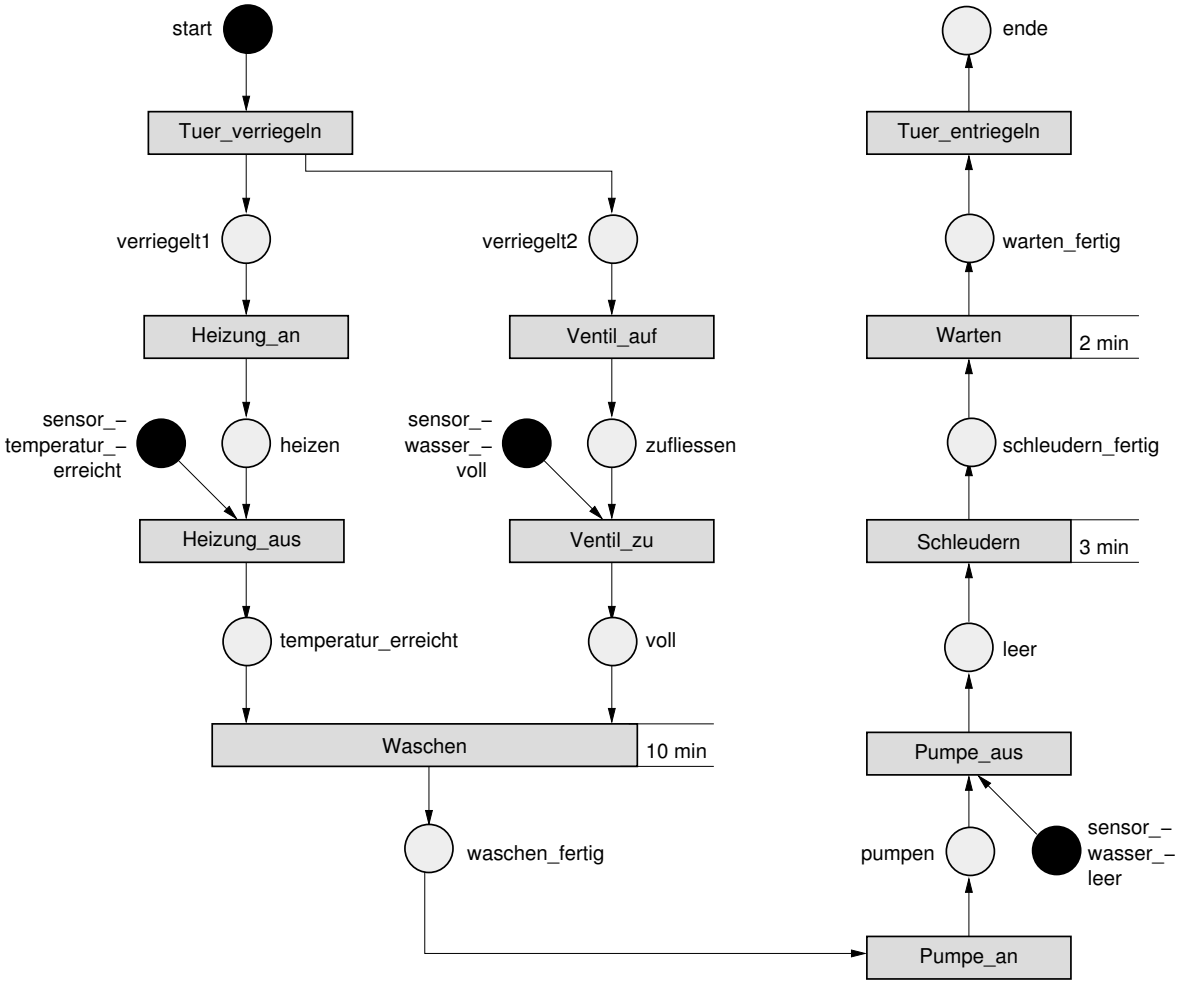


Abbildung 10.11: Stellen-Transitionen-Netz zur Steuerung der Waschmaschine

Die Ablaufbeschreibung in Abb. 10.11 beginnt mit der Stelle **start**. Diese symbolisiert den Zustand des Einschaltknopfes der Waschmaschine. Wird dieser gedrückt, entspricht dies dem Erzeugen einer Marke in der Stelle. Da die Stelle eine Verbindung zum zu steuernden System repräsentiert, die Informationen entgegennimmt, ist sie durch einen schwarz gefüllten Kreis dargestellt. Die Stellen, die die Sensoren für den Wasserstand bzw. die Wassertemperatur in der Maschine symbolisieren, werden auf die gleiche Art gekennzeichnet.

Nachdem in der Stelle **start** eine Marke erzeugt worden ist, schaltet die Transition **Tuer_verriegeln**. Die Marke wird dadurch verbraucht, und es werden zwei Marken in den Stellen **verriegelt1** und **verriegelt2** erzeugt. Die Aufteilung des Zustandes „Tür verriegelt“ in zwei Stellen ist notwendig, um einen Konflikt zwischen den Abläufen zum Einlassen bzw. Aufheizen des Wassers zu vermeiden. Diese Abläufe sollen parallel ausgeführt werden und benötigen daher gleichzeitig eine Marke, um starten zu können.

Das Einlassen des Wassers in die Maschine wird durch die Transition **Ventil_auf** eingeleitet und durch die Transition **Ventil_zu** abgeschlossen. Letztere kann genau dann schalten, wenn die Stellen **zufliessen** und **sensor_wasser_voll** jeweils eine Marke enthalten. Auf dieselbe Weise wird das Erwärmen des Wassers gesteuert. Die Transition **Heizung_an** schaltet die Heizung ein. Anschließend befindet sich das System in der Stelle **heizen**. Die folgende Transition **Heizung_aus**, die den Heizvorgang beendet, kann erst dann schalten, wenn auch die Stelle **sensor_temperatur_erreicht** eine Marke enthält. Diese ist, wie auch **sensor_wasser_voll**, ein Stellvertreter für ein Element des zu steuernden Systems und erhält die Marke erst bei Eintreten des entsprechenden externen Ereignisses.

Das **Waschen** beginnt, wenn Wasserstand und Wassertemperatur den Vorgaben entsprechen, und dauert zehn Minuten. Dies wird simuliert, indem die Marken des Vorbereichs verbraucht werden, und das System anschließend zehn Minuten wartet. Solche Transitionen werden in PN-Modellen stellvertretend für komplexe Abläufe verwendet, die eigentlich durch ein weiteres Netz beschrieben werden müssten. Während der Schaltzeit der Transition **Waschen** könnte beispielsweise die Drehrichtung der Trommel verändert, Wasser abgepumpt und neues Wasser eingelassen werden.

Nach Durchführung der Transition **Waschen** befindet sich eine Marke in der Stelle **waschen_fertig**, so dass die Transition **Pumpe_an** schalten kann. Diese verbraucht die Marke in der Stelle **waschen_fertig**, schaltet die Pumpe ein und erzeugt eine Marke in der Stelle **pumpen**. Hier muss erneut auf ein externes Ereignis gewartet werden. Ein Sensor überprüft, ob die Maschine noch Wasser enthält. Ist dies nicht der Fall, meldet der Sensor, dass die Maschine leer ist. Dies wird durch Erzeugen einer Marke in der Stelle **sensor_wasser_leer** symbolisiert. Diese Marke ermöglicht das Schalten der Transition **Pumpe_an**, die das Pumpen beendet.

Nachdem das Pumpen beendet worden ist, befindet sich eine Marke in der Stelle **leer**, die das Schalten der Transition **Schleudern** ermöglicht. Wiederum verbraucht die Transition die Marke. Anschließend wird die Verarbeitung für drei Minuten unterbrochen. Diese Zeit repräsentiert das Schleudern der Wäsche durch die Maschine. Nachdem die drei Minuten abgelaufen sind, wird eine Marke in der Stelle **schleudern_fertig** erzeugt. Diese Marke wird durch die Transition **Warten** verbraucht, die das zweiminütige Warten vor der Entriegelung der Tür übernimmt. **Warten** erzeugt eine Marke in der Stelle **warten_fertig**, die durch die Transition

Tuer_entriegeln verbraucht wird. Diese Transition überführt das System in seinen Endzustand, der durch eine Marke in der Stelle **ende** gekennzeichnet ist.

Mit Hilfe des MOmo-PN-Codegenerators, der in Abschnitt 10.1.2 beschrieben wurde, soll nun eine Java-Implementierung einer Steuerung des beschriebenen Ablaufs entwickelt werden. Dazu wird ein XMI-Dokument benötigt, das das Modell aus Abb. 10.11 beschreibt. Ein Ausschnitt aus einem entsprechenden Dokument ist in Abb. 10.12 dargestellt.

```
<xmi:XMI xmi:version="2.1"
         xmlns:xmi="http://www.omg.org/XMI"
         xmlns:pn="http://ist.unibwm.de/PN">

  <pn:Stelle xmi:id="s001" name="start" eingabe="true"/>
  <pn:Kante xmi:id="k001" knoten="s001 t001"
            stelle="s001" transition="t001"/>
  <pn:Transition xmi:id="t001" name="Tuer_verriegeln"/>
  <pn:Kante xmi:id="k002" knoten="t001 s002"
            stelle="s002" transition="t001"/>
  <pn:Kante xmi:id="k003" knoten="t001 s003"
            stelle="s003" transition="t001"/>
  <pn:Stelle xmi:id="s002" name="verriegelt1" eingabe="false"/>
  <pn:Stelle xmi:id="s003" name="verriegelt2" eingabe="false"/>

   ...

</xmi:XMI>
```

Abbildung 10.12: XMI-Dokument für das Beispielnetz

Der PN-Codegenerator liest das XMI-Dokument und baut eine Darstellung des Modells durch Instanzen der Elemente des PN-Metamodells auf. Im nächsten Schritt werden die Kontexte für die einzelnen Elemente des Modells erzeugt. Im hier verwendeten Beispiel repräsentieren Kontexte Kanten, Stellen und Transitionen. Beispiele für den Aufbau der Kontexte wurden bereits in Abb. 10.7 dargestellt. Nach der Erzeugung der Kontexte werden die Schablonen angewendet, die in Abschnitt 10.1.2.3 beschrieben sind. Welche Schablonen auf welche Kontextarten anzuwenden sind, wird durch die Konfiguration bestimmt. Da lediglich eine Klasse erzeugt werden soll, muss hier nur die Schablone **control**, die in Abb. 10.8 dargestellt ist, auf den Modellkontext angewendet werden. Ein Ausschnitt des generierten Quelltextes ist in Abb. 10.13 dargestellt.

```
public class Control {
    public void do() {
        // verbinde Transitionen mit Stellen
        tuer_verriegeln.addSource(start);
        tuer_verriegeln.addTarget(verriegelt1);
        tuer_verriegeln.addTarget(verriegelt2);
        heizung_ein.addSource(verriegelt1);
        ventil_auf.addSource(verriegelt2);
        ...

        // bilde Ringe
        List transitions = new ArrayList();
        transitions.add(tuer_verriegeln);
        ...
        buildRings(transitions);

        // starte Transitionen
        tuer_verriegeln.start();
        ...
    }

    private void buildRings(List transitions) {
        ...
    }

    private class Ring {
        ...
    }

    // accessors for accessible places
    public Place get_start() { return start; };
    public Place get_sensor_wasser_voll() { return sensor_wasser_voll; };
    public Place get_sensor_temperatur_erreicht() {
      return sensor_temperatur_erreicht;
    };
    public Place get_sensor_wasser_leer() { return sensor_wasser_leer; };

    // create places
    private Place start = new Place("start");
    private Place verriegelt1 = new Place("verriegelt1");
    ...

    // create transitions
    private Transition tuer_verriegeln
        = new Transition("tuer_verriegeln", 0);
        private Transition heizung_ein = new Transition("heizung_ein", 0);
    private Transition ventil_auf = new Transition("ventil_auf", 0);
    ...

} // Control
```

Abbildung 10.13: Ausschnitt des generierten Quelltextes der Waschmaschinensteuerung

10.2 MOF

Im vorhergehenden Abschnitt 10.1 wurde eine vollständige Darstellung der einzelnen Schritte zur Implementierung eines MOmo-Codegenerators für eine einfache Petri-Netz-Sprache angegeben, um die Vorgehensweise bei der Erstellung eines MOmo-Codegenerators zu zeigen. Die einzelnen Elemente der Petri-Netz-Sprache PN wurden durch ein MOF-Modell festgelegt und durch den MOmo-MOF-Codegenerator in eine Implementierung überführt. Diese Schritte können für jede Sprache durchgeführt werden, die durch ein MOF-Modell beschrieben werden kann. Dies gilt insbesondere für die durch die von der OMG standardisierten Modellierungssprachen MOF und UML, deren Implementierung als Grundlage für viele Werkzeuge zur Softwareentwicklung benötigt wird. In den folgenden Abschnitten werden die Bestandteile (Abschnitt 10.2.1) und das Laufzeitverhalten (Abschnitt 10.2.2) des MOmo-MOF-Codegenerators beschrieben.

10.2.1 Codegenerator

Der Aufbau des MOmo-MOF-Codegenerators entspricht prinzipiell dem Aufbau des PN-Codegenerators, der in Abschnitt 10.1.2 dargestellt ist. Die beiden Codegeneratoren unterscheiden sich in der verwendeten Metamodell-Implementierung und den Schablonen, die zur Codeerzeugung verwendet werden. Zum Erstellen des MOF-Codegenerators wird zunächst eine Implementierung des MOF-Metamodells erzeugt. An dieser Stelle erzeugt der MOmo-MOF-Codegenerator eines seiner eigenen Bestandteile. Während der Entwicklung des MOmo-Baukastens wurde zunächst eine Teilmenge der MOF manuell implementiert. Anschließend folgten mehrere Iterationszyklen, in denen der entstandene Codegenerator schrittweise erweitert wurde, bis er seine eigene Metamodell-Implementierung erzeugen konnte.

Die Realisierung der einzelnen Bestandteile des MOF-Modells entsprechen den Realisierungen der Bestandteile des PN-Metamodells (siehe Abb. 10.3). Die generierte Implementierung des MOF-Metamodells wurde mit den generischen Komponenten, die in Kapitel 9 beschrieben wurden, verbunden, so dass eine Ablaufumgebung für MOF-Codegeneratoren entsteht. Dies ist in Abb. 10.14 dargestellt.

Die beiden „durch sich selbst“ erzeugten Bestandteile sind durch den Pfeil gekennzeichnet. Neben der vollständig generierten Implementierung des MOF-Metamodells wird auch der *XMI-Reader* teilweise generiert. Die generierten Teile sind die Anteile, die zur Erzeugung der Instanzen des sprachspezifischen Metamodells benötigt werden. Daneben enthält der Code des XMI-Reader aber auch generische Anteile, die das Einlesen eines XML-Dokumentes für alle MOmo-XMI-Reader übernehmen.

Neben den generierten und generischen Modulen enthält der MOmo-MOF-Codegenerator manuell erstellte Bestandteile. Den größten Umfang haben die Schablonen zur Erzeugung der einzelnen Artefakte einer MOmo-JMI-Implementierung. Der MOF-Codegenerator umfasst derzeit ca. 240 Schablonen mit einem Gesamtumfang von ca. 2MB zur Erzeugung der MOF-Modell-Implementierungen. Außerdem umfasst der MOmo-MOF-Codegenerator

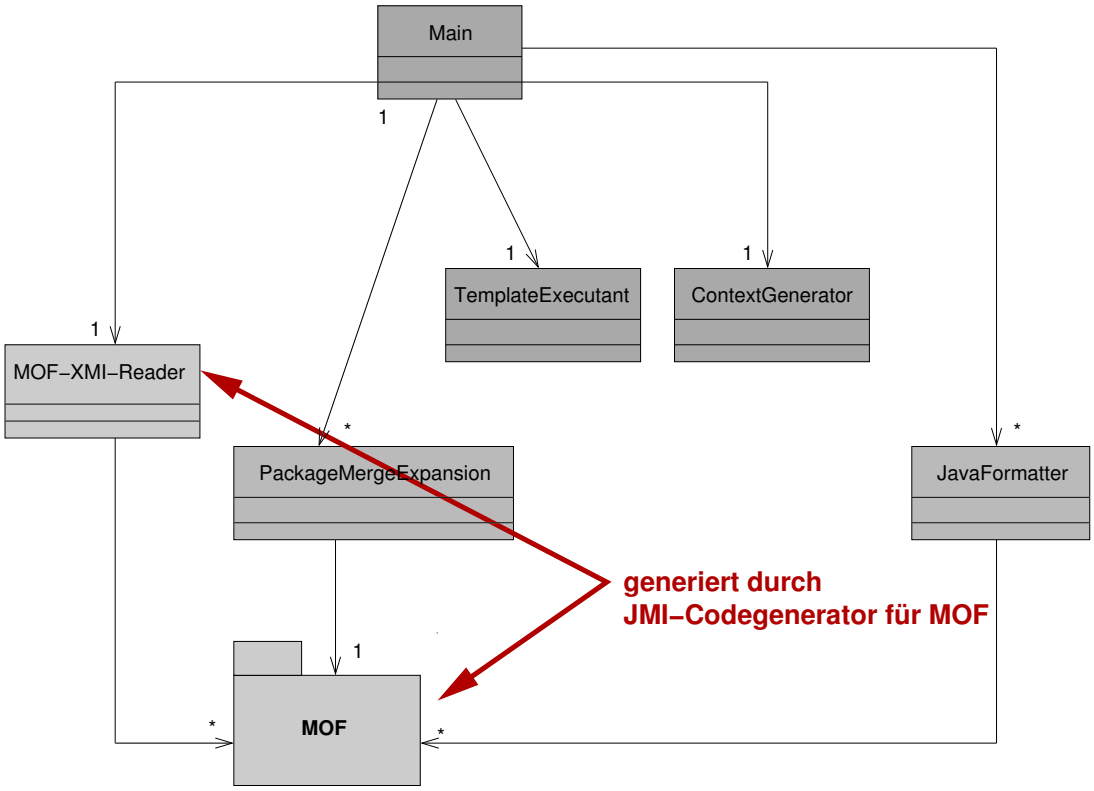


Abbildung 10.14: MOmo-MOF-Codegenerator

zwei manuell erstellte Module, die speziell für die Eingabesprache (PackageMergeExpansion-Modul) bzw. die Ausgabesprache (JavaFormatter-Modul) erstellt wurden.

Der MOF-Codegenerator wurde im Rahmen der Arbeit zur Erzeugung von Implementierungen verschiedener Versionen der MOF und der UML angewendet, um die Praxistauglichkeit des gewählten Ansatzes zu evaluieren. Die erste Anwendung bestand aus dem Ersatz der Bibliothek *nsuml* [NSU], die vom UML-Werkzeug ArgoUML [Arg] verwendet wird. Diese Bibliothek implementiert in ihrer letzten Version die UML-Version 1.3 sowie die XMI-Version 1.0 und stellt die Klassen des UML-Metamodells zur Verfügung. Da *nsuml* nicht mehr weiterentwickelt wurde, hat das ArgoUML-Projekt beschlossen, auf das MOF-Repository *mdr* [Mat03] umzustellen. Da sich die Schnittstellen von *nsuml* und *mdr* deutlich unterscheiden, ist die Umstellung sehr aufwändig und konnte bis heute nicht beendet werden. Zudem ergibt sich dasselbe Problem mit der Verwendung von *mdr* in abgeschwächter Form erneut, da diese nur die MOF-Version 1.4 implementiert und somit nur für Instanzen der MOF 1.4 verwendet werden kann. Der Umstieg auf die Version 2.0 der UML wird daher einen weiteren Wechsel der verwendeten Metamodellimplementierung erfordern. Durch die Verwendung eines flexibleren Ansatzes wie MOmo ließe sich die Anpassung UML-basierter Werkzeuge deutlich vereinfachen und beschleunigen. Als Demonstration der größeren Flexibilität wurden im Rahmen der Arbeit nsuml-kompatible Implementierungen der UML-Versionen 1.4 und 1.5 generiert und mit dem ArgoUML-Werkzeug verbunden. Dabei wurde die nsuml-Bibliothek vollständig ersetzt.

Mit dem Erscheinen der MOF-Version 2.0 wurde der MOmo-MOF-Codegenerator auf diese Version angepasst und zur Erzeugung von Implementierungen der MOF 2.0 und der UML-Version 2.0 verwendet. Die Abwärtskompatibilität zu älteren MOF-Versionen erfolgt durch spezielle XMI-Reader, die die Elemente alter MOF-Modelle auf MOF 2.0 abbilden. Auf diese Weise können mit der neuen MOF 2.0 basierten Infrastruktur sowohl die älteren Versionen der MOF als auch die Version 2.0 verarbeitet werden.

10.2.2 Laufzeitverhalten

In diesem Abschnitt werden Angaben über das Laufzeitverhalten des MOmo-MOF-Codegenerators angegeben. Die Bestandteile dieses Codegenerators entsprechen mit Ausnahme der Schablonen den Bestandteilen aller anderen MOmo-Codegeneratoren, so dass die Ergebnisse auch für alle anderen MOmo-Codegeneratoren gelten. Die Messungen des Laufzeitverhaltens wurden auf einem PC (Intel™ Pentium IV mit 1700 Mhz und 1 GB RAM) durchgeführt. Tab. 10.1 zeigt die Daten, die während der Erzeugung von Metamodellimplementierungen für die Metamodelle der Modellierungssprachen PN, MOF 1.4, UML 1.4, MOF 2.0 und UML 1.5 ermittelt wurden. Auf die Darstellung der Ergebnisse des Generierungsprozesses der UML 2.0 wird hier verzichtet, da der verwendete PC nicht über ausreichend Hauptspeicher verfügt, um das Metamodell der UML 2.0 ohne Auslagern auf die Festplatte erzeugen zu können. Dies liegt an der hohen Anzahl von Klassen[2], die für eine MOmo-JMI-Implementierung der UML 2.0 erzeugt werden müssen.

	PN	**MOF 1.4**	**UML 1.4**	**MOF 2.0**	**UML 1.5**
Anzahl generierter Klassen	37	134	724	1024	1156
Quelltextgröße (in KByte)	1120	1528	6234	9216	9608
Codegröße (in KByte)	296	626	2652	3734	4128
Laufzeit Generieren (in Sekunden)	14	15	27	112	35
Laufzeit Kompilieren (in Sekunden)	10	11	19	38	24

Tabelle 10.1: Laufzeitverhalten des MOmo-MOF-Codegenerators

Die Tabelle zeigt, dass das Laufzeitverhalten des MOmo-Codegenerators bei der Codeerzeugung für die Metamodelle der MOF 1.4, UML 1.4 und UML 1.5 gut skaliert, während für die Erzeugung der Metamodellimplementierung der MOF 2.0 eine deutlich längere Zeit benötigt wird. Dies liegt zum einen an den *Vereinigungsbeziehungen* zwischen Paketen (siehe Abschnitt 5.3.4.3), die in der Spezifikation der MOF 2.0 verwendet werden. Diese müssen vor der Codegenerierung expandiert werden und verursachen zudem einen deutlich höheren Analyseaufwand, da bei der Erzeugung des Codes für Attribute ständig komplexe Vererbungshierarchien durchlaufen werden müssen, um implizite Redefinitionsbeziehungen zu erkennen. Zum anderen ist das MOF 2.0 Metamodell auch das einzige Modell, das *subsets*- und *redefines*-Beziehungen (siehe Abschnitt 5.3.1.1) enthält, die für die Codegenererierung berücksichtigt werden müssen.

[2]Der MOmo-MOF-Codegenerator erzeugt für den aktuellen Stand der UML 2.0 14000 Klassen

Auch die Erzeugung des Quelltextes für die Metamodellimplementierung der Beispielsprache PN dauert vergleichsweise lange. Der Grund dafür ist, dass die tatsächliche Codeerzeugung in diesem Beispiel nur noch sehr geringen Einfluss auf die Dauer des Generierungsprozesses hat, während die statischen Anteile, z. B. die Initialisierung des Interpreters und das Einlesen des XMI-Dokumentes den größten Teil der Zeit in Anspruch nehmen.

Die Erzeugung jedes einzelnen Artefaktes ist unabhängig von der Erzeugung aller anderen Artefakte während eines Generatorlaufs. Aus diesem Grund ist die Zeitdauer, die zur Erzeugung einer Metamodellimplementierung benötigt wird, fast linear abhängig von der Rechenleistung der verwendeten Maschine. Beispielsweise benötigt die Erzeugung der Implementierung des MOF 2.0 Metamodells auf einem Pentium-IV-PC mit 3000 Mhz 62 Sekunden. Aufgrund der Unabhängigkeit der Generierungsprozesse für die einzelnen Artefakte ist eine ähnlich gute Skalierung auch bei einer verteilten Implementierung auf Mehrprozessormaschinen zu erwarten. Dazu müsste lediglich die Erzeugung der einzelnen Kontexte und Artefakte in den Kontextgenerator- und Schablonenausführer-Komponenten in einzelne Threads verlagert werden.

10.3 Zusammenfassung

In diesem Kapitel wurde zunächst die Anwendung des MOmo-Baukastens gezeigt (Abschnitt 10.1), indem ein Codegenerator für Stellen-Transitionen-Netze erstellt und zur Generierung einer einfachen Waschmaschinensteuerung eingesetzt wurde. Es wurden in Abschnitt 10.1.1 die Bestandteile eines Stellen-Transitionen-Netzes erläutert. Anschließend erfolgte in Abschnitt 10.1.2 die Entwicklung des Codegenerators für die Stellen-Transitionen-Sprache **PN**. Die Definition der Modellierungssprache PN erfolgte durch ein MOF-Modell, das als Eingabe für den MOmo-MOF-Codegenerator verwendet werden kann.

Mit Hilfe des Generators wurde eine Implementierung des Modells erzeugt, die die Grundlage eines MOmo-Codegenerators für PN darstellt. Es zeigt sich, dass selbst einfache Modellierungssprachen wie die PN verhältnismäßig komplexe und dementsprechend aufwändig zu erstellende Metamodell-Implementierungen erfordern. Die automatische Erzeugung solcher Implementierungen stellt daher einen wichtigen Beitrag zur Reduzierung des Aufwandes dar, der zur Entwicklung eines Codegenerators erbracht werden muss. Neben der Implementierung des PN-Metamodells werden die generischen Komponenten des MOmo-Baukastens und eine Abbildung der Elemente des PN-Metamodells auf Codefragmente benötigt, um einen vollständigen PN-Codegenerator zu erhalten. Eine einfache Abbildung auf Java-Codefragmente wurde entworfen und durch XSLT-Stylesheets implementiert.

In Abschnitt 10.1.3 wurde die Anwendung des PN-Codegenerators demonstriert. Dazu wurde eine Spezifikation einer einfachen Waschmaschinensteuerung angegeben und mit der Modellierungssprache PN modelliert. Anschließend erfolgte die Übersetzung des Modells mit Hilfe des PN-Codegenerators in einen Java-Quelltext, der den beschriebenen Ablauf implementiert.

Abschnitt 10.2 beschäftigte sich mit der Codegenerierung für **MOF**-Modelle durch den im MOmo-Baukasten enthaltenen MOF-Codegenerator. Abschnitt 10.2.1 gab einen Überblick

über den Aufbau des im MOmo-Baukasten enthaltenen MOF-Codegenerators. Anschließend wurde in Abschnitt 10.2.2 am Beispiel dieses Codegenerators das Laufzeitverhalten von MOmo-Codegeneratoren dargestellt. Es konnte gezeigt werden, dass MOmo-Codegeneratoren gut mit der Komplexität der zu übersetzenden Modelle skalieren. Abschließend wurden einige Möglichkeiten skizziert, die zur Beschleunigung des Codegenerierungsprozesses verwendet werden könnten.

Kapitel 11

Schlussbemerkungen

In den vorangegangenen Kapiteln wurden das Konzept und die Realisierung eines Baukastens beschrieben, der die Implementierung von Codegeneratoren für Modellierungssprachen erleichtert. Außerdem wurden zwei Beispiele angegeben, die die Anwendung des Baukastens zur Erzeugung von Codegeneratoren für MOF-basierte Modellierungssprachen demonstrieren und die Praxistauglichkeit des MOmo-Baukastens zeigen. Im Folgenden werden Konzept und Realisierung des MOmo-Baukastens noch einmal zusammengefasst. Insbesondere wird auf die Umsetzung der einzelnen Anforderungen eingegangen, die für die Entwicklung des MOmo-Baukastens maßgeblich waren. Abschließend erfolgt ein Ausblick auf weiterführende Arbeiten.

11.1 Zusammenfassung

Der MOmo-Baukasten soll die Entwicklung von Codegeneratoren für alle Modellierungssprachen unterstützen, deren Syntax in der Metamodellierungssprache MOF [ACC+03] beschrieben werden kann. Dies unterscheidet ihn von anderen Ansätzen, die nur die Implementierung von Codegeneratoren für genau eine Modellierungssprache erlauben. Im Einzelnen erfüllt der MOmo-Baukasten folgende Anforderungen:

1. Das Eingabeformat für einen MOmo-Codegenerator soll *XML Metadata Interchange* (XMI) [Obj03b] sein. XMI ist eine Abbildung der MOF auf XML und kann daher Modelle aller Modellierungssprachen, die durch ein MOF-Modell festgelegt sind, repräsentieren.

2. Möglichst viele Komponenten eines Codegenerators sollen durch Werkzeuge automatisch erzeugt werden können. Ist dies nicht möglich oder nicht sinnvoll, sollen alternativ generische Komponenten bereitgestellt werden, die sich für alle MOmo-Codegeneratoren verwenden lassen.

3. Die Anpassbarkeit generierter Quelltexte ist ein wichtiger Faktor, der die Wartbarkeit aller auf dem generierten Code basierenden Anwendungen maßgeblich beeinflusst. Aus diesem Grund müssen kleine Änderungen am zu generierenden Code leicht durchführbar sein.

4. Obwohl die Dauer eines Generierungsprozesses in der Regel gegenüber dem insgesamt erforderlichen Zeitaufwand für die Implementierung einer Anwendung sehr gering ist, ist sie doch ein bedeutendes Kriterium für die Benutzbarkeit eines Generators. Aus diesem Grund sollen MOmo-Codegeneratoren zumindest mit dem Umfang des Modells und der Komplexität der Modellierungssprache skalieren. Außerdem soll die Zeit, die zum Erzeugen der Quelltexte benötigt wird, die zum Übersetzen der Quelltexte notwendige Dauer nicht wesentlich überschreiten.

5. Das Konzept des MOmo-Baukastens soll unabhängig von der verwendeten Implementierungstechnologie sein.

6. Sowohl der Baukasten als auch die Codegeneratoren sollen soweit wie möglich durch standardisierte Werkzeuge und Technologien realisiert werden.

Anforderung 1 wurde umgesetzt, indem ein MOF-Codegenerator erstellt wurde, der für jedes MOF-Modell eine Implementierung generieren kann, die Klassen zum Lesen und Schreiben von XMI-Dokumenten umfasst. Dazu wurde eine Abbildung der Elemente der Modellierungssprache MOF auf Schnittstellen und Klassen der Programmiersprache *Java* [GJSB00] definiert. Die Abbildung basiert auf der *Java Metadata Interface*-Spezifiaktion (JMI) [Dir02], die eine entsprechende Abbildung für frühere MOF-Versionen festlegt. Im Rahmen der Arbeit wurde eine Erweiterung bzw. Anpassung der Bestandteile der JMI-Spezifikation an die MOF 2.0 entwickelt. Dies erforderte Änderungen an den zu generierenden Schnittstellen, um die neuen bzw. gegenüber MOF 1.4 veränderten oder erweiterten Elemente abzubilden. Dies gilt insbesondere für das Modellelement „Assoziation“ [ABS04]. Außerdem wurden weitere Änderungen vorgenommen, um eine klarere Abbildung einiger Elemente zu erreichen. Als Erweiterung der JMI-Spezifikation umfasst die MOmo-Metadaten-Implementierung auch Vorschläge für die Implementierungen der JMI-Schnittstellen. Im Unterschied zu anderen JMI-Implementierungenwie *nsmdr* [Mat03] kommt die MOmo-JMI-Implementierung ohne Verwendung des Java-Reflexions-Mechanismus' aus, und unterstützt bereits die neuen Mechanismen der MOF-Version 2.0.

Anforderung 2 wird zum Teil durch den MOF-Codegenerator erfüllt, der im MOmo-Baukasten enthalten ist. Darüber hinaus stellt der Baukasten generische Implementierungen weiterer Komponenten von MOmo-Codegeneratoren zur Verfügung, die auf der generierten Implementierung des Metamodells der zu verarbeitenden Modellierungssprache aufsetzen. Die generischen Komponenten steuern den Generierungsprozess, erstellen sog. Kontexte und wenden Schablonen auf die Kontexte an. Ein Kontext ist in diesem Zusammenhang eine Darstellung eines einzelnen Modellelementes, die die zur Codegenerierung notwendigen Informationen so repräsentiert, dass sie durch Schablonen leicht ausgewertet werden können. Im Rahmen der Arbeit wurde der MOF-Codegenerator zur Erzeugung von Implementierungen der Metamodelle verschiedener

UML-Versionen verwendet. Es konnte gezeigt werden, dass der Ansatz sowohl zur Umsetzung komplexer Implementierungen als auch zur Umsetzung umfangreicher Modelle geeignet ist.

Die Realisierung des MOmo-Baukastens, die im Rahmen dieser Arbeit entwickelt wurde, verwendet XSLT [Cla99b] zur Beschreibung der Schablonen und erfüllt damit die Anforderung 6. XSLT ist auf die Verarbeitung von XML-Dokumenten spezialisiert, so dass die Kontexte durch XML-Dokumente implementiert werden. Der grundlegende Unterschied zwischen den XMI-Dokumenten und einem XML-Kontextdokument ist, dass das XMI-Dokument ein komplettes Modell beschreibt, während das Kontextdokument genau ein Modellelement darstellt.

Anforderung 3 wird durch den schablonenbasierten Ansatz ebenfalls erfüllt. Kleine Änderungen am zu generierenden Code können durch Änderungen der Schablonen durchgeführt werden. Da jede Schablone in der Regel lediglich einzelne Modellelemente verarbeitet, d. h. Codefragmente für einzelne Bestandteile des Modells erzeugt, sind die Schablonen vergleichsweise einfach und daher auch entsprechend einfach anzupassen. Weiterhin hat die Aufteilung eines Modells positive Auswirkungen auf den Zeitbedarf eines Codegenerierungsprozesses. Dies gilt insbesondere bei dem gewählten XSLT-basierten Ansatz, da die Dokumentgröße deutlichen Einfluss auf die Verarbeitungsgeschwindigkeit eines XSLT-Prozessors hat. Nach den bisherigen Erfahrungen ist die Zeitdauer eines Generierungsprozesses abhängig von der Komplexität des zu übersetzenden Modells. Die Erfüllung der Anforderung 4 kann in den meisten Fällen gewährleistet werden.

Das Konzept des MOmo-Baukastens ist problemlos auf andere Implementierungstechnologien übertragbar und erfüllt daher die Anforderung 5, weil es nur die Mechanismen erfordert, die durch die Modellierungssprache MOF bereitgestellt werden. Aus diesem Grund ist auch die gewählte Realisierung des Baukastens nicht an die Programmiersprache Java gebunden, sondern es kann jede objektorientierte Programmiersprache verwendet werden, die Anbindungen an XML und XSLT enthält. Daraus ergeben sich zwei wesentliche Vorteile des MOmo-Ansatzes gegenüber Ansätzen mit vergleichbarer Zielstellung wie AndroMDA [And], Open ArchitectureWare [Voe], CodaGen [CTC] und ArcStyler [Obj]:

1. Der MOmo-Ansatz ist nicht an eine bestimmte Implementierungstechnologie gebunden. Die Implementierung der Schablonenablaufumgebung kann in nahezu jeder objektorientierten Programmiersprache erfolgen, und der Ansatz ist in der Lage, beliebige Artefakte zu erzeugen.

2. Der MOmo-Ansatz ermöglicht die Implementierung von Codegeneratoren für alle Modellierungssprachen, deren Syntax durch ein MOF-Modell festgelegt wird. Die einzigen Anteile entsprechender Codegeneratoren, die vom Anwender implementiert werden müssen, sind die Schablonen, die die Abbildung von Elementen der Modellierungssprache auf Artefakte der Zielsprache beschreiben.

Die prototypische Implementierung des MOmo-Baukastens, die im Rahmen dieser Arbeit entwickelt wurde, ist weitgehend abgeschlossen. Lediglich einige Schnittstellen der Metamodellimplementierung, insbesondere Reflexion und XMI, sind noch nicht fertiggestellt. An der Vervollständigung der Implementierungen dieser Schnittstellen wird zur Zeit gearbeitet. Der aktuelle Stand der Implementierung ist immer unter *http://gforge.es.e-technik.tu-darmstadt.de/projects/momoc* erhältlich.

11.2 Ausblick

Softwaresysteme müssen in immer kürzerer Zeit entwickelt werden. Gleichzeitig steigen die qualitativen Anforderungen an die zu entwickelnden Systeme an. Diesem Problem kann durch die Erhöhung des Abstraktionsniveaus begegnet werden. Das Anheben des Abstraktionsniveaus erfordert neuartige Werkzeuge, die einen Teil der bisherigen Entwicklungsarbeit automatisch durchführen können. Wichtige Werkzeuge in diesem Zusammenhang sind Modelltransformatoren und Codegeneratoren, die eine schrittweise Überführung des Modells eines Softwaresystems in eine Implementierung ermöglichen.

Mit dieser Arbeit liegt eine Implementierung eines Baukastens vor, der die Entwicklung von Codegeneratoren für Modellierungssprachen unterstützt. Entsprechende Codegeneratoren können einen wichtigen Beitrag zur Realisierung von Konzepten wie der *Model Driven Architecture* (MDA) [MM01] leisten und so zur Anhebung des Abstraktionsniveaus bei der Entwicklung von Softwaresystemen beitragen. Dazu müssen sie allerdings mit weiteren Werkzeugen zur Softwareentwicklung gekoppelt werden, um eine durchgängige Werkzeugunterstützung zu realisieren.

Die weiterführenden Arbeiten können in zwei Kategorien unterschieden werden, die im Folgenden skizziert werden: (1) Erweiterungen des MOmo-Baukastens, die eine weiterführende Unterstützung des Entwicklers von Codegeneratoren ermöglichen, und (2) Anwendungen des MOmo-Baukastens, d. h. der Einsatz von Komponenten des Baukastens als Teil einer Werkzeugkette, die einen Softwareentwicklungsprozess unterstützt.

11.2.1 Erweiterungen des MOmo-Baukastens

Der MOmo-Baukasten unterstützt in seiner vorliegenden Form die Entwicklung von Codegeneratoren, indem die Grundbestandteile eines Codegenerators durch den Baukasten bereitgestellt bzw. erzeugt werden können. Der Entwickler kann sich vollständig auf die Definition der Sprache sowie die Definition von Abbildungen der Sprache auf Implementierungstechnologien konzentrieren. Sinnvolle Erweiterungen für den MOmo-Baukasten sind daher Werkzeuge, die den Entwickler bei der Durchführung dieser Arbeiten unterstützen.

Derzeit gibt es keinen Editor, der die Konstrukte der MOF-2.0 vollständig unterstützt und Modelle in der XMI-Version 2.1 speichern kann. Deshalb muss die Definition der Metamodelle, die den Sprachumfang einer Modellierungssprache festlegen, derzeit mit Hilfe eines UML-Werkzeuges erfolgen. Dieses Problem wird mit der offiziellen Freigabe der Version-2.0 der UML sowie der Verfügbarkeit entsprechender Werkzeuge teilweise gelöst werden. Allerdings enthält die MOF einige Erweiterungen gegenüber der UML-Infrastruktur, die in der UML-Superstruktur nicht enthalten sind. Um die MOF vollständig nutzen zu können, wird daher ein Editor für MOF-Modelle benötigt. Der MOmo-Baukasten kann zur Entwicklung eines entsprechenden Editors verwendet werden, da er eine Implementierung des MOF-Metamodells erzeugen kann. Eine solche Implementierung wird derzeit an der TU Darmstadt in Form einer MOF-Editorintegration in das UML- und Graphtransformationswerkzeug FUJABA [FUJ] entwickelt.

Des Weiteren sollte das Erstellen der Schablonen unterstützt werden, die die Abbildung der Elemente einer Modellierungssprache auf eine Implementierungstechnologie festlegen. Zur Zeit müssen die Schablonen ohne Unterstützung durch den MOmo-Baukasten in XSLT geschrieben und verwaltet werden. Um das Erstellen zu unterstützen, könnte ein Editor für XSLT, z. B. *jEdit* [PG03], um eine Anbindung an den MOmo-Baukasten erweitert werden. Darüber hinaus ist auch eine Entwicklungsumgebung denkbar, die den XSLT-Anteil gegenüber dem Anwender „versteckt“ und eine kompaktere Beschreibungssprache für die Schablonen zur Verfügung stellt.

In beiden Fällen könnte ein MOF-Codegenerator verwendet werden, um XML-Schemata zu erzeugen, die den Aufbau der Kontextdokumente beschreiben. Die XML-Schemata lassen sich verwenden, um dem Entwickler von Schablonen genaue Informationen über den Aufbau der Kontexte zur Verfügung zu stellen. Auf diese Weise ließe sich das Entwickeln der Schablonen deutlich vereinfachen.

Außerdem sollte der Entwickler bei der Verwaltung seiner Schablonen unterstützt werden. Dies würde die Entwicklung von Anpassungen einer Schablone vereinfachen und damit eine bessere Umsetzung der Anforderung 3 ermöglichen. Vorstellbar ist hier ein „Repository“, das die Schablonen enthält und die Variantenbildung unterstützt. Ein entsprechendes Repository sollte in die Entwicklungsumgebung für Schablonen integriert werden, um seine Vorteile in vollem Umfang nutzbar zu machen.

11.2.2 Anwendung des MOmo-Baukastens

Die Codegeneratoren, die bislang mit Hilfe des MOmo-Baukastens entwickelt wurden, setzen nahezu ausschließlich strukturelle Elemente einer Modellierungssprache in Quelltexte einer Programmiersprache um. Derartige Codegeneratoren profitieren besonders deutlich von der Aufteilung eines Modells in seine Modellierungselemente. Es ist aber zu erwarten, dass dies auch für Codegeneratoren gilt, die das modellierte Verhalten eines Systems in Quelltext transformieren, da auch in diesem Fall oft Quelltextfragmente für einzelne Modellelemente erzeugt werden müssen. Entsprechende Generatoren wurden bislang nur für Abbildungen sehr geringer Komplexität entwickelt. Es sind daher weiterführende Untersuchungen notwendig, um endgültige Aussagen über die Anwendbarkeit des Konzeptes für dieses Einsatzgebiet treffen zu können.

Die Entwicklung eines Codegenerators für eine Modellierungssprache erfordert eine genaue Festlegung der Semantik der verschiedenen Modellierungselemente, die die Sprache bereitstellt. Diese Festlegung ist einer der größten Schwachpunkte der UML, weil die Semantik hauptsächlich durch textuelle Beschreibungen definiert wird (bzw. werden soll). Neben formalen Lösungsansätzen dieses Problems könnte die Semantik einer Modellierungssprache auch durch eine Referenzimplementierung der Sprache festgelegt werden. Die Übertragung dieses Ansatzes auf domänenspezifische Erweiterungen oder Anpassungen der UML (siehe z. B. [BRS01] oder [BRS04]) erfordert die Entwicklung entsprechender Referenzimplementierungen, die durch den Einsatz der Komponenten des MOmo-Baukastens effizienter erstellt werden können.

Die Komponenten des MOmo-Baukastens lassen sich mit Elementen anderer Ansätze kombinieren, um die Implementierungsphase eines Softwareentwicklungsprozesses zu automatisieren. In diesem Zusammenhang ist insbesondere die Kopplung eines MOmo-Codegenerators mit einem Ansatz zur Modelltransformation interessant. Vielversprechend ist die Anwendung eines MOmo-Codegenerators, der ein Modell, das bereits durch die Elemente einer Programmiersprache beschrieben wird, auf Artefakte derselben Programmiersprache abbildet. Die Abbildung ist vergleichsweise einfach, so dass die Fehleranfälligkeit deutlich reduziert werden könnte. Durch einen Modelltransformator könnte beispielsweise ein UML-Modell in ein Java-Modell transformiert werden. Eine Definition des Sprachumfangs der Programmiersprache Java ist bereits verfügbar [DM02]. Ein MOmo-Codegenerator könnte anschließend die Umsetzung der Java-Modellelemente in Java-Quelltexte vornehmen.

Anhang A

Aufbau der Konfigurationsdatei

Die Konfigurationsdatei enthält alle Informationen, die zur Steuerung des Transformationsprozesses durch einen MOmo Generator benötigt werden. Im Folgenden ist die vollständige Konfiguration des JMI-Generators für MOF-Modelle angegeben, die verwendet wird, um eine JMI-konforme Implementierung des MOF-Metamodells zu erzeugen. Die einzelnen Elemente der Konfiguration sind kommentiert, um ihre Wirkungsweise kurz zu erläutern.

```
#MOMoC configuration file
#Sun Jun 16 17:21:24 CEST 2002
#Logging
#
log4j.appender.first=org.apache.log4j.ConsoleAppender
log4j.rootLogger=INFO, first
log4j.appender.first.layout.ConversionPattern=%d %-5p %c - %m%n
log4j.appender.first.layout=org.apache.log4j.PatternLayout
#
momoc.debug=true
momoc.debug.directory=debug/cmof20
#
# Quelldatei
#
momoc.source=models/mof2.0/MOF2.cmof
#
momoc.result.directory=tmp/cmof20
momoc.result.directory.prefix=de.unibwm.ist
#
# Reader
#
momoc.reader=de.unibw_muenchen.momoc.reader.CMOFReader
#
# Module
#
momoc.module.expansion\
=de.unibw_muenchen.momoc.modules.PackageMergeExpansionModule
```

```
#
# Schablonenablaufumgebung
#
momoc.contextgenerator=de.unibw_muenchen.momoc.generator.XmlGenerator
momoc.templateexecutant=de.unibw_muenchen.momoc.generator.XmlTransformer
#
# Formatierer
#
momoc.format=true
momoc.formatter.java=de.unibw_muenchen.momoc.formatter.JavaCodeFormatter
momoc.formatter.dtd=de.unibw_muenchen.momoc.formatter.DTDCodeFormatter
momoc.formatter.xml=de.unibw_muenchen.momoc.formatter.XMLCodeFormatter
momoc.formatter.xsd=de.unibw_muenchen.momoc.formatter.XMLCodeFormatter
#
# Welche Ziele sollen erzeugt werden?
#
momoc.generate.package=true
momoc.generate.class=true
momoc.generate.association=true
momoc.generate.enumeration=true
momoc.generate.datatype=true
momoc.generate.model=true
momoc.generate.model.name=jmimof20
#
momoc.artefact.prefix.use=true
momoc.artefact.prefix.case_sensitive=false
momoc.artefact.prefix.delete=org.omg
momoc.artefact.prefix.add=de.unibwm.ist
#
# für jedes Paket
#
momoc.artefact.package.interface=stylesheets/jmi/PackageInterface.xsl
momoc.artefact.package.interface.prefix=Mof
momoc.artefact.package.interface.suffix=Package
momoc.artefact.package.interface.extension=java
#
momoc.artefact.package.implementation\
=stylesheets/jmi/PackageInterfaceImpl.xsl
momoc.artefact.package.implementation.directory=impl
momoc.artefact.package.implementation.prefix=Mof
momoc.artefact.package.implementation.suffix=PackageImpl
momoc.artefact.package.implementation.extension=java
#
# für jede Klasse
#
momoc.artefact.class.proxyinterface=stylesheets/jmi/ClassProxy.xsl
momoc.artefact.class.proxyinterface.prefix=Mof
momoc.artefact.class.proxyinterface.suffix=Class
momoc.artefact.class.proxyinterface.extension=java
#
```

```
momoc.artefact.class.proxyimplementation=stylesheets/jmi/ClassProxyImpl.xsl
momoc.artefact.class.proxyimplementation.directory=impl
momoc.artefact.class.proxyimplementation.prefix=Mof
momoc.artefact.class.proxyimplementation.suffix=ClassImpl
momoc.artefact.class.proxyimplementation.extension=java
#
momoc.artefact.class.interface=stylesheets/jmi/Instance.xsl
momoc.artefact.class.interface.prefix=Mof
momoc.artefact.class.interface.suffix=
momoc.artefact.class.interface.extension=java
#
momoc.artefact.class.implementation=stylesheets/jmi/InstanceImpl.xsl
momoc.artefact.class.implementation.directory=impl
momoc.artefact.class.implementation.prefix=Mof
momoc.artefact.class.implementation.suffix=Impl
momoc.artefact.class.implementation.extension=java
#
# für jede Assoziation
#
momoc.artefact.association.interface=stylesheets/jmi/Association.xsl
momoc.artefact.association.interface.prefix=Mof
momoc.artefact.association.interface.extension=java
#
momoc.artefact.association.implementation\
=stylesheets/jmi/AssociationImpl.xsl
momoc.artefact.association.implementation.directory=impl
momoc.artefact.association.implementation.prefix=Mof
momoc.artefact.association.implementation.suffix=Impl
momoc.artefact.association.implementation.extension=java
#
# für jeden Aufzählungstyp
#
momoc.artefact.enumeration.interface=stylesheets/jmi/EnumerationType.xsl
momoc.artefact.enumeration.interface.prefix=Mof
momoc.artefact.enumeration.interface.extension=java
#
momoc.artefact.enumeration.implementation\
=stylesheets/jmi/EnumerationTypeEnum.xsl
momoc.artefact.enumeration.implementation.prefix=Mof
momoc.artefact.enumeration.implementation.suffix=Enum
momoc.artefact.enumeration.implementation.extension=java
#
# für jeden Datentyp
#
momoc.artefact.datatype.interface=stylesheets/jmi/StructureType.xsl
momoc.artefact.datatype.interface.prefix=Mof
momoc.artefact.datatype.interface.extension=java
#
momoc.artefact.datatype.implementation\
=stylesheets/jmi/StructureTypeImpl.xsl
momoc.artefact.datatype.implementation.directory=impl
momoc.artefact.datatype.implementation.prefix=Mof
```

```
momoc.artefact.datatype.implementation.suffix=Impl
momoc.artefact.datatype.implementation.extension=java
#
# für das Modell
#
# Hauptklasse
#
momoc.artefact.model.main=stylesheets/jmi/Main.xsl
momoc.artefact.model.main.name=Main.java
#
# Reflexion
#
momoc.artefact.model.mof2instance=stylesheets/jmi/MOF-20.xsl
momoc.artefact.model.mof2instance.name=Model.java
momoc.artefact.model.refbaseobjectimpl\
=stylesheets/jmi/reflect/RefBaseObjectImpl.xsl
momoc.artefact.model.refbaseobjectimpl.name\
=jmi/reflect/RefBaseObjectImpl.java
#
# Implementierung der "Property"-Objekte
#
momoc.artefact.model.jmicollection\
=stylesheets/jmi/collections/JmiCollection.xsl
momoc.artefact.model.jmicollection.name=jmi/collections/JmiCollection.java
momoc.artefact.model.jmilist=stylesheets/jmi/collections/JmiList.xsl
momoc.artefact.model.jmilist.name=jmi/collections/JmiList.java
momoc.artefact.model.jmibag=stylesheets/jmi/collections/JmiBag.xsl
momoc.artefact.model.jmibag.name=jmi/collections/JmiBag.java
momoc.artefact.model.jmisequence=stylesheets/jmi/collections/JmiSequence.xsl
momoc.artefact.model.jmisequence.name=jmi/collections/JmiSequence.java
momoc.artefact.model.jmiorderedset\
=stylesheets/jmi/collections/JmiOrderedSet.xsl
momoc.artefact.model.jmiorderedset.name=jmi/collections/JmiOrderedSet.java
momoc.artefact.model.jmiset=stylesheets/jmi/collections/JmiSet.xsl
momoc.artefact.model.jmiset.name=jmi/collections/JmiSet.java
#
# XMI
#
momoc.artefact.model.schema=stylesheets/jmi/xmi/schema/Schema.xsl
momoc.artefact.model.schema.name=xmi/mof20.xsd
momoc.artefact.model.reader=stylesheets/jmi/xmi/document/Reader.xsl
momoc.artefact.model.reader.name=xmi/XmiReaderImpl.java
momoc.artefact.model.writer=stylesheets/jmi/xmi/document/Writer.xsl
momoc.artefact.model.writer.name=xmi/XmiWriterImpl.java
#
# ant - Datei zum Übersetzen des generierten Codes
#
momoc.artefact.model.buildfile=stylesheets/jmi/BuildFile.xsl
momoc.artefact.model.buildfile.name=../../../build.xml
```

Abbildungsverzeichnis

1.1 Automatisierung der Implementierungsphase 2
1.2 Automatische Codegenerierung mit einem MOmo-Codegenerator 4
1.3 Schema eines MOmo-Codegenerators 4

2.1 Sprachbasierter Metamodellbegriff nach Strahringer [Str98] 15
2.2 Dynamische und statische Anteile des Rational Unified Process 18
2.3 Komponenten der MDA (nach [MM01]) 20
2.4 Grundprinzip der MDA (nach [MM01]) 21

3.1 Beispiel einer BOTL-Transformation [BM03b] 31
3.2 Eine Regel in GReAT [KASS03] 32
3.3 Elemente der Transformationssprache UMLX [Wil03] 32
3.4 Ablauf einer schablonenbasierten Codegenerierung 37

4.1 Implementierung eines Metamodells 43
4.2 Aufbau des UML-Werkzeuges ArgoUML 44
4.3 Implementierung eines Codegenerators 45
4.4 Aufgabe eines MOmo-Generators 49
4.5 Prinzipieller Aufbau eines MOmo-Codegenerators 51
4.6 Vollständiger Aufbau eines MOmo-Codegenerators 52
4.7 Aufbau eines MOmo-Codegenerators mit optionalen Komponenten 53
4.8 Definition der *Minimal Component Language* als MOF-Modell 54
4.9 Beispiel in MCL 55
4.10 Ausschnitt eines XMI-Dokumentes für Abb. 4.9 57

4.11 Kontexte für einige Modellelemente aus Abb. 4.10 59
4.12 Schablonen für die Hauptkomponente 59
4.13 Schablone zur Erzeugung des Komponentenrumpfes 60
4.14 Erzeugter Code für die Komponente **Waschmaschine** 61
4.15 Erzeugter Code für die Komponente **Steuerung** 62

5.1 Die OMG-Metadatenarchitektur 66
5.2 Beziehungen zwischen MOF und UML 68
5.3 Struktur der UML-Infrastrukturbibliothek 70
5.4 Beziehungen zwischen Paketen der UML-Infrastruktur und Paketen der MOF . 71
5.5 Klassendefinition im Paket **Constructs** der UML-Infrastruktur 72
5.6 Notation einer MOF-Klasse 73
5.7 Notation einer Vererbung 77
5.8 Repräsentation von Assoziationen im Metamodell der UML-Infrastruktur . . . 78
5.9 Grundlegende Notation von Assoziationen 80
5.10 Repräsentation von Datentypen im Metamodell der UML-Infrastruktur 81
5.11 Namensräume im Metamodell der UML-Infrastruktur 83
5.12 Paketrepräsentation im Metamodell der CMOF 84
5.13 Notation einer Importbeziehung 85
5.14 Namenskonflikt zwischen enthaltenem und importiertem Modellelement 86
5.15 Namenskonflikt zwischen sichtbarem und importiertem Modellelement 86
5.16 Notation für die Vereinigung von Paketen 87
5.17 Umsetzung geschachtelter Klassen in Vereinigungsbeziehungen 88
5.18 Umsetzung geschachtelter Pakete bei Vereinigungsbeziehungen 88
5.19 Elemente zur Definition von Bedingungen 90

6.1 Aufbau eines XML-Elementes 94
6.2 Schachtelung von XML-Elementen 95
6.3 Repräsentation von Attributen in XML 95
6.4 Verwendung von Sonderzeichen in einem Textelement 96
6.5 Verwendung von Namensräumen in XML-Dokumenten 98
6.6 Deklaration eines Namenraumes in einem XML-Element 98
6.7 Benennung von XML-Attributen (aus [Nam99]) 99
6.8 Referenz eines Elementes innerhalb eines XML-Dokumentes 100
6.9 Referenz mehrerer Elemente innerhalb eines XML-Dokumentes 100
6.10 Einfache Verknüpfung nach XLink-Standard 101

6.11 Erweiterte Verknüpfung nach XLink-Standard 102
6.12 Aufbau eines XML-Schemas 104
6.13 Einschränkungen von Attributen und Elementen 105
6.14 Vorgabewerte und Konstanten 106
6.15 Typdefinitionen in XML-Schema (aus [XML01c]) 107
6.16 Einschränkung des Wertebereichs eines Zahlentyps 108
6.17 Einschränkung des Wertebereichs einer Zeichenkette 108
6.18 Aufbau eines komplexen Datentyps 109
6.19 Listen und Vereinigungen 110

7.1 Beziehungen zwischen MOF und XML 115
7.2 Aufbau eines XMI-Schemas 117
7.3 Repräsentation des Paketes **MCL** in XMI-Schema 118
7.4 Elementdeklaration für die Klasse **Component** 119
7.5 Abbildung einer Vererbungsbeziehung durch Kopieren 119
7.6 Vererbung durch XML-Schema-Mechanismus 120
7.7 Repräsentation der Klasse **Component** in der Grundeinstellung 121
7.8 Vollständiger Typ der Klasse **Component** mit XML-Schema-Vererbung 122
7.9 Definition einer Assoziation 123
7.10 Definition des Aufzählungstyps **PackageMergeKind** 124
7.11 Aufbau eines XMI-Dokumentes 125
7.12 Wurzelknoten des Beispieldokumentes 126
7.13 Repräsentation von Eigenschaften durch XML-Attribute 127
7.14 Repräsentation von Eigenschaften durch XML-Elemente 128

8.1 Bestandteile einer MOmo-Implementierung 133
8.2 Erzeugung von Schnittstellen für Pakete 135
8.3 Erzeugung von Schnittstellen für Klassen 137
8.4 Erzeugung von Schnittstellen für Objekte 138
8.5 Abbildung einer MOF-Klassenhierarchie auf Java 140
8.6 Implementierung einer Mehrfachvererbung durch Delegation 141
8.7 Implementierung einer Mehrfachvererbung durch Kopieren 142
8.8 Namenskonflikte zwischen Attributen 145
8.9 Klassen zur Implementierung von Attributen in MOmo 147
8.10 Auswirkung einer Redefinition auf die Instanzebene 149
8.11 Auswirkung einer Teilmengenbeziehung auf die Instanzebene 151

8.12 Erzeugung von Assoziationen . . . 152
8.13 Schnittstellen für strukturierte Datentypen . . . 154
8.14 Schnittstellen für Instanzen strukturierter Datentypen . . . 154
8.15 Aufzählungstypen in JMI . . . 156
8.16 Reflexive Basisschnittstellen . . . 157
8.17 JMI-Schnittstellen zu XMI . . . 158

9.1 Minimale Architektur eines MOmo-Generators . . . 162
9.2 MOmo-konforme Reader-Schnittstelle . . . 164
9.3 MOmo-konforme Modul-Schnittstelle . . . 164
9.4 MOmo-konforme Kontextgenerator-Schnittstelle . . . 165
9.5 MOmo-konforme Schablonenausführer-Schnittstelle . . . 165
9.6 MOmo-konforme Formatierer-Schnittstelle . . . 165
9.7 Velocity-Schablone zur Erzeugung einer JMI-Schnittstelle . . . 167
9.8 XSLT-Schablone zur Erzeugung einer JMI-Schnittstelle . . . 171
9.9 Die Aufgabe des XML-Generators . . . 174
9.10 Kontexteerzeugung für eine JMI-Klasse . . . 176
9.11 Erzeugung der XML-Repräsentation . . . 176
9.12 Der MOmo-Schablonenausführer . . . 177
9.13 Beispiel für einen Konfigurationsblock des Schablonenausführers . . . 178
9.14 Erzeugung der Artefakte . . . 179

10.1 Elemente eines Stellen-Transitionen-Netzes . . . 184
10.2 Definition von Stellen-Transitionen-Netzen . . . 186
10.3 XMI-Darstellung des Metamodells aus Abb. 10.2 . . . 188
10.4 Generierte Implementierung des MOF-Modells . . . 189
10.5 Implementierung einer Stelle . . . 190
10.6 Implementierung einer Transition . . . 192
10.7 Beispiele für Kontexte . . . 193
10.8 Schablone zur Erzeugung einer Steuerung . . . 195
10.9 Schablone zur Erzeugung der Verbindungen einer Transition . . . 196
10.10 Schablone zur Deklaration und Initialisierung einer Stelle . . . 197
10.11 Stellen-Transitionen-Netz zur Steuerung der Waschmaschine . . . 198
10.12 XMI-Dokument für das Beispielnetz . . . 200
10.13 Ausschnitt des generierten Quelltextes der Waschmaschinensteuerung . . . 201
10.14 MOmo-MOF-Codegenerator . . . 203

Tabellenverzeichnis

2.1 Klassifizierung von Modellen nach Lorenz [Lor04] 11

3.1 Eigenschaften von Ansätzen zur Codegenerierung aus Modellen 38

7.1 Eigenschaften der verschiedenen XMI-Versionen 114

8.1 Vor- und Nachteile von Abbildungsmöglichkeiten 143
8.2 Abbildung der Attributarten (nach [UP03a]) 148
8.3 Abbildung einfacher Datentypen 154

9.1 Bewertung der Schablonenmechanismen 173

10.1 Laufzeitverhalten des MOmo-MOF-Codegenerators 204

Literaturverzeichnis

[ABS04] Carsten Amelunxen, Lutz Bichler und Andy Schürr. Codegenerierung für Assoziationen in MOF 2.0. In *Proceedings of the Modellierung 2004*. Marburg, Deutschland, March 2004.

[ACC+03] Adaptive Ltd, Ceira Technologies Inc., Compuware Corporation, Data Access Technologies Inc., DSTC, Gentleware, Hewlett-Packard, International Business Machines, IONA Technologies, MetaMatrix, Rational Software, Softeam, Sun Microssystems, Telelogic AB, Unisys und WebGain. *Meta Object Facility (MOF) 2.0 Core Proposal*. Object Management Group, April 2003. Ftp://ftp.omg.org/pub/docs/ad/030407.pdf.

[AKP03] David Akehurst, Stuart Kent und Octavian Patrascoiu. A Relational Approach to Defining and Implementing Transformations in Metamodels. *Software and Systems Modeling*, 2(4):215–239, December 2003. Http://www.cs.kent.ac.uk/pubs/2003/1764.

[AKZ96] Maher Awad, Juha Kuusela und Jürgen Ziegler. *Object-Oriented Technology for Real-Time Systems*. Prentice Hall, London, 1996. ISBN 0-13-227943-6.

[And] AndroMDA. Http://www.andromda.org.

[Arg] ArgoUML. Http://www.argouml.org.

[AST+03] Alcatel, Softeam, Thales, TNI-Valiosys und Codagen Technologies Corp. *Response to the MOF 2.0 Query/Views/Transformations RFP*, August 2003. Ftp://ftp.omg.org/pub/docs/ad/03-08-05.pdf.

[Bal99] Heide Balzert. *Lehrbuch der Objektmodellierung*. Spektrum Akademischer Verlag, Hedelberg, Berlin, 1999. ISBN 3-8274-0285-9.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette und Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *Online Proceedings of the OOPSLA´03 Workshop on Generative Techniques in the Context of the MDA*. Oktober 2003. Http://www.softmetaware.com/oopsla2003/bezivin.pdf.

[BLFM98] T. Berners-Lee, R. Fielding und L. Masinter. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*. IETF (International Engineering Task Force), August 1998.

[BLV03] L. Bettini, M. Loreti und B. Venneri. On Multiple Inheritance in Java. In *Proceedings of TOOLS Eastern Europe 2002*, Seiten 1–15. Kluwer Academic Publishers, 2003.

[BM03a] Peter Braun und Frank Marschall. BOTL - The Bidirectional Object Oriented Transformation Language. Technischer Bericht TUM-I0307, Technische Universität München, München, Mai 2003.

[BM03b] Peter Braun und Frank Marschall. Transforming Object Oriented Models with BOTL. In Paolo Bottoni und Mark Minas (Herausgeber), *Electronic Notes in Theoretical Computer Science*, Band 72. Elsevier, 2003.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design*. Benjamin Cummings Series in Object-Oriented Software Engineering. Benjamin Cummings, Redwood City, CA, 1994.

[Boo03] Paul Boocock. *Jamda Model Compiler Framework*. The Jamda Project, 0. Auflage, 2003. Http://jamda.sourceforge.net/docs/index.html.

[Bro98] Brockhaus (Herausgeber). *Die Enzyklopädie*, Band 20. Brockhaus, Leipzig, Mannheim, 1998. ISBN 3-7653-3100-7.

[BRS01] Lutz Bichler, Ansgar Radermacher und Andy Schürr. Combining Data Flow Equations with UML/Realtime. In Edgar Nett, Doug Jensen und Makoto Takizawa (Herausgeber), *Proceedings of the The Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE, IEEE Computer Society, Magdeburg, Deutschland, Mai 2001.

[BRS04] Lutz Bichler, Ansgar Radermacher und Andy Schürr. Integrating Data Flow Equations with UML/Realtime. *Real-Time Systems*, 26(1):107–125, January 2004.

[Bun97] Bundesamt für Wehrtechnik und Beschaffung. *Entwicklungsstandard für IT-Systeme des Bundes, Vorgehensmodell*, Juni 1997. Allgemeiner Umdruck 250.

[CAB94] Derek Coleman, Patrick Arnold und Stephanie Bodoff. *Object-Oriented Development: The Fusion Method*. Prentice Hall, London, 1994.

[CDI+97] Stephen Crawley, Scott Davis, Jaga Indulska, Simon McBride und Kerry Raymond. Meta-meta is better-better! In Hartmut König, Kurt Geihs und Thomas Preuß (Herausgeber), *Proceedings of the IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*. Chapman & Hall, Cottbus, Germany, September 1997. ISBN 0-412-82340-3.

[CEK+00] Tony Clark, Andy Evans, Stuart Kent, Steve Brodsky und Steve Cook. A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach. Technischer Bericht, pUML Group, September 2000. Http://www.cs.york.ac.uk/puml/mmf/mmf.pdf.

[CH03] Krzysztof Czarnecki und Simon Helsen. Classification of Model Transformation Approaches. 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, Oktober 2003.

[CHM+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza und Dániel Varró. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE'02)*, Seiten 267–276. Edinburgh, UK, September 2002.

[Cla99a] James Clark. *Associating Style Sheets with XML documents*, Juni 1999. W3C Recommendation, http://www.w3.org/TR/xml-stylesheet.

[Cla99b] James Clark. *XSL Transformations (XSLT) Version 1.0*. World Wide Web Committee (W3C), November 1999. W3C Recommendation, http://www.w3.org/TR/xslt.

[CTC] Codagen Technologies Corp., *Codagen Architect*. Http://www.codagen.com/products/architect/default.htm.

[DeM03] Linda G. DeMichiel. *Enterprise JavaBeans™Specification*. SUN Microsystems, Santa Clara, USA, zweite Auflage, November 2003.

[Dir02] Ravi Dirckze. *Java™ Metadata Interface (JMI) Specification, Version 1.0*. Unisys, erste Auflage, Juni 2002.

[DM02] Svata Dedic und Martin Matula. Metamodel for the Java language, 2002. Http://java.netbeans.org/models/java/java-model.html.

[DMC03] DSTC, International Business Machines und CBOP. *MOF Query/-Views/Transformations*, first revised submission Auflage, August 2003. Ftp://ftp.omg.org/pub/docs/ad/03-08-03.pdf.

[DST02] DSTC. dMOF 1.1 — An OMG Meta Object Facility Implementation, Mai 2002. Http://www.dstc.edu.au/Products/CORBA/MOF/.

[DW99] D.F. D'Souza und A.C. Wills. *Objects, Components, and Frameworks with UML - The Catalysis Approach*. Addison Wesley, 1999.

[FIT04] Fraunhofer Institute FOKUS, IKV++ Technologies AG und Technical University Berlin. *MOF 2.0 IDL RFP*, zweite Auflage, Januar 2004. OMG document ad/2004-01-09.

[FUJ] Arbeitsgruppe Softwaretechnik der Universität Paderborn, *FUJABA Tool Suite*. http://www.fujaba.de.

[GdCL03] Gonzalo Génova, Carlos Ruiz del Castillo und Juan Llorens. Mapping UML Associations into Java Code. *Journal of Object Technology*, 2(5):135–162, September-October 2003. Http://www.jot.fm/issues/issue_2003_09/article4.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns*. Addison-Wesley, September 1994. ISBN 0-201-63361-2.

[GJSB00] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *The Java™ Language Specification*. The Java™Series. Addison Wesley, Reading, Massachusetts, zweite Auflage, Juni 2000.

[GLR+02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel und Andrew Wood. Transformation: The Missing Link of MDA. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski und Grzegorz Rozenberg (Herausgeber), *Graph Transformation, First International Conference, ICGT 2002*, Band 2505 von *Lecture Notes in Computer Science*, Seiten 90–105. Springer, Barcelona, Spain, 2002. ISBN 3-540-44310-X.

[HBR00] William Harrison, Charles Barton und Mukund Raghavchari. Mapping UML Designs to Java. In *Proceedings of the 15th Annual ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'2000)*, Band 35, Seiten 178–187. ACM SIGPLAN Notices, ACM Press, Minneapolis, Minnesota, USA, Oktober 2000.

[HBvB+94] Wolfgang Hesse, Georg Barkow, Hubert von Braun, Hans-Bernd Kittlaus und Gert Scheschonk. Terminologie der Softwaretechnik, Ein Begriffssystem für die Analyse und Modellierung von Anwendungssystemen, Teil 1: Begriffssystematik und Grundbegriffe. *Informatik Spektrum*, 17(1):39–47–135, April 1994.

[Hes00] Wolfgang Hesse. Software-Projektmanagement braucht klare Strukturen - Kritische Anmerkungen zum R̈ational Unified Process.̈ In J. Ebert und U. Frank (Herausgeber), *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Proc. "Modellierung 2000*, Seiten 143–150. Fölbach-Verlag, Koblenz, 2000.

[HS98] Brian Henderson-Sellars. Clarifying Specialized Forms of Association in UML and OML. *Journal of Object-Oriented Programming*, 11(2):47–54, Februar 1998.

[IBM] IBM/Rational. Extensible Development Environment (XDE). Http://www-306.ibm.com/software/awdtools/developer/rosexde/.

[Int86] International Organization or Standardization. *Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*, 1986. ISO 8879:1986(E).

[Jac94] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, MA, vierte Auflage, 1994.

[Jak] Das Apache Jakarta Projekt. Http://jakarta.apache.org.

[JBR99] Ivar Jacobson, Grady Booch und James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999. ISBN 0-201-57169-2.

[JDC] Sun Microsystems, *Javadoc*. Ttp://java.sun.com/j2se/javadoc/.

[Jec00] Mario Jeckle. Konzepte der Metamodellierung — Zum Begriff Metamodell. *Softwaretechnik Trends*, 20(2), Mai 2000.

[Jen97] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag, Berlin, zweite Auflage, 1997. ISBN 3-540-60943-[1|2|3].

[JRY04] Das JRuby Projekt, *JRuby 0.7*. http://jruby.sourceforge.net, March 2004.

[KASS03] Gabor Karsai, Aditya Agrawal, Feng Shi und Jonathan Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9(11), 2003. Http://www.jucs.org/jucs_9_11/on_the_use_of.

[KLW95] Michael Kifer, Georg Lausen und James Wu. Logical Foundations of Object Oriented and Frame Based Languages. *Journal of the ACM*, 42(4):741–843, Juli 1995.

[Kru98] Philippe Kruchten. *The Rational Unfied Process: an introduction*. Object Technologie. Addison Wesley, 1998. ISBN 0-201-60459-0.

[Lor04] Peter Lorenz. Simulation & Animation. http://isgsim1.cs.uni-magdeburg.de/pelo/sa/sim1.php, Mai 2004. Vorlesungsskript.

[Mat03] Martin Matula. *NetBeans Metadata Repository*. SUN Microsystems, März 2003.

[MB03] Frank Marschall und Peter Braun. Model Transformations for the MDA with BOTL. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, Nummer 03-37 in CTIT, Seiten 25–36. University of Twente, Enschede, Juni 2003.

[Met02] What is metamodeling and what is it good for?, Oktober 2002. Http://www.metamodel.com/staticpages/index.php?page=20021010231056977.

[Mey76] Meyer (Herausgeber). *Meyers Enzyklopädisches Lexikon in 25 Bänden*, Band 9. Bibliographisches Institut, Mannheim, Wien, 1976.

[MM01] Joaquin Miller und Jishnu Mukerji. *Model Driven Architecture*. Object Management Group, Juli 2001. Document number ormsc/2001-07-01.

[MM03] Joaquin Miller und Jishnu Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, Juni 2003. Document number omg/2003-06-01.

[Nag96] Manfred Nagl (Herausgeber). *Building tightly integrated software development environments: the IPSEN approach*. Nummer 1170 in LNCS. Springer-Verlag, 1996.

[Nam99] W3C. *Namespaces in XML*, Januar 1999. W3C Recommendation, http://www.w3.org/TR/REC-xml-names-19990114.

[NET] Microsoft Corporation, *.NET*.

[Nob96] James Noble. Some Patterns for Relationships. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific 21)*. Prentice Hall, Melbourne, Australien, 1996.

[Nov01] Dimitre Novatchev. The Functional Programming Language XSLT - A proof through examples. http://fxsl.sourceforge.net/articles/FuncProg/Functional Programming.html, November 2001.

[NSU] Novosoft, *Novosoft UML Library*. Http://nsuml.sourceforge.net.

[Obj] Interactive Objects. ArcStyler Modeling Style and User Guide. Http://www.io-software.com/as_support/docu/Users_Guide.pdf.

[Obj00a] Object Management Group. *XML Metadata Interchange, Version 1.0*, Juni 2000. OMG document formal/2000-06-01.

[Obj00b] Object Management Group. *XML Metadata Interchange (XMI) Specification, Version 1.1*, November 2000. OMG document formal/2000-11-02.

[Obj02a] Object Management Group. *Common Object Request Broker Architecture: Core Specification, Version 3.0.2*, Dezember 2002. Formal/2002-12-02.

[Obj02b] Object Management Group. *Meta Object Facility (MOF), Version 1.4*, April 2002. Ftp://ftp.omg.org/pub/docs/formal/2002-04-03.

[Obj02c] Object Management Group. *OMG XML Metadata Interchange (XMI) Specification, Version 1.2*, Januar 2002. OMG document formal/2002-01-01.

[Obj03a] Object Management Group. *Common Warehouse Metamodel, Version 1.2*, März 2003. OMG document formal/2003-03-02.

[Obj03b] Object Management Group. *Meta Object Facility (MOF) 2.0 XMI Mapping*, April 2003. OMG document ad/2003-04-04.

[Obj03c] Object Management Group. *XML Metadata Interchange (XMI) Specification, Version 2.0*, Mai 2003. OMG document formal/2003-05-03.

[OCL03] Object Management Group. *UML 2.0 OCL Specification*, Oktober 2003.

[OpJ] Compuware, *OptimalJ*. Http://www.compuware.com/products/optimalj/.

[PBG01] Mikael Peltier, Jean Bézivin und Gabriel Guillaume. MTRANS: A general framework, based on XSLT, for model transformations. In *Proceedings of the Workshop on Transformations in UML*. Genova, Italien, April 2001.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. Dissertation, Institut für Mathematik, Universität Bonn, 1962.

[PG03] Slava Pestov und John Gellene. jEdit 4.1 User's Guide, 2003. Http://www.jedit.org/users-guide.

[Plo02] Constantine Plotnikov. Novosoft UML Library for Java, Juli 2002. Http://nsuml.sourceforge.net.

[PR02] Samuele Pedroni und Noel Rappin. *Jython Essentials*. O'Reilly, March 2002. ISBN 0-596-00247-5.

[Pru97] Peeter Pruuden. *Inheritance from the standpoint of specification and modeling*. Diplomarbeit, Tampere University of Technology, August 1997.

[PSV98] Klaus Pohl, Andy Schürr und Gottfried Vossen (Herausgeber). *Modellierung '98, Proceedings des GI-Workshops in Münster, 11.-13. März 1998*, Band 9 von *CEUR Workshop Proceedings*, 1998.

[PZ87] Louchka Popova-Zeugmann. Zeit-Petri-Netze. In *Beiträge zur Theorie und Anwendung von Petri-Netzen*, Band 8 von *Wissenschaftliche Schriften der Technischen Universität*, Seiten 37–47. Karl-Marx-Stadt, DDR, 1987.

[QP03] QVT-Partners. *Revised submission for MOF 2.0 Query/Views/Transformations RFP*, erste Auflage, August 2003. Ftp://ftp.omg.org/pub/docs/ad/03-08-08.pdf.

[RA01] Gianna Reggio und Egidio Astesiano. A proposal of a dynamic core for UML metamodelling with MML. Technischer Bericht, DISI Universitiá di Genova, April 2001.

[RBEL91] James Rumbaugh, Michael Blaha, William Premerlani Frederick Eddy und William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rum87] James Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*, Seiten 466–481. ACM Press, Orlando, Florida, USA, Oktober 1987.

[Sch81] Friedemann Schulz von Thun. *Miteinander reden 1*. Rowohlt, Reinbek bei Hamburg, 1981.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In G. Tinhofer (Herausgeber), *WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, Nummer 903 in LNCS, Seiten 151–163. Springer-Verlag, Herrsching, Juni 1994.

[Sch99] Peter Schefe. Softwaretechnik und Erkenntnistheorie. *Informatik Spektrum*, 22(2):122–135, April 1999.

[SGW94] Bran Selic, Garth Gullekson und Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley, New York, 1994.

[SHC95] Zoltan Somogyi, Fergus Henderson und Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, Seiten 499–512. Glenelg, Australia, February 1995.

[SK03] Shane Sendall und Wojtek Kozaczynski. Model Transformation — the Heart and Soul of Model-Driven Software Development. Technischer Bericht, École Polytechnique Fédérale de Lausanne, Juli 2003. Http://icwww.epfl.ch/publications/abstract.php?ID=200352.

[SPGB03] Shane Sendall, Gilles Perrouin, Nicolas Guelfi und Olivier Biberstein. Supporting Model-to-Model Transformations: The VMT Approach. Technischer Bericht, École Polytechnique Fédérale de Lausanne, Lausanne, Juli 2003.

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien, 1973.

[Ste01] Perdita Stevens. On associations in the Unified Modeling Language. In M. Gogolla und C. Kobryn (Herausgeber), *UML 2001 - The Unified Modeling Language.Modeling Languages, Concepts, and Tools: 4th International Conference*, Seiten 361–375. Springer Verlag, Toronto, Canada, October 2001.

[Str98] Susanne Strahringer. Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips. In Klaus Pohl, Andy Schürr und Gottfried Vossen (Herausgeber), *Modellierung '98, Proceedings des*

GI-Workshops in Münster, Band 9 von *CEUR Workshop Proceedings*. Gesellschaft für Informatik, 1998.

[SWZ95] Andy Schürr, Andreas Winter und Albert Zündorf. Graph Grammar Engineering with PROGRES. In W. Schäfer und P. Botella (Herausgeber), *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, Seiten 219–234. Springer-Verlag, London, UK, 1995.

[TH00] David Thomas und Andrew Hunt. *Programming Ruby*. Addison-Wesley, 2000. ISBN 0-201-71089-7.

[Tho01] Marco Thomas. Die Vielfalt der Modelle in der Informatik. In Reinhard Keil-Slawik und Johannes Magenheim (Herausgeber), *Informatikunterricht und Medienbildung*, Band 8 von *LNI*, Seiten 173–186. GI, 2001. ISBN 3-88579-334-2.

[UP03a] U2-Partners. *Unified Modeling Language: Infrastructure, Version 2.0*, März 2003. OMG document ad/2003-03-01.

[UP03b] U2-Partners. *Unified Modeling Language: Superstructure, Version 2.0*, April 2003. OMG document ad/2003-04-01.

[vEB] Ghica van Emde Boas. FUUT — Fantastic, Unique, UML Tool for the Java(tm) Environment. Http://www.bronstee.com/index.php?id=FUUT-je.

[Vel] Apache Jakarta Projekt, *Velocity 1.4*. Http://jakarta.apache.org/velocity/index.html.

[Voe] Markus Voelter. Open ArchitectureWare . Http://sourceforge.net/projects/architecturware/.

[Voe03] Markus Voelter. A Catalog of Patterns for Program Generation. In *EuroPLop*. Juni 2003. Http://hillside.net/europlop/europlop2003/papers/WorkshopB/B6_VoelterM.pdf.

[vRD03] Guido van Rossum und Fred L. Drake. *Python Reference Manual*. PythonLabs, December 2003. Release 2.3.3.

[VTB98] John Viega, Bill Tutt und Reimer Behrends. Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages. Technischer Bericht CS-98-03, 2, 1998.

[W3C] *World Wide Web Committee*. Http://www.w3.org.

[WCO00] Larry Wall, Tom Christiansen und Jon Orwant. *Programming Perl*. O'Reilly, dritte Auflage, July 2000. ISBN 0-596-00027-8.

[Wil03] Edward Willink. UMLX: A graphical transformation language for MDA. In A.Rensink (Herausgeber), *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, Seiten 13–24. University of Twente, CTIT, Enschede, Niederlande, June 2003. Http://trese.cs.utwente.nl/mdafa2003.

[Wis01] Wissenschaftlicher Rat der Dudenredaktion (Herausgeber). *Duden Fremdwörterbuch*. Nummer Band 5 in Der Duden;. Dudenverlag, Mannheim, 2001. ISBN 3-411-04057-2.

[WR03] Craig Walls und Norman Richards. *XDoclet in Action*. Manning Publication Co., Greenwich, Dezember 2003. ISBN 1932394052.

[XDT] XDoclet Team, *XDoclet*. Http://xdoclet.sourceforge.net.

[XLi01] W3C. *XML Linking Language (XLink) Version 1.0*, Juni 2001. W3C Recommendation, http://www.w3.org/TR/xlink.

[XML00] W3C. *Extensible Markup Language (Second Version)*, Oktober 2000. W3C Recommendation, http://www.w3.org/TR/REC-xml.

[XML01a] W3C. *XML Schema Part 0: Primer*, Mai 2001. W3C Recommendation, http://www.w3c.org/TR/xmlschema-0.

[XML01b] W3C. *XML Schema Part 1: Structures*, Mai 2001. W3C Recommendation, http://www.w3c.org/TR/xmlschema-1.

[XML01c] W3C. *XML Schema Part 2: Datatypes*, Mai 2001. W3C Recommendation, http://www.w3c.org/TR/xmlschema-2.

[XPo01] W3C. *XML Pointer Language (XPointer) Version 1.0*, Januar 2001. W3C Last Call Working Draft, http://www.w3.org/TR/2001/WD-xptr-20010108.

Abkürzungsverzeichnis

ATL	Atlas Transformation Language
BOTL	Bidirectional Object Oriented Transformation Language
CORBA	Common Object Request Broker Architecture
CWM	Common Warehouse Metamodel
GReAT	Gaph Rewriting and Transformation Language
IPSEN	Integrated Software Project Support Environment
JDK	Java Development Kit
JMI	Java Metadata Interface
MDA	Model Driven Architecture
MDR	Meta Data Repository
MOF	Meta Object Facility
MOmo	MOF Metamodellierung
OCL	Object Constraint Language
PIM	Plattform Independent Model
PROGRES	Programmed Graph Rewriting System
PSM	Plattform Specific Model
QVT	Query Views Transformations
RUP	Rational Unified Process
TGG	Triple Graph Grammar
TRL	Transformation Rule Language
UML	Unified Modeling Language
VIATRA	Visual Automated Model Transformations
VMT	Visual Model Transformation
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

Stichwortverzeichnis

Abstraktion, 7, 9
Abstraktionsgrad, 25
Abstraktionsniveau, 3, 23
Analysephase, 1
AndroMDA, 36, 166
Anforderungsdefinition, 42
Anwendung
    Struktur, 25
    Verhalten, 25
Anwendung des MOmo-Baukastens, 183
Anwendungen, 183
Anwendungsfall, 42
    Codegenerator, 44
    Metamodell, 42
Anwendungsreihenfolge, 29
ArchitectureWare, 37
ArcStyler, 38
ArgoUML, 31, 44, 166
Artefakt, 2, 3, 5, 6, 19, 26, 41, 46, 49, 161
Assoziation, 78, 123, 151
    explizite Darstellung, 151
    gerichtete, 140
    implizite Darstellung, 151
Assoziationsende, 78, 132, 149
ATL, 34
Attribut, 73, 93, 95, 100, 119, 132, 137
    einwertig, 138
    geordnet, 146
    mehrwertig, 139
    Name, 73
    optional, 138
    Typ, 73
    ungeordnet, 146
Attribute, 144
Auslösemechanismus, 75
Automatisierung, 2

Basic, 38
Basisdokument, 125
Bedingungen, 89
BOTL, 30

C++, 38, 136, 144
C#, 38
Catalysis, 16
CDATA, 96
Clustering, 134
CMOF, 70, 78
Codagen Architect, 38
Codegenerator, 3, 5, 131
Codegenerierung, 3, 25, 26
    Definition, 27
    direkte Programmierung, 36
    schablonenbasiert, 36
CORBA, 19
CWM, 20, 34, 67

Datenaustausch, 3
Datentyp, 80, 124, 153
    Aufzählungs-, 81, 124, 154
    einfacher, 81, 124, 153
    strukturierter, 81, 124, 153

Delegat, 143
Delegation, 140, 143
Delgations-Idiom, 143
Diagrammeditor, 2
dMOF, 29
Dokument, 124
Dokumentenformat, 6
Dokumentklasse, 102
dotNET, 19
DTD, 93

EBNF, 116
Editor, 3
Einfachvererbung, 139
EJB, 19, 37
Element, 93, 95, 119
Elementdeklaration, 103, 104, 119
Elementinhalt, 94
EMOF, 69, 78
Entität, 93, 96
Entwurfsphase, 1
Evaluation, 204
Export-Schnittstelle, 113

F-Logic, 30
Flussrelation, 184
Formatierer, 163
Fusion, 16
FUUT-je, 37

Generics, 149
Generierungsprozess, 162
Gewicht, 185
Grammatik, 12, 26
GReAT, 31

Hardware, 1

ID, 100
IDREF, 100
IDREFS, 100
Implementierung, 2
Implementierungsphase, 3
Implementierungstechnologie, 2, 3, 5, 20
Implementierungsvererbung, 139
Import-Schnittstelle, 113
Instanzdokument, 103
Instanziierung, 3
ist-ein Beziehung, 77

Jakarta Projekt, 166
Jamda, 29
Java, 6, 37, 38, 47, 131, 143
    Metamodell, 41
    Paket, 47
JDK, 146
JMI, 46, 131

Kennzeichnung
    beendende, 94
    einleitende, 94
Kindelement, 96
Klasse, 71, 118, 136
Klassenbibliothek, 82
Knoten, 94
Komponente
    generiert, 45
    generierte, 4, 131
    generisch, 45
    generische, 4, 161
    modellabhängige, 132, 133
    modellunabhängige, 132, 155
Komposition, 80
Konfiguration, 19
Konfigurationsblock, 178
Konfigurationsvariable, 117, 119, 120, 122, 163, 178
Kontext, 51, 58, 162, 174
Kontextgenerator, 51, 162, 174
Kontrollflusssprache, 31
Kopieren, 140

Laufzeit, 135, 144
Laufzeiteffizienz, 141, 143
Laufzeitfehler, 146
Laufzeitverhalten, 204
Lebenszyklus, 74

Marke, 184

Erzeugen einer, 185
Verbrauchen einer, 185
Markierung, 184
Markierungszeichen, 94
MCL, 54, 117, 126, 128
MDA, 1, 5, 20, 25
Mehrfachvererbung, 120, 137, 139
Mercury, 30
Meta, 12
Meta-Ebene, 12
Meta-Kommunikation, 13
Metadatenarchitektur, 14, 66
Metamodell, 3, 5, 7, 12, 14, 65, 131
Definition, 15
Java, 41
prozessbasiertes, 16
sprachbasiertes, 15
Metaobjekt, 50
Modell, 1, 3, 5, 7
Analyse-, 25
Anwendungsbereiche, 8
Begriffsdefinition, 10
Design-, 25
Eigenschaften, 10
Klassifizierung, 8
mittelbares, 15
plattformspezifisch, 3, 21, 25
plattformspezifisches, 2
plattformunabhängig, 21, 25
plattformunabhängiges, 2
unmittelbares, 15
Modellierung, 16
Modellierungssprache, 3, 4, 25
domänenspezifisch, 2, 41
MOF-basiert, 4, 49
Syntax, 3
Vokabular, 12
Modelltransformation, 2, 3, 25, 26
Definition, 27
deklarative Ansätze, 29
direkte Programmierung, 28
graphtransformations-basierte, 30
hybride Ansätze, 33
Modelltransformator, 5
Modul, 52, 162
MOF, 4–6, 20, 42, 48, 54, 65, 114, 202
OVT, 29
Paket, 53
Vereinigung, 53
MOmo, 3
Baukasten, 4, 5, 25, 41, 65
Codegenerator, 3, 4, 48
MTRANS, 34
Multiplizität, 74, 76, 79, 119, 146
Muster, 31
Factory, 136, 137
Singleton, 136
MVC, 44

Nachbedingung, 76
Namenskonflikt, 145
Namensraum, 83, 94, 97, 116
NetBeans MDR, 29, 134
nsmdr, 29
nsuml, 38

Objectory, 16
OCL, 33, 90
OCTOPUS, 16
OMG, 65, 113
OMT, 16
OOAD, 16
Operation, 75, 137
Operationen, 144
Optima-J, 37
Original, 8

Paket, 82, 118, 134
Import, 85
Schachtelung, 85
Vereinigung, 86
Paketinstanz, 134
Parameter, 76
Perl, 170
Petri-Netz, 3, 6, 183
PIM, 21
Produktion, 126

Produktionsregel, 126
Produktqualität, 19
Programmiersprache, 2, 5, 26
Prototyp, 7
prototypisch, 3
Prozess, 7
    iterativ, 17
    linear, 17
Prozessqualität, 19
PSM, 21
Python, 170

Quellmodell, 26
Quelltext, 2–5, 41, 49

Redefinitionsbeziehung, 75, 149
Referenz, 99
Reflexion, 155
Relation, 29
Reverse Engineering, 22
ROOM, 16
Ruby, 170
RUP, 5, 17
Rückgabewert, 139

Schablone, 4, 50, 135
Schablonenablaufumgebung, 5, 6, 51, 166
Schablonenausführer, 51, 163, 177
Schablonenmechanismus, 162, 166
Schachtelung, 95
Schaltregel, 185
Schnittstelle, 131, 132
Schnittstellenvererbung, 139
Semantik, 13, 29, 77, 131
SGML, 93
Skriptsprache, 170
Software, 1
Softwareentwicklung, 3
Softwareentwicklungsprozess, 1
Softwaresystem, 1, 25
Speicherverbrauch, 141
Stelle, 184
Stellen-Transitionen-Netz, 184
Stereotyp, 29, 41, 70
Steuerkomponente, 6, 162, 163
Stylesheet, 169
Superklasse, 139
Syntax
    abstrakte, 13, 29, 65
    konkret, 13

Teilmengenbeziehung, 75, 150
Terminierungssemantik, 29
Textelement, 96
TLR, 33
Transformation, 4
    Modell-zu-Modell, 27
    Modell-zu-Text, 27
Transition, 184
Typ, 120, 139
Typdefinition, 103, 106, 119
typkompatibel, 139
Typkompatibilität, 75

Ubersetzungszeit, 144
UML, 2–5, 16, 20, 41, 48, 65, 67
    Infrastruktur, 65
    Metamodell, 114
    Profil, 2
    Superstruktur, 65
UMLX, 31
URI, 97
URN, 97

V-Modell, 17
Validierung, 26, 116
Velocity, 37, 58, 166
Vereinigungsmengenbeziehung, 75
Vererbung, 72, 77, 119, 139
Verhalten, 131
Verifikation, 26
VIATRA, 33
VMT, 33
Vorbedingung, 76
Vorgehensmodell, 25

W3C, 93
Werkzeug, 2, 3

Werkzeugkette, 2, 3
Wertebreich, 82
Wiederverwertbarkeit, 7
wohlgeformt, 94, 126
Wurzeleigenschaft, 145
Wurzelelement, 94, 98
Wurzelknoten, 116, 169, 178
WWW, 97

XDoclet, 37
XMI, 3, 6, 34, 49, 58, 113
    JMI-Schnittstelle, 158
    Reader, 50, 58, 162
XML, 3, 6, 93, 113
XML-Schema, 102
    Datentyp, 102
    Einführung, 103
    Struktur, 102
XSLT, 34, 54, 168

Zeichenkette, 82, 157
Zielmodell, 26
Zwangsbedingungen, 89