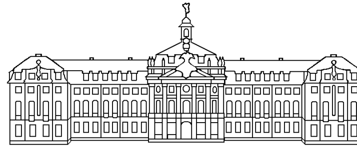
Westfälische Wilhelms-Universität Münster



Diplomhausarbeit

Modellgetriebene Entwicklung von Webapplikationen

Ulrich Wolffgang

In Kooperation mit der

Themensteller: Prof. Dr. Herbert Kuchen
Betreuer: Dipl.-Wirt.Inform. Christoph Lembeck
Institut für Wirtschaftsinformatik
Praktische Informatik in der Wirtschaft

Inhaltsverzeichnis

1 Übersicht **1**

2 Model Driven Development **3**

2.1 Terminologie . . . 3
2.2 Metamodellierung . . . 4
2.3 Domänenspezifische Sprachen . . . 6

3 openArchitectureWare **9**

3.1 Überblick . . . 9
3.2 Ecore und EMF . . . 11
3.3 Xpand . . . 15
3.3.1 Polymorphie . . . 18
3.3.2 Schlüsselwörter . . . 21
3.4 Xtend . . . 22

4 Modellierung von Webapplikationen **26**

4.1 Webapplikationen . . . 26
4.2 Web Engineering . . . 27
4.3 Evaluation von Webmodellierungssprachen . . . 29
4.3.1 Web Application Extension . . . 31
4.3.2 Web Modeling Language . . . 33
4.3.3 UML-based Web Engineering . . . 34

5 Web Application Modeling Language **40**

5.1 Profil Content . . . 40
5.2 Profil Control . . . 43
5.3 Profil View . . . 43
5.4 Meta-Assoziationen . . . 44
5.5 Selektion eines Navigationsdiagrammtyps . . . 45

6 Entwicklung einer prototypischen Webapplikation **47**

6.1 Modelle . . . 47
6.2 Zielplattform . . . 50
6.3 Projektressourcen . . . 51
6.3.1 Struktur . . . 51
6.3.2 Workflow . . . 53
6.4 Basistemplates und -funktionen . . . 54

6.4.1 Primäres Roottemplate . . . 54
6.4.2 Templates für Klassen . . . 55
6.4.3 Funktionen für Klassen . . . 58
6.4.4 Funktionen für Typen . . . 60
6.4.5 Funktionen für Tagged Values . . . 62
6.5 Profil Content . . . 63
6.5.1 Templates für Contentklassen . . . 64
6.5.2 Funktionen für Contentklassen . . . 65
6.5.3 Templates für Geschäftsklassen . . . 66
6.5.4 Funktionen für Geschäftsklassen . . . 69
6.6 Profil Control . . . 69
6.6.1 Templates für Controller . . . 70
6.6.2 Funktionen für Controller . . . 75
6.6.3 Templates für Navigationsknoten . . . 76
6.6.4 Templates für Descriptoren der Navigation . . . 78
6.7 Profil View . . . 80
6.7.1 Template für CreateView . . . 81
6.7.2 Template für IndexView . . . 83
6.8 Generierte Webapplikation . . . 84

7 Bewertung und Ausblick 86

Literaturverzeichnis 88

Abkürzungsverzeichnis

CASE .... Computer Aided Software Engineering
CIM .... Computation Independent Model
CRUD .... Create, Read, Update and Delete
DSL .... Domain Specific Language
DTD .... Document Type Definition
EJB .... Enterprise JavaBeans
EMF .... Eclipse Modeling Framework
EMOF .... Essential MOF
Java EE .... Java Platform, Enterprise Edition
JSF .... JavaServer Faces
JSP .... JavaServer Pages
MDA .... Model Driven Architecture
MDD .... Model Driven Development
MOF .... Meta-Object Facility
oAW .... openArchitectureWare
OMG .... Object Management Group
PIM .... Platform Independent Model
PSM .... Platform Specific Model
UWE .... UML-based Web Engineering
W3C .... World Wide Web Consortium
WAE .... Web Application Extension
WAML .... Web Application Modeling Language
WebML .... Web Modeling Language
XMI .... XML Metadata Interchange

Abbildungsverzeichnis

1 Metaebenen 5
2 oAW Strukturdiagramm [48] 12
3 Metametamodell von Ecore [3] 13
4 Metamodell mit Polymorphie durch Überladen 19
5 Metamodell mit Typvererbungshierarchie 21
6 Methodenbegriff [25] 30
7 Beispielmodell in WAE [17] 32
8 Beispielmodell in WebML [14] 34
9 Ausschnitt aus dem UWE-Metamodell 35
10 Metamodell der Web Application Modeling Language 41
11 Navigationsmodell als Aktivitätsdiagramm 46
12 Klassendiagramm des Prototyps 48
13 Navigationsdiagramm des Prototyps 49
14 Projektressourcen 52
15 Ausschnitt aus der generierten Webapplikation 85

1 Übersicht

> „Controlling complexity is the essence of computer programming.“
> (Brian W. Kernighan)

Mit der zunehmenden Mächtigkeit von Computersystemen und den steigenden Anforderungen an Software ist seither nicht selten ein Scheitern von Software-Projekten verbunden. Methoden zur Beherrschung der Softwarekrise ist stets gemein, die Komplexität von Softwaresystemen durch Abstraktionsmechanismen beherrschbar zu gestalten. Durch Abstraktion soll stets die Menge kritischer Systemelemente bzw. deren Abhängigkeiten voneinander reduziert werden und sollen fachliche Aspekte der Problemdomäne in den Vordergrund gerückt werden.

Modelle als zweckorientierte abstrahierende Abbildungen von Sachverhalten bieten sich als eine Form der Repräsentation von Softwaresystemen an. Das Computer-Aided Software Engineering (CASE) der neunziger Jahre verfolgt den methodischen Ansatz, Software werkzeuggestützt ingenieurmäßig zu entwickeln. Ein Bestandteil von CASE ist die intensive Verwendung von Modellen zum Zweck der Dokumentation und partiellen Codegenerierung. Die modellgetriebene Entwicklung verfeinert die CASE-Methode durch die Möglichkeit, die Transformation von Modellen in Code adaptierbar zu gestalten. Daraus sollen Vorteile im Sinne von Effizienz- und Qualitätssteigerungen, erhöhter Portabilität und Flexibilität resultieren.

Diese Arbeit widmet sich der Betrachtung der modellgetriebenen Entwicklung von Webapplikationen in Theorie und Praxis. Dazu werden in Kapitel 2 die Vision und die grundlegenden theoretischen Konzepte der modellgetriebenen Softwareentwicklung vorgestellt. Die Konzepte werden in Kapitel 3 mit dem Generatorframework openArchitectureWare (oAW), einem in der Praxis verwendeten Codegenerator, gegenübergestellt, der eine Realisierung von Projekten auf Basis der modellgetriebenen Entwicklung ermöglichen soll. oAW repräsentiert den status quo in der praktischen Anwendung der modellgetriebenen Entwicklung und wird für die Generierung einer prototypischen Webapplikation im Rahmen dieser Arbeit verwendet. In Kapitel 4 wird auf die Modellierung von Webapplikationen eingegangen. Nach einer Festlegung der Auswahlkriterien werden gängige Modellierungssprachen für Webapplikationen hinsichtlich der Eignung für die modellgetriebene Entwicklung einer Evaluation unterzogen. Kapitel 5 umfasst die Spezifikation und Beschreibung der Web Application Modeling Language (WAML). Diese ist eine Alternative zu den Sprachen in Kapitel 4 und soll konstruktiv einen Lösungsvorschlag für die kritisierten Aspekte dieser Sprachen darstellen. In Kapitel 6 werden die Inhalte der

theoretischen Kapitel praktisch angewendet, indem exemplarisch eine Webapplikation in der WAML modelliert wird und ein Generator vorgestellt wird, mit dem die WAML-Modelle in lauffähigen Quelltext übersetzt werden können.

Das Ziel dieser Arbeit ist, die mit der modellgetriebenen Entwicklung verbundenen Versprechungen durch eine konkrete prototypische Realisierung auf Praktikabilität zu überprüfen. Insbesondere ist es von Relevanz zu prüfen, ob eine Webapplikation mit vertretbarem Aufwand ohne manuelles Eingreifen vollständig automatisiert generiert werden kann.

2 Model Driven Development

2.1 Terminologie

Modellgetriebene Softwareentwicklung (Model driven software development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen. [45] Zentral für den Softwareentwicklungsprozess des MDSD, auch Model Driven Development (MDD) genannt, ist also, dass Modelle nicht ausschließlich der Dokumentation eines Softwaresystems dienen, sondern aus den Modellen konstruktiv die ausführbare Anwendung erzeugt wird. Begrifflich verwandt mit dem MDD ist die Model Driven Architecture (MDA), die eine herstellerunabhängige Standardisierung des MDD durch die Object Management Group (OMG) darstellt.

Die Erstellung von Software im MDD geschieht ähnlich dem Computer-Aided Software Engineering (CASE) als direkte Generierung von Quelltext aus Modellen. Im Unterschied zu CASE sind beim MDD aber die Transformationsdefinitionen nicht proprietär oder hart kodiert, sondern können eingesehen und modifiziert werden. Außerdem ist ein Bestandteil der Vision des MDD, dass ausgehend von fachlichen Modellen technische Modelle generiert werden, aus welchen Quelltext generiert wird, was auch forward engineering genannt wird. Wesentlich ist also die Konkretisierung abstrakter fachlicher Konzepte. Dies steht im Gegensatz zum reverse engineering, bei dem aus der Implementierung oder aus spezifischeren Modellen abstraktere Modelle generiert werden.

Auf der Modellebene wird in der Terminologie der OMG zwischen dem fachlichen Computation Independent Model (CIM), dem plattformunabhängigen Plattform Independent Model (PIM) und dem technischen Platform Specific Model (PSM) unterschieden. Das CIM ist eine Beschreibung der domänenspezifischen Zusammenhänge, unabhängig von technischen Aspekten einer möglichen Implementierung. Das PIM beschreibt bereits technische Anforderungen, lässt aber plattformspezifische Details aus, während das PSM die Konzepte einer Plattform verwendet, um ein System zu beschreiben. [44] Eine Plattform ist eine Menge von Subsystemen und Technologien, die eine Menge zusammenhängender Funktionalität durch Schnittstellen und spezifische Nutzungsmuster bereitstellt, welche durch die Plattform unterstützte Applikationen nutzen können. Die Applikationen müssen dazu keine detaillierten Informationen über die Implementierung der Funktionalität der Plattform besitzen. Beispiele für Plattformen sind Java EE, Microsoft .NET und Intel x86. [36]

Das PSM wird aus dem PIM mittels eines Transformators bzw. Generators ge-

wonnen. Aus dem PSM wird mittels eines weiteren Transformators Quelltext generiert. So kann theoretisch über Zwischenschritte ein Modell mit einem hohen Abstraktionsgrad in eine implementierungsnahe Repräsentation umgewandelt werden. Transformationen sind konzeptionell auch von PIM zu PIM möglich, also auf derselben Abstraktionsebene. Dies kann bei einer Transformation zwischen Modellen derselben Sprache, z. B. bei der Normalisierung eines Entity-Relationship-Modells oder dem Coderefacturing von Java-Quelltext, genutzt werden. Eine Transformation ist definiert als die automatische Generierung eines Zielmodells aus einem Quellmodell, entsprechend einer Transformationsdefinition. [29] Die Transformationsdefinition besteht aus Transformationsregeln, die beschreiben, wie ein Modell in einer Quellsprache in ein Modell in einer Zielsprache übersetzt werden kann. In der Praxis wird die Trennung von CIM, PIM und PSM häufig nicht durchgeführt, sondern stattdessen das PIM mit technischen Zusatzinformationen angereichert und daraus direkt Quelltext erzeugt. Dies ist in einem reduzierten Pflegeaufwand und in der mangelnden Verfügbarkeit entsprechender Generatoren begründet. [42, S. 14]

Die Transformatoren setzen formale Modelle voraus, deren statische Semantik durch Metamodelle definiert werden kann. Das Metamodell kann eine Spezialisierung des Metamodells einer bestehenden Sprache wie der Unified Modeling Language (UML) sein und sich damit an die Semantik einer Sprache anlehnen. Alternativ ist die Definition einer Domain Specific Language (DSL) möglich, die eine Domäne durch ihre passgenaue Semantik optimal beschreiben kann.

2.2 Metamodellierung

Die Modellierung von Sachverhalten geschieht in der Regel aus Gründen der Einheitlichkeit in einer formalisierten Modellierungssprache. Die Modellierungssprache kann durch ein Modell beschrieben werden, indem die Sprachelemente der Modellierungssprache durch Modellelemente spezifiziert werden. Für die MDD bzw. MDA wird eine Untergliederung in vier aufeinander aufbauende Ebenen vorgenommen, die in der Terminologie der OMG M0 bis M3 genannt werden: [23]

Ebene M0: Die Ebene M0 ist die Instanzebene: Sie repräsentiert das laufende System mit seinen aktuellen Instanzen, wie z. B. Objekten in einem objektorientierten Programm oder Datensätzen in einem relationalen Schema. Dabei muss je nach Kontext zwischen verschiedenen Arten von Instanzen unterschieden werden. Z. B. repräsentieren im Kontext einer Datenbankmodellierung die Instanzen einer Rechnungstabelle Rechnungen, wogegen im Kontext einer Geschäftprozessmodellierung die Instanzen die konkreten physischen Rechnungen

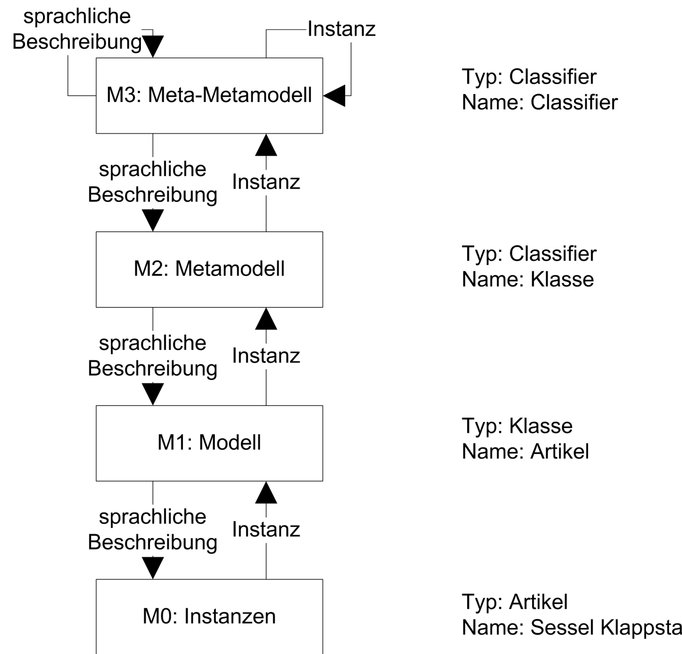


Abbildung 1: Metaebenen

sein können.

Ebene M1: Die Ebene M1 stellt die Abstraktion der Ebene M0 dar. Dabei wird eine Abbildung der Realität durch Aggregation der relevanten Eigenschaften der Instanzen zu einem bestimmten Zweck erstellt. Die Elemente der Ebene M1 können durch Attribute in ihren Eigenschaften beschrieben werden und zueinander in Beziehung stehen. Auf der Ebene M0 werden die Attribute der Elemente der Ebene M1 mit Werten ausgeprägt. Die Wertebereiche der Attribute können durch Constraints beschränkt werden. Beispiele für Modelle der Ebene M1 sind UML-Klassendiagramme oder Entity-Relationship-Modelle.

Zwischen den Instanzen auf der Ebene M0 und den Modellelementen auf der Ebene M1 besteht eine Typbeziehung, durch die sich die Instanzen klassifizieren lassen. Instanzen müssen durch mindestens ein Modellelement klassifiziert bzw. typisiert werden, um im Sinne des Modells zulässig zu sein. Im Rahmen des MDD können Modelle im herstellerunabhängigen Format XML Metadata Interchange (XMI) gespeichert werden.

Ebene M2: Auf der Ebene M2 befindet sich das sprachliche Modell eines Modells, also das Metamodell. Im Metamodell werden die Sprachelemente spezifiziert, mit denen Modelle auf der Ebene M1 erstellt werden können. Analog zu der Beziehung von Ebene M0 zu Ebene M1 drücken Instanz-Typ-Beziehungen zwischen M1 und M2 aus, ob das Modell M1 im Sinne von M2 zulässig ist.

Mit den auf der Ebene M2 spezifizierten Modellierungssprachen können syntaktisch und formal semantisch korrekte Modelle erstellt werden. Z. B. hat die OMG die Modellierungssprache UML mit einem Metamodell spezifiziert. Modellelemente des UML-Metamodells auf der Ebene M2 sind unter anderem die UML Class und das UML Attribute.

Ebene M3: Ein Modell der Ebene M3 ist das Metametamodell der Instanzen auf der Ebene M0. Ein sprachliches Metametamodell auf der Ebene M3 spezifiziert eine Metasprache, mit der Modelle der Ebene M2 formuliert werden können. Beispielsweise kann auf der Ebene M2 die Sprache UML mit Elementen des Metametamodells spezifiziert werden. Wenn mehrere Modellierungssprachen durch dieselbe Metasprache beschrieben werden, können sie einheitlich durch sprachunabhängige Generatoren verarbeitet werden. Die Flexibilität der Generatoren, beliebige Modellierungssprachen der Ebene M2 verarbeiten zu können, unterscheidet das MDD vom Case-Ansatz, bei dem Modellierungssprachen bzw. deren Generatoren gewöhnlich hart kodiert sind. Metasprachen ermöglichen es, neben UML auch domänenspezifische Sprachen zu spezifizieren und für die Transformation zu verwenden. Analog zu den Beziehungen zwischen Elementen der Ebene M1 und Ebene M2 sind Typbeziehungen zwischen Elementen der Ebene M2 und M3 vorhanden. Für die Spezifikation der Elemente der Metasprache existiert keine Metametasprache, sondern die Metasprache beschreibt sich rekursiv selbst, die Elemente der Metasprache sind also zugleich ihre Typen bzw. Instanzen. Dies ist möglich, weil das Metametamodell generisch Elemente und Beziehungen zwischen den Elementen beschreibt.

Die OMG sieht auf der Ebene M3 die Meta-Object Facility (MOF) vor. Ein weiteres Metametamodell ist Ecore, das im Eclipse Modeling Framework (EMF) enthalten ist. Ecore entspricht weitestgehend dem Essential MOF (EMOF), das eine Teilmenge des MOF ist. Ursprünglich nicht für das MDD entwickelt ist zusätzlich XML Schema ein weiteres Metametamodell.

2.3 Domänenspezifische Sprachen

Im CASE-Ansatz werden Modelle primär in den Sprachen UML und ERM erstellt. Das MDD besitzt in dieser Hinsicht eine größere Flexibilität, weil neben neben UML und ERM domänenspezifische Sprachen (domain-specific language, DSL) für die Modellierung und Transformation erstellt und genutzt werden können. Mit einer DSL können Sachverhalte einer Domäne optimal abgebildet werden, weil sie die Konzepte der Domäne namentlich repräsentieren und somit eine höhere semantische Präzision

besitzen. Für die Spezifikation domänenspezifischer Modellierungssprachen existieren drei Ansätze: [38, S. 9]

eigenes Metamodell: Die Modellierungssprache kann mit einem eigenen Metamodell in einer Metasprache wie z. B. MOF oder Ecore beschrieben werden. Dazu werden sämtliche Elemente und Beziehungen der domänenspezifischen Modellierungssprache unabhängig von existierenden Sprachen wie UML oder ERM grundlegend neu spezifiziert. Auf diese Weise können die Struktur und die Semantik der Elemente der Modellierungssprache so definiert werden, dass sie optimal die Charakteristika der Domäne abbilden. Nachteile ergeben sich aber bei der Werkzeugunterstützung, da CASE-Tools bzw. deren Modellierungsumgebungen üblicherweise auf UML basieren.

leichtgewichtige Erweiterungen: Bei leichtgewichtigen Erweiterungen wird das UML-Metamodell um zusätzliche Elemente angereichert. Dies kann auf zwei Arten geschehen: [45]

Spezialisierung: Das Metamodell kann erweitert werden, indem dem Metamodell weitere Elemente hinzugefügt werden, die Vererbungsbeziehungen zu bestehenden Metamodellelementen besitzt. Dazu muss die Metasprache des Metamodells den Spezialisierungsmechanismus unterstützen, was im Fall der UML mit dem MOF-Element *generalizes* gewährleistet ist. Die Erweiterung durch Spezialisierung kann generell bei allen Sprachen genutzt werden, deren Metasprachen Spezialisierungsbeziehungen vorsehen, ist aber im Gegensatz zu DSLs typisch für UML, da eine Sprache vorausgesetzt wird, deren Elemente spezialisiert werden.

Profil: Das UML-Metamodell stellt seit der zweiten Version mit dem Element *extension* ein eigenes Sprachkonstrukt für Erweiterungen des Metamodells bereit, mit dem UML-Klassen durch Stereotypen spezialisiert werden können. Stereotypen können Eigenschaften besitzen, die Tagged Values genannt werden und seit der zweiten UML-Version typisiert sind. In der ersten UML-Version konnten Tagged Values nur Werte vom Datentyp String enthalten [8, S. 647], sodass Werte von Tagged Values keine Referenzen auf Modellelemente enthalten konnten, sondern nur eindeutige Namen der referenzierten Modellelemente. Die Ermittlung von Referenzen aus den eindeutigen Namen schlägt sich in der Komplexität der Transformationsdefinitionen nieder.

Das Metamodellelement *Profil* ist eine Spezialisierung von *Package* mit der Besonderheit, Stereotypen enthalten zu können. Im Gegensatz zur Erweiterung des Metamodells durch die Spezialisierung von Packages werden Profile standardisiert als Sprachpaket interpretiert. Unterschiedlich zu Packages werden Profile nicht auf der Modellebene M1 zur Modellierung verwendet, sondern ausschließlich auf der Metamodellebene M2 zur Metamodellierung.

schwergewichtige Erweiterungen: Im Unterschied zu leichtgewichtigen Erweiterungen werden bei schwergewichtigen Erweiterungen neben der Erweiterung des UML-Metamodells durch neue Elemente oder Stereotypen auch Bestandteile des UML-Metamodells ausgeblendet. Diese Form der Einschränkung ist in UML nativ nicht vorgesehen, kann aber durch Constraints nachgebildet werden, indem diese die Verwendung von bestimmten UML-Metamodellelementen verbieten. Das UML-Metamodell wird durch schwergewichtige Erweiterungen nicht reduziert, sondern nur die Nutzung von Elementen restringiert, die Modellierung wird also auch aufgrund fehlender Werkzeugunterstützung nicht substantiell vereinfacht. Auch wenn sich in der Terminologie der Begriff der schwergewichtigen Erweiterung durchgesetzt hat, ist im Sinne der Ausblendung von Metamodellelementen eigentlich der Ausdruck Adaption präziser.

3 openArchitectureWare

3.1 Überblick

Für die modellgetriebene Entwicklung von Webapplikationen existieren mit den Generatorframeworks openArchitectureWare und AndroMDA zwei vorherrschende Alternativen. Beide können sowohl UML-basierte Modelle als auch Modelle in domänenspezifischen Sprachen verarbeiten. Der Fokus liegt bei AndroMDA historisch auf der Verwendung UML-basierter Cartridges, in denen vorgefertigte plattformspezifische Transformationsdefinitionen enthalten sind. Bei openArchitectureWare steht die Entwicklung durch domänenspezifische Transformationsdefinitionen in der Entwicklungsumgebung Eclipse im Vordergrund. Für diese Arbeit wird openArchitectureWare verwendet, da diese eine typisierte Templatesprache enthält, umfangreiche Integration in Eclipse bietet und in der viadee IT-Unternehmensberatung als Kooperationspartner für diese Arbeit bereits produktiv eingesetzt wird.

openArchitectureWare (oAW) ist ein MDA/MDSD-Generatorframework, das aus der b+m ArchitectureWare der b+m Information AG hervorgegangen ist. Als Teil des Eclipse Modeling Project wird openArchitectureWare im Generative Modeling Technologies (GMT) Project weiterentwickelt und steht unter der Eclipse Public License (EPL). [48] Die EPL [2] entspricht der Open Source Definition [6] der Open Source Initiative (OSI) und ist von der OSI offiziell als Open Source Lizenz anerkannt, sodass die openArchitectureWare als quelloffen und frei angesehen werden kann.

openArchitectureWare ist ein in Java implementierter Codegenerator, welcher Werkzeugunterstützung bestehend aus Modellbrowsern und Editoren enthält und auf der Eclipse Plattform basiert. Mit oAW können Modelle, Metamodelle und Transformationsdefinitionen erstellt und visualisiert werden. Eine Workflowengine unterstützt die Verarbeitung von Modellen hinsichtlich des Imports und der Instanziierung, der Überprüfung gegen Constraints und der Transformation durch templatebasierte Erweiterungen.

Die Transformationsengine *Xpand* kann beliebige textuelle Ausgaben erzeugen, sodass oAW bei Transformationen auch hinsichtlich der Zielsprache flexibel ist. Die Transformationsdefinitionen werden templatebasiert formuliert und können Typen des Quellmetamodells referenzieren. So kann die Struktur der Templates an der Struktur des Quellmetamodells ausgerichtet werden. Ein mit der oAW verbundenes Ziel ist es, die Etablierung von vorgefertigten Transformationsdefinitionspaketen für häufig auftretende Transformationsprobleme voranzutreiben. Diese Cartridges ge-

nannten Pakete unterstützen z. B. die Transformation von plattforumunabhängigen Quellmodellen in plattformspezifische J2EE-Zielimplementierungen. Im Rahmen der Fornax-Plattform werden z. B. Cartridges für die Transformation von UML2-Quellmodellen in die Zielframeworks EJB3, Hibernate und Spring entwickelt. [4]

Wesentlich für die MDA ist die Schichtung von Modellen anhand ihres Abstraktionsgrades bzw. ihrer Plattformabhängigkeit. oAW ermöglicht die Umsetzung dieses Ansatzes durch die Unterstützung von Modell-zu-Modell-Transformationen mittels der funktionalen Sprache *Xtend*. Durch Xtend können Funktionen definiert werden, mit denen unter anderem Modelle verschiedener Metamodelle anhand von Zuordnungsvorschriften zwischen den Metamodellelementen ineinander überführt werden.

Mit der Sprache *Check* können Metamodelle um Constraints erweitert werden, um Restriktionen zu erfassen, die nicht durch die Struktur des Metamodells ausgedrückt werden können. Z. B. kann durch Check die Einhaltung von Namenskonventionen bei Modellelementen validiert werden, die nicht durch Beziehungen der Elemente im Metamodell sichergestellt werden können.

xText ist ein Framework zur Verarbeitung von textuellen Modellen, deren Sprachen in textuellen Metamodellen ähnlich zur erweiterten Backus-Naur-Form (EBNF) formuliert werden können. Passend zu der textuellen Sprachdefinition wird ein Eclipse-Editor bereitgestellt, in dem die Modelle entsprechend der Syntax erstellt werden können. Die Editoren verfügen über die Fähigkeiten des Syntax Highlighting, der Auto Completion und der syntaxabhängigen Fehlermarkierung. Die Modelle bzw. deren Metamodelle können in Xpand zur Transformation benutzt werden. [42] [20] Analog zu xText können mit dem *Graphical Modeling Framework* (GMF) graphische Editoren für domänenspezifische Sprachen generiert werden. Bei der Generierung eines graphischen Editors wird das Metamodell der domänenspezifischen Sprache zugrunde gelegt. Falls während des Modellierungsvorganges in einem solchen Editor Constraints des Metamodells verletzt werden, wird dies entsprechend visualisiert.

Das *Recipe Framework* ermöglicht im Anschluß an eine Generierung von Quelltext die Validierung, dass manuell hinzugefügter Code vollständig und korrekt im Sinne der Sprache ist. Z. B. kann nach der Generierung eines Javainterfaces überprüft werden, ob dieses mindestens einmal von einer manuell hinzugefügten Klasse implementiert wird.

Der generierte Quellcode ist häufig nicht optimal formatiert, weil die korrekte Angabe von Einrücktiefen und Zeilenumbrüchen in Xpand-Templates die Erstellung von Templates verkompliziert. Um den Problemaspekt der korrekten Formatierung außerhalb der Templates zu behandeln, können in den Generatorprozess, der auch

Workflow genannt wird, Beautifier eingebunden werden. Diese parsen und formatieren den generierten Quellcode entsprechend der Syntax und sind somit spezifisch hinsichtlich der Zielsprache. Im Unfang von oAW sind Beautifier für Java-Quelltext und XML-Code enthalten.

oAW enthält einen Debugger, der sich in die Eclipse Debugging Perspective integriert. Er unterstützt die schrittweise Analyse von Workflows, Transformationen und Templates im Kontext geladener Modelle.

Mit *Ecore* enthält oAW ein Metametamodell, das weitestgehend dem Essential MOF (EMOF) entspricht, einer Teilmenge des MOF. Ecore stellt Metametamodellelemente zur Verfügung, mit denen Metamodelle erstellt werden können. Aufgrund des gemeinsamen Metametamodells können die erstellten Metamodelle gleichermaßen in oAW verwendet werden.

3.2 Ecore und EMF

Ecore ist ein auf der Ebene M3 einzuordnendes Metametamodell, mit dem Metamodelle der Ebene M2 wie z. B. das UML-Metamodell und domänenspezifische Sprachen definiert werden können. Modelle der Ebene M1, die Elemente dieser Metamodelle instanziieren, können im Eclipse Modeling Framework (EMF) verarbeitet werden, um Modell-zu-Modell-Transformationen durchzuführen und Quelltext zu generieren. Das EMF ist zum einen eine Implementierung der Strukturen von Ecore in Form von Javaklassen, zum anderen ist es ein Framework zur Verarbeitung dieser Daten.

Im Umfang von Ecore sind Sprachelemente enthalten, mit denen durch den Modellierer eigene Sprachen definiert werden können. Im Wesentlichen drückt Ecore in einer allgemeinen Form aus, dass es Elemente gibt, und dass diese Elemente miteinander in Beziehung stehen können. Im Mittelpunkt stehen die Elemente *EClass*, *EDataType*, *EAttribute* und *EReference*, die Spezialisierungen von *ENamedElement* sind. [3] ENamedElement hat das Attribut *name*, mit dem Instanzen von Ecore, also den Metamodellelementen, Namen zugeordnet werden können.

Die Elemente von Ecore werden folgendermaßen benutzt: [12]

EClass repräsentiert eine Metamodellklasse und hat mittels ENamedElement einen Namen. Über EAttribute können Instanzen von EClass Attribute enthalten und analog zu EReference Referenzen zu anderen Instanzen von EClass besitzen. Um Vererbung zu unterstützen, kann eine EClass mit dem Attribut eSuperTypes mehrere Oberklassen haben.

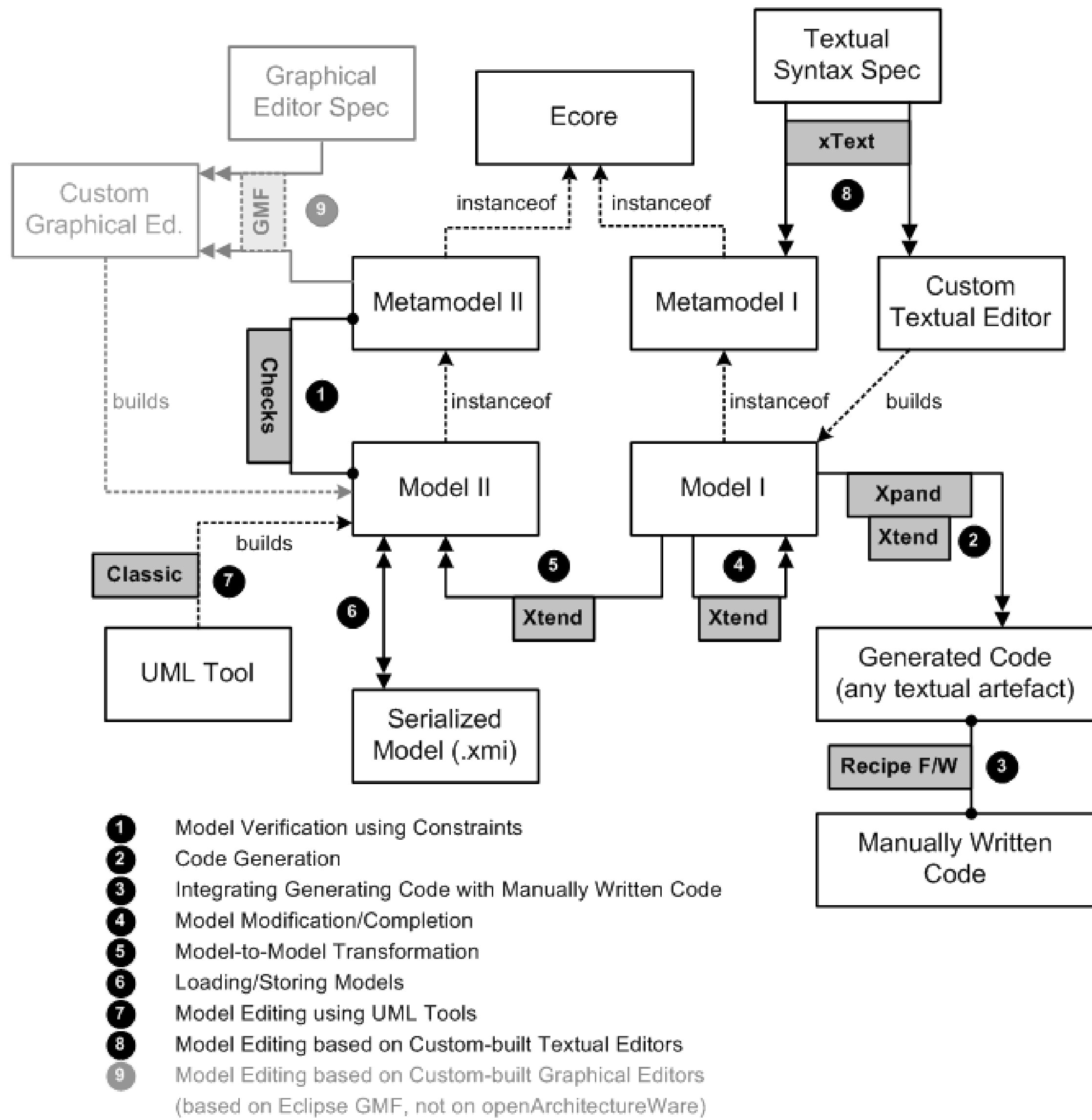


Abbildung 2: oAW Strukturdiagramm [48]

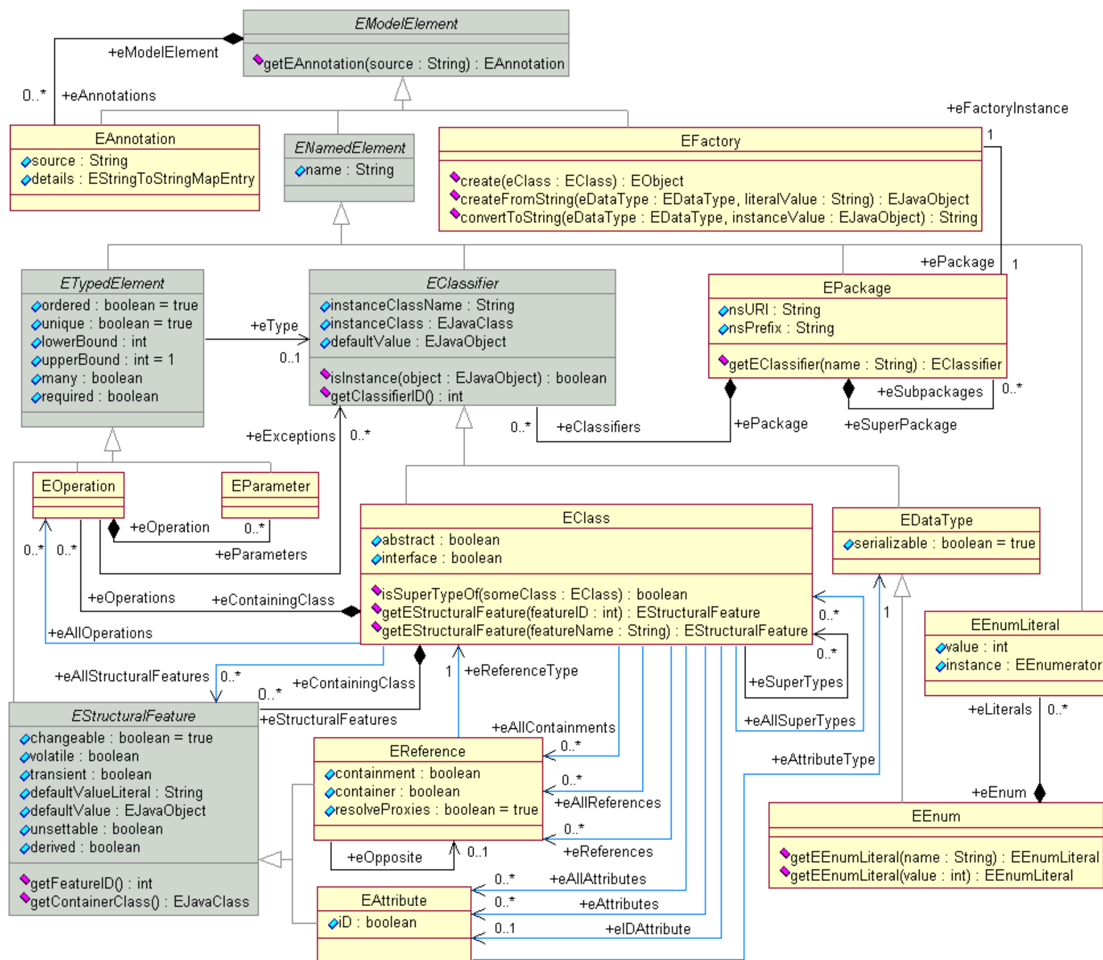


Abbildung 3: Metametamodell von Ecore [3]

EAttribute wird genutzt, um Attribute zu modellieren, die einen Namen und einen Typ haben. Der Typ eines Attributes ist festgelegt über die Assoziation zu EDataType.

EReference wird genutzt, um ein Assoziationsende zwischen Klassen darzustellen. Über die Assoziation eReferenceType wird das Ziel der Referenz erfasst, das vom Typ EClass sein muss. Falls eine Referenz bidirektional ist, existiert eine entgegengesetzte korrespondierende EReference.

EDataType repräsentiert Typen außerhalb von Ecore, wie z. B. Typen aus dem Typsystem von Java. Diese können sowohl primitive als auch komplexe Datentypen wie z. B. int oder java.util.Date sein.

EStructuralFeature aggregiert die gemeinsamen Eigenschaften von EReference und EAttribute. So kann z. B. mit dem Boolean *changeable* angegeben werden, ob ein EAttribute bzw. eine EReference außerhalb der EClass modifiziert werden darf.

EOperation dient der Modellierung von Verhaltensaspekten einer EClass, indem mit EOperation Signaturen von Operationen definiert werden können. Anzumerken ist, dass mit EOperation nicht herkömmliche Methoden einer Klasse auf der Modellebene M1 wie z. B. Gettermethoden modelliert werden, sondern Metamethoden auf der Metamodellebene M2. Z. B. enthält im EMF-basierten UML-Metamodell das Element *org.eclipse.emf.ecore.EModelElement* die Methode *getOwnedElements()*, die eine Liste von enthaltenen Elementen zurückgibt. [7] In einem UML-Modell kann in einer Transformationsdefinition diese Operation z. B. auf einem Paket ausgeführt werden, um eine Liste von Klassen und Subpaketen des Pakets zu erhalten.

EParameter ermöglicht die Angabe von Parametern in der Signatur von EOperations. Eine EOperation kann mehrere EParameter besitzen.

Die in Ecoreklassen und -interfaces verwendeten Primitive *boolean* und *int* und der Typ *String* sind Java-Typen, da EMF in Java implementiert ist. Bei der Programmierung von EMF-basierten Transformationsdefinitionen sind z. B. die in Ecoreelementen enthaltenen Strings einfache Java-Strings und stellen die üblichen Stringmethoden wie z. B. *trim()* zur Verfügung.

Die Benennung der Elemente in Ecore weist eine Ähnlichkeit mit der Benennung der Elemente im Metamodell von UML auf, da Ecore in Anlehnung an EMOF definiert ist, und EMOF eine für die Metamodellierung uminterpretierte Teilmenge des

UML-Metamodells ist. Dennoch wird nicht UML als Metasprache bzw. Metametamodell benutzt, da es für den Zweck der Metametamodellierung zu umfangreich ist. So werden sprachlich keine Verhaltensdiagramme benötigt, um die Metamodelle der Ebene M2 zu definieren, da dazu ausschließlich strukturelle Aspekte beschrieben werden müssen. Aus diesem Grund ist es sinnvoll, UML zum Zwecke der Metametamodellierung die genannten Strukturbeschreibungsaspekte von EMOF zu reduzieren.

In EMF ist Ecore durch Java-Interfaces umgesetzt. Interfaces wie z. B. *EClassifier* und *EStructuralFeature* aggregieren Eigenschaften und werden durch ihre Subinterfaces *EClass* und *EDataType* respektive *EReference* und *EAttribute* spezialisiert. Ein Metamodell der Ebene M2 kann in EMF mit Bezug auf Ecore-Elemente erstellt werden, indem die Metamodellelemente die Ecore-Elemente implementieren. Eine Besonderheit von EMF ist die Existenz von abstrakten Interfaces wie z. B. bei *EClassifier* im Gegensatz zu einfachen Interfaces wie z. B. bei *EClass*. Da in Java keine abstrakten Interfaces vorgesehen sind, werden beide Fälle als einfache Java-Interfaces implementiert.

Ein Beispiel für die Verwendung von Ecore im Rahmen einer Implementierung ist die Definition einer Sprache für die Modellierung von Geschäftsklassen. Beispielsweise kann das Metamodell einer solchen Sprache auf der Ebene M2 ein Element BusinessClass vom Ecore-Metametamodelltyp EClass enthalten. In einem Modell auf der Ebene M1 kann dieses Sprachelement BusinessClass dann z. B. als die Klasse BibliotheksSessionBean instanziiert werden, die Methoden für die Geschäftslogik einer Bibliothek enthält . In der laufenden Applikation auf der Ebene M0 wird diese Klasse dann als Objekt instanziiert.

Da die Sprache auf der Ebene M2 in Ecore verfasst ist, kann EMF für die Interpretation dieser Sprache und die Generierung von Zielmodellen auf der Ebene M1 verwendet werden. Die Sprache implementiert im Metamodell die Interfaces von Ecore, sodass EMF über die in diesen Interfaces spezifizierten Methoden standardisiert auf die Sprachelemente zugreifen kann.

3.3 Xpand

Xpand ist eine im Umfang von oAW enthaltene Templatesprache, die es ermöglicht, Werte aus Eingabemodellen abzufragen, diese mit statischem Text zu konkatenieren und in Dateien zu schreiben. Mit Xpand können Definitionen für Modell-zu-Text-Transformationen in Form von Templates erfasst werden. Die Templates bestehen aus beliebigem Text, der punktuell mit Xpand-Tags versehen ist. Während des Ge-

nerierungsvorgangs werden diese Tags von der Xpand-Templateengine evaluiert und die Ergebnisse in die entsprechenden Stellen im Text eingefügt. [45] Dieser Ansatz ähnelt templatebasierten Ansätzen im Webbereich, wie z. B. JavaServer Pages (JSP) und dem PHP Hypertext Preprocessor (PHP).

Die Verwendung von Templates im Gegensatz zur Generierung von Text durch eine imperative Sprache ist sinnvoll, weil damit der zu generierende Text im Vordergrund steht und nicht die Kommandos zur Erstellung des Textes, wie z. B. Konkatenationsoperatoren für Strings. So hat sich auch im Webbereich die Programmierung von dynamischen Webseiten durch Servlets als zu unhandlich erwiesen und führte zu alternativen templateorientierten Repräsentationsformen wie JSP.

Sämtliche Tags in Xpand werden durch Klammerung vom übrigen statischen Text getrennt. Die Klammerzeichen müssen dabei dem Anspruch genügen, möglichst selten als Zeichen im Korpus der vorstellbaren statischen Texte verwendet zu werden. Die Problematik wird z. B. an der Einleitung von Strings in Java deutlich. So führt die Verwendung von Anführungszeichen als Begrenzung von Strings in Java zu der Notwendigkeit, Anführungszeichen innerhalb eines Strings per Escaping kenntlich zu machen. Da Xpand generisch hinsichtlich der Zielsprachen sein soll, können die üblichen Klammern nicht als Trennzeichen verwendet werden, ohne Kollisionen in der Verwendung von Klammerungszeichen in den Zielsprachen zu erzeugen. Aus diesem Grund werden in Xpand Tags durch das öffnende « und schließende » Guillemot vom übrigen statischen Text getrennt. Da Guillemots nicht im ASCII-Zeichensatz enthalten sind, muss bei der Erstellung von Templatedateien darauf geachtet werden, eine Zeichenkodierung zu wählen, die diese Zeichen enthält. Dies kann z. B. ISO-8859-1 oder UTF-8 sein.

Ein einfaches Template zur Generierung eines Javaklassenrumpfes für ein Modellelement Entity kann folgendermaßen definiert werden:

```
1  «IMPORT metamodel»
2  «EXTENSION my::ExtensionFile»
3  «DEFINE javaClass FOR Entity»
4     «FILE fileName()»
5        package «javaPackage(this)»;
6        public class «this.name» {
7           // Implementierung
8        }
9     «ENDFILE»
10 «ENDDEFINE»
```

Listing 1: Xpand Beispieltemplate [21]

Das Template wird während der Generierung sequentiell evaluiert und das Er-

gebnis in eine Datei geschrieben.

Durch den Tag *IMPORT* wird der Namensraum *metamodel* in den Kontext der

```
1 «IMPORT meta::model»
```

Templatedatei importiert, die Listing 1 enthält. Nach dem Import können die Elemente des Metamodells ohne eine Angabe des Präfix *metamodel::* in den DEFINE-Blöcken referenziert werden. Dies ist bei dem Typ *Entity* der Fall, der ohne die Importanweisung mit *metamodel::Entity* referenziert werden müsste. Operationen, mit denen z. B. Java-Klassen- und Dateinamen von Modellelementen ermittelt werden, sollten außerhalb der Templates definiert werden. Die Ausgliederung dieser Operationen verbessert die Wiederverwendbarkeit und ermöglicht die Kapselung plattformabhängiger Detailaspekte wie z. B. der korrekten Auswahl von Dateiendungen.

Mit dem Tag *EXTENSION* wird eine Extend-Datei importiert. Falls diese in

```
1 «EXTENSION my::ExtensionFile»
```

Xtend formulierte Operationen enthält, können diese im Template wie im Fall der Operation *javaPackage()* aufgerufen werden. Das Ergebnis des Aufrufes wird von der Xpand-Engine in die entsprechende Stelle im Template eingefügt.

Templates werden in Xpand durch den Tag *DEFINE* eingeleitet und durch den Tag *ENDDEFINE* geschlossen. In der Signatur des Templates müssen der Name des Templates und eines Metamodellelements angegeben werden. Optional ist die Angabe einer kommaseparierten Liste von typisierten Parametern möglich, die dann im Kontext des Templates verfügbar sind.

```
1 «DEFINE nameDesTemplates(parameterListe) FOR MetaModelElement»
2   Templateinhalt...
3 «ENDDEFINE»
```

Nach der Evaluation eines Templates kann die Ausgabe in Dateien im Dateisystem geschrieben werden. Der zu speichernde Bereich wird mit dem Tag *FILE* eingeleitet und dem Tag *ENDFILE* geschlossen. Der Dateiname kann Verzeichnisangaben enthalten und wird relativ zum Outlet interpretiert. Bei der Definition von Generatorworkflows können mehrere Zielverzeichnisse angegeben werden, die in oAW Outlets genannt werden. Durch die Abstraktion vom Dateisystem werden die Templates von Dateisysteminformationen befreit, sodass in einer Gruppe von Mo-

```
1 «FILE "dateiName" outletName»
2   Templateinhalt...
3 «ENDFILE»
```

dellierern Probleme durch kollidierende Dateisystemstrukturen verhindert werden können und Templates plattformunabhängig verbreitet werden können.

Die Angabe des Metamodellelements ist für die Auswahl eines Templates während des Generatorlaufes von Bedeutung. Während der Transformation können die Elemente eines Eingabemodells durchlaufen werden und dabei Templates mit dem Tag *EXPAND* entfaltet werden. Das Template wird nur ausgeführt, falls der Typ

```
1 «EXPAND nameDesTemplates FOR InputMetaModelElement»
2 «EXPAND nameDesTemplates FOREACH InputMetaModelElementCollection»
```

eines Modellelementes, also das InputMetaModelElement, mit der Angabe des MetaModelElement im Template konform ist. Durch diesen Vergleich wird in Xpand Typsicherheit hergestellt. Falls bei einem Templateaufruf der in der Templatesignatur geforderte Typ mit dem übergebenen konform ist, wird das Template ausgeführt und eine Referenz auf das Modellelement unter dem Namen *this* im Template zur Verfügung gestellt. Analog zu *FOR* kann eine Collection von Elementen mit *FOREACH* expandiert werden.

Das mit Xpand verwendbare statische Typsystem umfasst sowohl die Elemente des in Ecore definierten Metamodells als auch die in Ecore vorgesehenen EDataType-Primitive, die automatisch auf Javatypen abgebildet werden. Z. B. sind im Umfang des UML-Metamodells die Primitive *Integer*, *String*, *Boolean* und *Unlimited Natural* enthalten.

3.3.1 Polymorphie

Die Templates können als eine funktionale Erweiterung ihrer Metaklasse im Sinne einer Methodendefinition interpretiert werden. So wird in Listing 1 mit dem Template *javaClass* die Metaklasse *Entity* um die Methode *javaClass()* angereichert.

Überladen: Überladen wird nach Cardelli u. Wegner [13] als ad-hoc-Variante der Polymorphie klassifiziert und erlaubt es, eine Methode bzw. ein Template für eine Metaklasse mehrfach mit unterschiedlichen Signaturen zu definieren. Z. B. können zwei Templates mit dem Namen *javaClass* für dasselbe Metamodellelement mit zwei verschiedenen Signaturen definiert werden. Falls mit einem EXPAND-Befehl eines

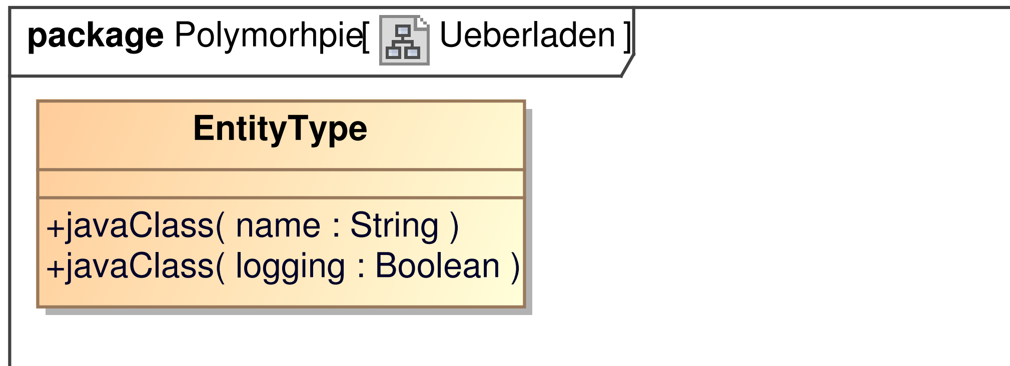


Abbildung 4: Metamodell mit Polymorphie durch Überladen

der beiden javaClass-Templates expandiert wird, kann schon vor dem Generatordurchlauf bestimmt werden, welches der beiden Templates expandiert werden muss bzw. ob der EXPAND-Befehl valide im Sinne des Metamodells ist, ob also überhaupt ein Template mit dem Namen und der geforderten Signatur existiert.

Im Fall von Abbildung 4 und Listing 2 liegt ein Metamodell vor, bei dem für ein Metamodellelement *EntityType* zwei Templates bzw. Operationen mit demselben Bezeichner definiert sind. Da beide Templates bzgl. ihrer Parameter unterschiedliche Signaturen aufweisen, wird das erste Template durch das zweite überladen. Die beiden Templates können nur durch ihren Typparameter unterschieden werden.

```
1  «IMPORT metamodel»
2  «EXTENSION my::ExtensionFile»
3  «DEFINE javaClass(uml:String name) FOR EntityType»
4          // Implementierung für EntityType mit Verwendung des Namen
5  «ENDDEFINE»
6
7  «DEFINE javaClass(uml:Boolean logging) FOR EntityType»
8          // Implementierung für EntityType mit Beachtung von Logging
9  «ENDDEFINE»
```

Listing 2: Polymorphie durch Überladen in Xpand

inklusionsbasierte Polymorphie: In EMF können mit dem Attribut *eSuperTypes* der Metametaklasse *EClass* Mehrfachvererbungstrukturen auf der Metamodellebene M2 definiert werden. Im Fall einer Vererbungsstruktur im Metamodell kann in Xpand inklusionsbasierte Polymorphie nach Cardelli u. Wegner [13] genutzt werden, falls Templates als eine funktionale Erweiterung ihrer Metaklasse im Sinne einer Methodendefinition interpretiert werden. Bei inklusionsbasierter Polymorphie kann eine Methode sowohl für Instanzen einer Klasse aufgerufen werden, in der sie definiert wurde, als auch für Unterklasseninstanzen dieser Klasse.

Durch die Mehrfachvererbungstrukturen in EMF kann ein Template auch für die direkten und indirekten Untermetaklassen einer Metaklasse expandiert werden. Des Weiteren ist es möglich, ein Template für Subtypen eines Typs zu redefinieren, also innerhalb der Typhierarchie zu überschreiben. Durch die Unterstützung der inklusionsbasierten Polymorphie wird während des Generierungsvorganges dasjenige Template zum Expandieren ausgewählt, bei dem der zu expandierende Subtyp dem in der Templatesignatur geforderten Typen in der Typhierarchie am nächsten ist. Grundsätzlich gilt die Regel, dass der Generator im Vererbungsbaum aufwärts sucht, bis er ein anwendbares Template findet. [42, S. 129] Diese Automatisierung der Auswahl des passenden Templates ermöglicht einen Verzicht auf if-instanceOf-Kaskaden, in denen fallweise nach Typen unterschieden werden muss, um das passende Template zu ermitteln. [45, S. 308]

Im Fall von Abbildung 5 liegt ein Metamodell vor, bei dem drei Metaklassen mit Spezialisierungsbeziehungen definiert sind. Entsprechend sind in Listing 3 zwei Templates für diese drei Typen definiert. Da in Xpand inklusionsbasierte Polymorphie genutzt werden kann, kann das Template *javaClass* sowohl für Modellelemente vom Typ *EntityType1* expandiert werden, als auch für sämtliche Modellelemente, deren Typ ein Subtyp von *EntityType1* ist. Wie in Abbildung 5 dargestellt, existiert für *EntityType3* kein spezifisches Template. Wenn während der Generierung das Template *javaClass* für ein Modellelement vom Typ *ElementType3* expandiert werden soll, so wird automatisch das in der Typhierarchie höhere Template gewählt, in diesem Fall das erste aus Listing 3. Entsprechend wird für Modellelemente vom Typ *EntityType2* das zweite spezifischere Template expandiert.

Aus der Unterstützung inklusionsbasierter Polymorphie in oAW ergibt sich eine Unbestimmtheit bei EXPAND-Befehlen. Wenn in einem Template ein EXPAND-Befehl steht, kann vor der Ausführung des Templates mit einem Modell nicht bestimmt werden, welches Template durch den EXPAND-Befehl expandiert werden muss. Somit können die Templates erst zur Laufzeit den Modellelementen zugeordnet werden, was auch *spätes Binden* genannt wird.

Die inklusionsbasierte Polymorphie ist vom Mechanismus des Überladens zu unterscheiden. Beide Ansätze werden als Polymorphie klassifiziert, können aber danach differenziert werden, ob Vererbungen des Metamodells Grundlage der Polymorphie sind und ob vor der Laufzeit das zu expandierende Template identifiziert werden kann.

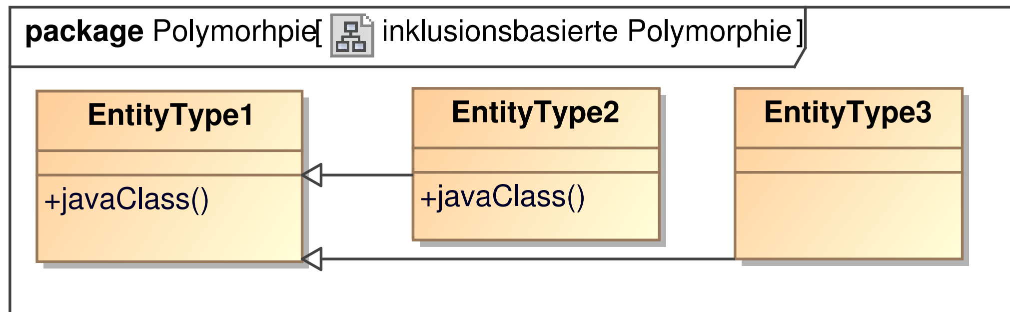


Abbildung 5: Metamodell mit Typvererbungshierarchie

```
«IMPORT metamodel»
«EXTENSION my::ExtensionFile»
«DEFINE javaClass FOR EntityType1»
         // Implementierung für EntityType 1
«ENDDEFINE»

«DEFINE javaClass FOR EntityType2»
         // Implementierung für EntityType 2
«ENDDEFINE»
```

Listing 3: inklusionsbasierte Polymorphie in Xpand

3.3.2 Schlüsselwörter

Xpand umfasst neben den bereits genannten Schlüsselwörtern die folgenden: [21]

Verzweigung: «IF <Kondition>» ... [«ELSEIF <Kondition>» ...] [«ELSE» ...] «ENDIF»
Verzweigung des Generatorflusses abhängig vom Ergebnis eines Ausdrucks bzw. einer Kondition

Schleife: «FOREACH <Ausdruck> AS <Var> [ITERATOR iterName] [SEPARATOR sep]» ... «ENDFOREACH»
Iteriert über die Elemente einer Liste oder Menge und bindet sie zwischen öffnendem und schließendem Tag an den Variablennamen <Var>. Optional kann ein Iterator verfügbar gemacht werden, der Metainformationen über die Iteration wie z. B. die aktuelle Iterationsposition bereitsstellt. Der optionale Stringparameter *sep* wird zwischen Iterationen in den generierten Text eingefügt.

geschützte Region: «PROTECT CSTART <Ausdruck> CEND <Ausdruck> ID <Ausdruck> [DISABLE]» ... «ENDPROTECT»
Schreibt eine geschützte Region in den Ausgabetext, die modifiziert werden

kann und bei einem weiteren Generatorlauf nicht überschrieben wird. Die geschützte Region wird im generierten Text durch eine Präambel und einen Schluß kenntlich gemacht, die nicht mit den Elementen der Zielsprache kollidieren dürfen. Für z. B. Java bietet es sich an, die Java-Kommentarzeichen /* und */ als Werte von CSTART und CEND zu benutzen. Die ID identifiziert die geschützte Region über Generatorläufe hinweg und muss templateübergreifend einzigartig und konstant sein. Damit der Generator nach geschützten Regionen innerhalb einer Xpand-Datei sucht, muss im Workflow mit dem Tag *prSrcPaths* eine entsprechende Verzeichnisdirektive gesetzt werden. Durch Hinzufügen des optionalen Parameters DISABLE wird der Inhalt der geschützten Region gelöscht und mit dem im Template vorgegebenen Standardwert überschrieben.

Variablendefinition: «LET <Ausdruck> AS <Var>» ... «ENDLET»
Definiert eine lokale Variable, die zwischen dem öffnenden und schließenden Tag verfügbar ist. Der Ausdruck wird nur einmal evaluiert, unabhängig davon, wie häufig der Wert der Variable abgefragt wird. Durch einen zweiten LET-Ausdruck kann der Wert einer Variable modifiziert werden.

Kommentar: «REM» ... «ENDREM»
Ein Kommentar, der keine Rem-Tags enthalten darf und somit nicht geschachtelt werden kann.

Abbruch: «ERROR <Ausdruck>»
Bricht die Ausführung des Workflows mit einer Fehlermeldung ab.

Generell ist Xpand umfangreich genug, um verschiedenste Zielsprachen abbilden zu können. Auch anspruchsvollere Aufgaben wie das Schreiben von zwei Dateien für ein Modellelement können erfüllt werden, indem in einem Template zwei FILE-Tags gesetzt werden. Dies ist z. B. notwendig, wenn aus Klassen in einem UML-Modell Quelltext in C++ erzeugt werden soll, da bei C++ die Schnittstelle einer Klasse in einer anderen Datei als die Implementierung gespeichert werden kann. Eine Einschränkung ist, dass zur Verwendung von geschützten Regionen in der Zielsprache Sprachelemente zum Anlegen von Kommentaren existieren müssen.

3.4 Xtend

Xtend ist eine im Umfang von oAW enthaltene funktionale Sprache, mit der Elemente eines Metamodells um Operationen erweitert werden können, die während

der Generierung ausgeführt werden können. In Xtend-Funktionen kann z. B. deklarativ formuliert werden, wie Werte berechnet werden sollen oder wie über die Elemente eines Eingabemodells navigiert wird. Xtend-Funktionen werden mit dem Zustand des Eingabemodells parametrisiert, innerhalb einer Xtend-Funktion ist es aber nicht möglich, Zustände in Form von Variablen zu definieren. Mit Xtend können außerdem Funktionen für die Modell-zu-Modell-Transformation formuliert werden, die aus Eingabemodellen Ausgabemodelle erzeugen.

Um Templates für verschiedene Zielsprachen bzw. -plattformen wiederverwenden zu können, sollten sie möglichst wenig spezifisch hinsichtlich einer bestimmten Zielsprache sein, die spezifischen Aspekte also aus den Templates herausgenommen werden. [45, S. 309] Außerdem ist es sinnvoll, wiederholte komplexe Funktionsaufrufe auf Metamodellelementen in Funktionen zusammenzufassen und zu benennen.

Ein Beispiel für ein Template mit geringem Abstraktionsgrad ist in Listing 4 formuliert, das Modellelemente in Klassendefinitionen transformiert. Bei der Transformation eines Modellelementnamens in einen Klassennamen ist es z. B. für die Zielsprache Java konventionell, den Modellelementnamen mit einem Majuskel zu beginnen. Im Template würde an der entsprechenden Stelle mit der Stringmethode

```
1 «DEFINE class FOR Entity»
2   public class «this.name.toFirstUpper()»{
3     // weiterer Code
4   }
5 «ENDDEFINE»
```

Listing 4: Plattformspezifisches Template

toFirstUpper() der Anfangsbuchstabe des Namensstrings groß geschrieben. Damit ist das Template spezifisch hinsichtlich der Zielsprache und kann schlecht für andersartige Zielsprachen wiederverwendet werden.

Die Extraktion der Detailaspekte der Zielsprache wird durch Xtend ermöglicht, indem diese Aspekte in Xtend-Funktionen bestimmt werden.

```
1 «EXTENSION Java»
2 «DEFINE class FOR Entity»
3   public class «getClassName(this.name)»{
4     // weiterer Code
5   }
6 «ENDDEFINE»
```

Listing 5: Bereinigtes Template

Die Extension wird in einer separaten Datei Java.ext definiert und enthält die

extrahierten Operationen. Für einen String aus dem UML-Metamodell kann die Extension *getClassName(uml::String name)* z. B. folgendermaßen definiert werden:

```
import uml;
extension WeitereXtendDatei;
cached String getClassName(uml::String name) :
  name.toFirstUpper();
```

Listing 6: Funktion für Klassennamen

Die Funktionen in Xtend unterstützen analog zu Xpand statisch das Typsystem des Metamodells, dessen Namensraum durch den Befehl *import* verfügbar gemacht wird. Das verfügbare Typsystem umfasst sowohl die Typen des auf Ecore basierenden Metamodells, als auch die in Ecore vorgesehenen EDataType-Primitive, die automatisch auf Javatypen abgebildet werden. Damit ist Xtend wie Xpand generisch hinsichtlich der verwendbaren Metamodelltypsysteme und für die Transformation von DSLs geeignet. Außerdem unterstützt Xtend analog zu Xpand inklusionsbasierte Polymorphie und Überladen.

Xtend ist eine funktionale Sprache, mit der innerhalb von Funktionen sowohl iterativ als auch rekursiv weitere Funktionen aufgerufen werden können. Im Rahmen einer rekursiven Funktion wird die Abbruchbedingung mit einem ternären Operator formuliert.

```
String fullyQualifiedName(NamedElement n) : n.parent == null ? n.name :
          fullyQualifiedName(n.parent)+'::'+n.name;
```

Listing 7: rekursive Funktion mit ternärem Ausdruck [21]

In Xtend kann eine Funktion auf den Elementen einer Liste oder Menge ausgeführt werden. Falls die Funktion eine Liste als Parameter erwartet, wird sie mit der übergebenen Liste aufgerufen. Falls die Funktion dagegen ein einzelnes Element als Parameter erwartet, wird sie automatisch für sämtliche in der Liste enthaltenen Elemente ausgeführt.

```
uml::String getFirstAttribute(uml::Package p) :
  p.ownedElement.typeSelect(uml::Class).getAllAttributes().first().name;
```

Listing 8: automatische Verarbeitung von Collections

In Listing 8 ist eine Funktion auf dem UML-Metamodell definiert, die für ein gegebenes Package die Liste der enthaltenen Elemente ermittelt, diese auf die Elemente mit dem UML-Typ Class reduziert, für alle Klassen in der reduzierten Liste

die Attribute in diesen Klassen ermittelt, das erste gefundene Attribut auswählt und den Namen dieses Attributes als String zurückgibt. Da die Funktion *getAllAttributes()* als Parameter ein Element vom UML-Typ *Classifier* voraussetzt, aber von der Funktione *typeSelect()* eine Liste erhält, greift Xtend ein und führt sie automatisch iterativ auf den Elementen der Liste aus.

Durch den Befehl *extension* können für die Xtend-Funktionen einer ext-Datei Funktionen aus anderen ext-Dateien bereitgestellt werden. Dies ermöglicht eine hierarchische Organisation der Bibliothek, indem allgemeine Funktionen zusammengefasst werden und in den spezifischeren ext-Dateien inkludiert werden. Mit dem mehrfachen Aufrufen einer Funktion für ein Modellelement wird diese jedesmal neu ausgewertet. Dies kann durch das Schlüsselwort *cached* verhindert werden, das eine einmalige Auswertung erzwingt. [21, S. 80] Durch das Schlüsselwort *private* kann die Sichtbarkeit von Funktionen auf die sie enthaltende ext-Datei begrenzt werden.

In Xtend können mittels des create-Befehls Modell-zu-Modell-Transformationen durchgeführt werden. Bei der Generierung eines Ausgabemodells mit Xtend werden die generierten Elemente des Ausgabemodells in einem Cache abgelegt und stehen als Referenz für weitere Modifikationen durch create-Befehle zur Verfügung. Elemente im Cache können innerhalb einer create-Funktion mehrmalig modifiziert werden. Im Fall von Collections im Cache bedeutet dies, dass die Reihenfolge der Elemente in der Collection von der Reihenfolge der Modifikationsaufrufe in den create-Funktionen abhängt.

Dies führt zu einer weniger strikten Umsetzung des funktionalen Programmierparadigmas, nach dem die Reihenfolge der Funktionsaufrufe irrelevant für den Wert des Ergebnisses ist. [27] [35] Ähnlich zu Monaden in Haskell [27] kann in Xtend mit dem Operator -> eine Reihenfolgebeziehung definiert werden, die den Fluß der Befehlsausführung ordnet.

4 Modellierung von Webapplikationen

4.1 Webapplikationen

Eine Webapplikation bzw. synonym Webanwendung ist nach Kappel u. a. [28]

> „[...] ein Softwaresystem, das auf der Spezifikation des World Wide Web Consortiums (W3C) beruht und Web-spezifische Ressourcen wie Inhalte und Dienste bereitstellt, die über eine Benutzerschnittstelle, den Web-Browser, verwendet werden.“

Damit steht bei *Kappel* die Standardisierung der Oberflächenelemente durch das W3C sowie die Interaktion über einen Browser im Vordergrund. Ceri [14] bezeichnet mit datenintensiven Webappliationen

> „[...] Web sites for accessing and maintainung large amounts of structured data, typically stored as records in a database management system.“

und betont damit die Datensatzorientierung. Conallen [16] sieht Webapplikationen als dynamische Webseiten, die Geschäftslogik implementieren und deren Geschäftsdaten durch Benutzer geändert werden können. Wesentlich ist, dass eine Webapplikation eine Software ist, die auf einem Webserver läuft und mit der der Benutzer mit Hilfe eines Webbrowsers interagiert.

Webapplikationen unterliegen im Vergleich zu Applikationen allgemeiner Natur insbesondere den folgenden Anforderungen: [34]

Skalierbarkeit: Durch die weltweite Erreichbarkeit des Internets stehen Webapplikationen potentiell große Nutzerkreise gegenüber. Steigende Nutzerzahlen müssen unter Gesichtspunkten der Performanz beachtet werden, z. B. soll die Software der Webapplikation durch load balancing auf mehrere Maschinen verteilt werden können.

Integrierbarkeit: Webapplikationen müssen in gegebene IT-Infrastrukturen von Organisationen integriert werden können. Durch verschiedenartige Fremdsysteme wird Heterogenität der Schnittstellen erzeugt, der durch offene Standards entgegengewirkt werden muss. Ein Beispiel dafür ist SOAP.

Sicherheit: Aus der offenen Zugänglichkeit vieler Webapplikationen im Internet resultiert eine höhere Gefährdung bei Sicherheitslücken. Zur Schadensbegrenzung sollten Sicherheitsmechanismen genutzt werden, wie z. B. das gesaltete Hashen von Passwörtern anstatt der Klartextspeicherung, um Zugriffe auf die

Klartextpasswörter bei einer Kompromittierung der Benutzerdaten zu verhindern.

Bedienbarkeit: Der große Nutzerkreis und die schwache organisationelle Bindung der Benutzer führt zu anonymen und sporadischen Benutzerzugriffen. Die Benutzerschnittstelle soll deshalb intuitiv zu bedienen sein.

Verteiltheit: Webapplikationen sollen organisationsübergreifend und verteilt realisiert werden können, um unternehmensübergreifende Geschäftsprozesse abzubilden.

Seit der Entstehung des World Wide Webs wurden die anfänglich rein dokumentenzentrierten Webapplikationen erweitert um Aspekte der Interaktion, Kollaboration, Workfloworientierung und semantischen Auszeichnung von Inhalten. [28] Damit ging eine Steigerung der Komplexität einher, die strukturierte Methoden der Entwicklung und Anwendung von Webapplikationen notwendig macht.

4.2 Web Engineering

Die zunehmende Komplexität und Relevanz Web-basierter Systeme macht es notwendig, ein ingenieurmäßiges Vorgehen bei der Entwicklung von Web-basierten Systemen zu wählen, um den Entwicklungsprozess plan- und wiederholbar zu gestalten. Die Problematik scheiternder Softwareprojekte ist vor dem Hintergrund der Softwarekrise zu sehen, die seit 1969 [40] diskutiert wird. Ursächlich für die Krise ist nach Dijkstra [18], dass die zu programmierenden Maschinen mächtiger geworden sind und mit der Mächtigkeit der Maschinen auch die Probleme der Programmierung gewachsen sind. Als Folge der Krise wurde die Herangehensweise bei der Entwicklung und Anwendung von Software weiterentwickelt und der Begriff der Softwaretechnik bzw. synonym des Software Engineering geprägt. Nach Balzert [9] ist Softwaretechnik die

> „[...] zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Software-Systemen. Zielorientiert bedeutet die Berücksichtigung z. B. von Kosten, Zeit, Qualität."

Aus dem Software Engineering ist das Web Engineering hervorgegangen, bei dem die Methoden der Softwaretechnik auf die Entwicklung und Anwendung von komplexen web-basierten Webapplikationen bzw. synonym Web-Systemen angewendet werden. Nach Dumke [19, S. 47] ist

> „Web Engineering [...] die methodenbasierte, werkzeugunterstützte, quantifizierte, standardgerechte, erfahrungsnutzende und Community-bezogene Entwicklung und Wartung von webbasierten Softwaresystemen.“

Murugesan u. a. [39] definieren Web Engineering als

> „[...] establishment and use of sound scientific, engineering and management principles and disciplined and systematic approaches to the successful development, deployment and maintenance of high quality Web-based systems and applications.“

Wesentlich für das Software Engineering ist die Beherrschung der Komplexität und die Abstraktion in Form von Modellen. Analog sollen im Web Engineering die Struktur- und Abstraktionsmechanismen des Software Engineering wie z. B. Vorgehensmodelle und Modellierungssprachen genutzt werden, also der künstlerische Schaffungsprozess systematisiert werden. Im Unterschied zum Software Engineering müssen aber Spezifika von Web-Systemen beachtet werden: [39]

1. Die meisten Web-Systeme sind dokumentenorientiert und bestehen aus statischen oder dynamisch generierten Webseiten mit textuellen Inhalten. Benutzereingaben und -anfragen werden als HTTP-POST oder HTTP-GET Anfragen verarbeitet und resultieren in einem Wechsel oder Neuladen von Webseiten. Mit Asynchronous JavaScript and XML (Ajax) können zwar innerhalb einer HTML-Seite Inhalte durch HTTP-Anfragen vom Server asynchron nachgeladen werden ohne die Webseite komplett neu laden zu müssen, sodass der Look and Feel klassischer GUI-basierter Anwendungen reproduziert werden kann. Dennoch werden gewöhnlich statische Strukturelemente wie z. B. Tabellen nicht durch Ajax innerhalb des DOM-Tree einer Webseite generiert, sondern nur die Inhalte der Strukturelemente wie z. B. Tabellenzeilen dynamisch per Ajax synchronisiert.

2. Der Look and Feel von Web-Systemen ist grafikorientiert und durch Multimediainhalte angereichert. Im Gegensatz zu GUI-Applikationen haben künstlerische Aspekte und kreative Designs einen hohen Stellenwert. Für das Web-Engineering müssen die Konzepte der künstlerisch arbeitenden Designer mit Konzepten ingenieursmäßig arbeitender Softwareentwickler strukturiert verbunden werden. Dies wird durch aktuelle Umgebungen wie z. B. JavaServer Faces unterstützt, indem Webseiten und Applikationsquelltext getrennt entwickelt werden und durch einheitliche Schnittstellen verknüpft werden.

3. Häufig sind Web-Systeme Mehrbenutzersysteme mit verschiedenen Typen und Rollengruppen von Benutzern. Die Präsentation und der Funktionsumfang von Web-Systemen werden anhand der Rechte und der Profile der Benutzer personalisiert.

4. Der Verteilprozess von Web-Systemen unterscheidet sich von Software. Modifikationen an Web-Systemen können zentralisiert auf dem Webserver vorgenommen werden, sodass selbst Änderungen auf der Präsentationsebene nicht zu Installationen auf der Clientseite führen.

5. Entwickler von Web-Systemen haben häufig keinen originär technischen Hintergrund, sondern kommen aus einem künstlerischen Umfeld. Dies schlägt sich in den verwendeten Methoden und der Wahrnehmung nieder, indem Präsentationsaspekte im Vordergrund stehen.

4.3 Evaluation von Webmodellierungssprachen

Für das Web Engineering existieren verschiedene methodische Ansätze wie z. B. UML-based Web Engineering (UWE). Die Methoden umfassen methodische Vorgehensweisen, Konzepte und Notationen [9] und können zur Modellierung von Webapplikationen in Modellierungssprachen zusammengefasst werden. Die Konzepte und Notationen können für die Methode in einer spezifischen Modellierungssprache definiert sein oder aber es werden gegebene Sprachen wie z. B. UML verwendet und erweitert.

Die Modellierungssprache kann, wie in Abbildung 6 dargestellt, strukturell durch ein Metamodell spezifiziert werden, das damit ein sprachbasiertes Metamodell für die Modelle der Modellierungssprache ist. Analog kann das Vorgehen zur Erstellung von Modellen durch ein Vorgehensmodell spezifiziert werden. Das Vorgehensmodell und das sprachbasierte Metamodell sind konstituierend für die Methode.

Für die modellgetriebene Entwicklung von Webapplikationen ist insbesondere die Wahl einer geeigneten Modellierungssprache und damit eines sprachbasierten Metamodells wichtig, um die üblichen Konzepte von Webapplikationen optimal in Modellen und Transformationsdefinitionen abbilden zu können.

In diesem Kapitel werden einige Anforderungen an Modellierungssprachen zur Modellierung von Webapplikationen definiert. Anschließend wird ein kurzer Überblick über verfügbare Sprachen gegeben. Diejenigen Sprachen, die für die modellgetriebene Entwicklung einer prototypischen Webapplikation im Rahmen dieser Arbeit von Relevanz sind, werden in den nachfolgenden Kapiteln genauer erläutert.

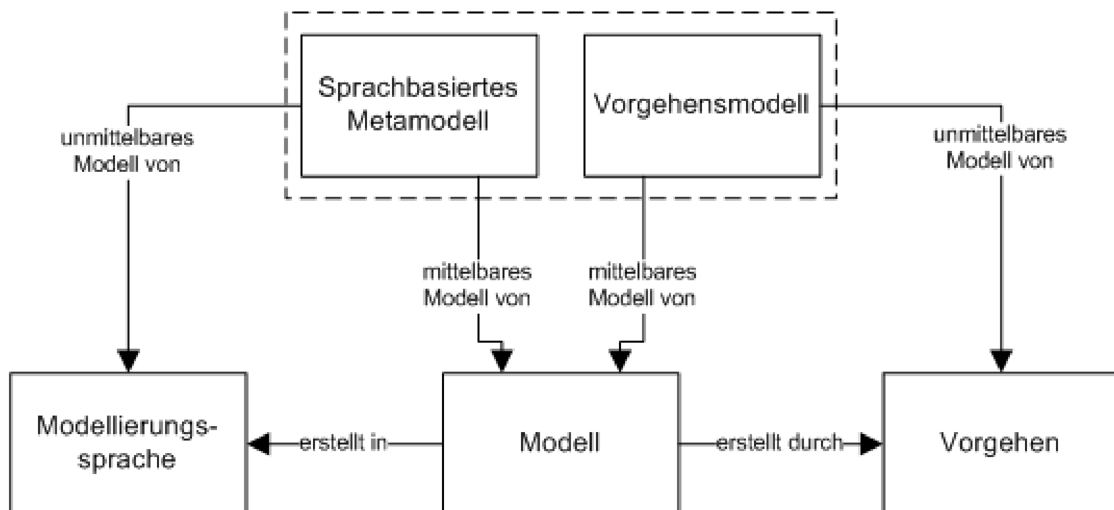


Abbildung 6: Methodenbegriff [25]

Die Auswahl erfolgt anhand der folgenden Kriterien:

enthaltene Modelltypen: Webapplikationen besitzen im Gegensatz zu GUIs von Standardapplikationen hypertextuellen Charakter und Navigationsstrukturen. Deshalb wird bei der Modellierung von Webapplikationen anders als bei der Modellierung von GUI-basierter Software zwischen Inhalts- und Datenmodellen, Präsentationsmodellen und zusätzlich Navigationsmodellen bzw. Hypertextmodellen unterschieden. [28, S. 52] *Inhaltsmodelle* werden untergliedert in Datenmodelle, mit denen Datenstrukturen erfasst werden, und Klassenmodelle der Geschäftslogik, in denen Anwendungslogik modelliert wird. In *Navigationsmodellen* werden Inhalte und Prozesse als Knoten und Verweise zwischen den Knoten dargestellt. In *Präsentationsmodellen* werden auf Basis des Navigationsmodells Gestaltungen der Benutzerschnittstelle erfasst. Durch die Trennung der Belange (separation of concerns) können für ein Datenmodell mehrere Zugangsmöglichkeiten in Form von verschiedenen Navigationsmodellen und für ein Navigationsmodell mehrere Präsentationsformen definiert werden. Z. B. kann eine Präsentation für klassische Browser und eine alternative reduzierte Form für mobile Geräte auf dem selben Navigationsmodell basieren. Benutzerdaten können den Datenmodellen zugeordnet werden, einige Ansätze sehen sie aber als Metadaten, die gesondert modelliert werden. Von Relevanz ist, dass die Sprachen die Erstellung von Inhalts-, Navigations- und Präsentationsmodellen ermöglichen.

Toolunterstützung: Die Modellierung des Prototypen im Rahmen dieser Arbeit soll durch ein graphisches UML-Werkzeug unterstützt werden. Für UML spricht,

dass sie als Standardsprache im Software Engineering bereits etabliert ist und generisch die geforderten Modelltypen enthält. Die Semantik von UML sollte Modellierern und Entwicklern bekannt sein und kann durch UML-Profile verfeinert werden. Des Weiteren wird UML im Gegensatz zu Ecore bereits großflächig durch etablierte CASE-Werkzeuge unterstützt. Die auszuwählenden Modellierungssprachen müssen für eine Unterstützung von UML das UML-Metamodell in Form von UML-Profilen erweitern. Bei dem alternativen Vorgehen, die Modellierung in einer domänenspezifischen Ecore-basierten Sprache vorzunehmen, würde die Dokumentation der Sprachsemantik umfangreicher und es müsste ein grafisches Modellierungswerkzeug erstellt werden. Dies kann zwar durch das in EMF enthaltene Graphical Modeling Framework (GMF) geschehen, ist aber zu aufwendig für den Umfang dieser Arbeit.

Abstraktionsgrad: Die Erhöhung des Abstraktionsgrades von Modellen soll die Reduktion der Modellkomplexität ermöglichen, führt aber auch automatisch zu Informationsverlusten. Zur Generierung von Quelltext müssen diese Informationsverluste entweder durch konkretisierende, nichtparametisierte Ausdrücke in den Transformationsdefinitionen oder durch manuelle Eingriffe ausgeglichen werden. Die auszuwählenden Sprachen müssen einen akzeptablen Kompromiss zwischen Abstraktion und Explikation darstellen.

Metaanforderungen: Die Sprache bzw. das Metamodell muss durch Begleittexte erläutert werden und mit Beispielmodellen dokumentiert sein. Zudem sollte die Sprache herstellerunabhängig sein, kontinuierlich weiterentwickelt werden und die Dokumentation frei verfügbar sein.

Im Folgenden werden einige etablierte Modellierungssprachen für Webapplikationen erläutert.

4.3.1 Web Application Extension

Die *Web Application Extension* (WAE) ist eine UML-basierte Modellierungssprache, deren Publikation erstmals [16] im Jahr 1999 und in einer überarbeiteten Version [17] im Jahr 2003 erfolgte. Das Metamodell von WAE ist als UML-Profil verfügbar [17] und enthält Elemente für die Modellierung von Inhalts- und Präsentationsmodellen. Navigationsstrukturen werden nicht explizit in Navigationsmodellen modelliert, sondern mit Pages und Links im Präsentationsmodell konstruiert. Dem Anspruch der separation of concerns wird also nicht vollständig entsprochen.

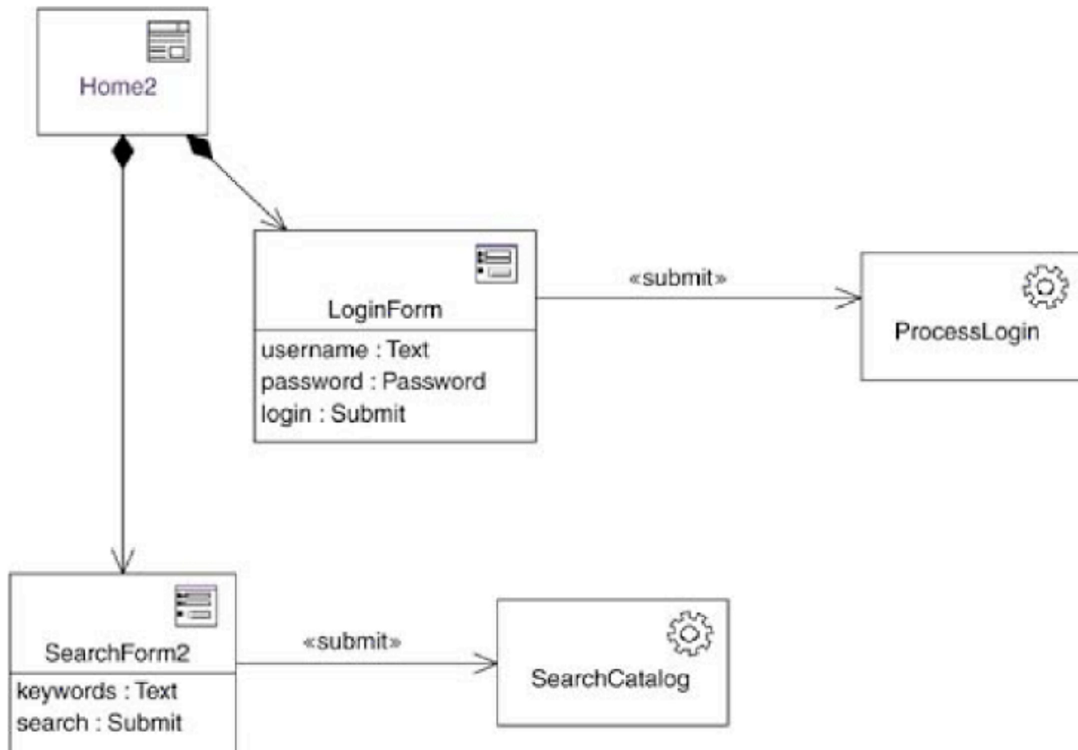


Abbildung 7: Beispielmodell in WAE [17]

Die Sprachelemente für Präsentationsmodelle sind Stereotypen und entsprechen HTML-Tags. Für WAE ist vorgesehen, dass das Basisprofil durch weitere Profile plattformspezifisch und herstellerabhängig erweitert werden kann, im Kern aber nicht herstellerspezifisch ist. Im Umfang der Dokumentation ist ein Profil für JavaServer Pages (JSP) enthalten, die Erweiterung durch alternative Profile für z. B. Active Server Pages (ASP) und PHP soll zumindest ermöglicht werden.

Beispielsweise kann eine Webseite mit zwei Formularen modelliert werden, indem wie in Abbildung 7 dargestellt zwei HTML-Formularklassen über Komposita der Clientseitenklasse zugeordnet werden. Die mit dem Submit-Button verbundene Aktion wird durch stereotypisierte Assoziationen zu Klassen für dynamische Serverwebseiten ausgedrückt. Die Stereotypen *HTML Form*, *Client Page*, *Server Page* und *submit* sind Sprachelemente von WAE. Die Stereotypen für Klassen werden im Diagramm nicht wie bei UML üblich geklammert im Klassenkopf visualisiert, sondern durch entsprechende Symbole. Aus der Orientierung an HTML folgt ein geringer Abstraktionsgrad. Es existiert z. B. kein Sprachelement für die Suche nach Datensätzen, sondern nur das Element *Form*, das auch für die Eingabe von Datensätzen genutzt wird. Es kann also im Modell nicht zwischen der Eingabe von Datensätzen und der Suche nach diesen unterschieden werden, sodass diese Semantik nicht expli-

ziert werden kann. Durch die Deckungsgleichheit von HTML- bzw. JSP-Tags und Stereotypen sind die Modelle zudem plattformspezifisch.

4.3.2 Web Modeling Language

Die *Web Modeling Language* (WebML) ist eine domänenspezifische Modellierungssprache für datenorientierte Webapplikationen, die im Jahr 2000 veröffentlicht [15] wurde. WebML enthält Sprachelemente für Navigations-, Präsentations- und Benutzermodelle. Für die Datenmodellierung wird die Verwendung alternativer Modelltypen wie z. B. E/R-Modelle oder Klassendiagramme empfohlen. Die Navigationsstrukturen werden durch Units ausgedrückt, die durch Links verbunden sind. WebML unterscheidet Units nach ihrer Funktionalität. Unter anderem sind Units für Indexe, Suchmasken, Dateneingaben [11] und Datenausgaben vorgesehen. Die Daten des Datenmodells werden den Units zugeordnet und durch deren Funktionalität modifiziert. Im Präsentationmodell wird das Erscheinungsbild der Elemente des Navigationsmodells in Form von XML-basierten Stylesheets festgelegt. Die Stylesheets beziehen sich entweder auf sämtliche Seiten oder nur auf einzelne Seiten bestimmter Units. Mit Webratio existiert ein CASE-Tool für WebML, mit dem aus den Modellen und Stylesheets Code für JSP und Jakarta Struts erzeugt werden kann.

Beispielsweise kann für eine Webapplikation zur Verwaltung von Musikalben wie in Abbildung 8 dargestellt ein kaskadierter Index für die Suche nach Alben modelliert werden. Mit einer IndexUnit wird eine Listendarstellung für die Auswahl von Datensätzen aus der Datenquelle für Künstler modelliert. Die Datensätze von Künstlern können einzeln betrachtet werden und enthalten einen Index mit den Alben des Künstlers. Datensätze von Alben können in dem Index ausgewählt und einzeln angezeigt werden.

Für WebML existiert neben dem domänenspezifischen und DTD-basierten Metamodell ein UML-basiertes Profil [38] [37], in dem die WebML-Sprachelemente für Daten- und Navigationsmodelle durch Stereotypen repräsentiert werden. Für Präsentationsmodelle ist zwar ein Profil mit dem Namen *presentationview* vorgesehen, dieses ist aber nicht weiter in UML dokumentiert, sondern nur in Form einer DTD. Die Präsentations-DTD beschreibt Sprachelemente für die Formatierung von Seiten, Links, Styleeigenschaften und Datenfeldern. Es wird also nur die Darstellung der Sprachelemente des datenzentrierten Navigationsmodells definiert, mit dem Präsentationsmodell können einzelne Units aber nicht in mehrere Webseiten unterteilt werden oder die Formatierung von Texten vorgegeben werden.

Der Abstraktionsgrad von WebML ist als hoch anzusehen [38, S. 19], da anstatt

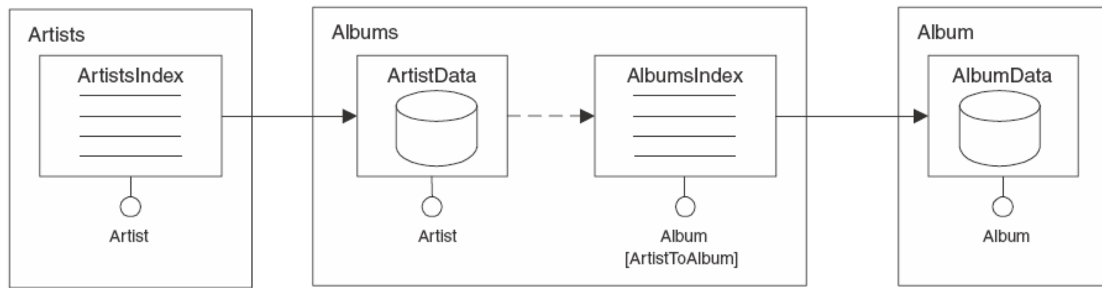


Abbildung 8: Beispielmodell in WebML [14]

von Webseiten Funktionalität modelliert wird. Die Funktionalität ist zudem stark datenorientiert und an CRUD-Operationen ausgerichtet. Dies schlägt sich auch in der Definition von Webapplikationen nach Ceri in Kapitel 4.1 nieder.

4.3.3 UML-based Web Engineering

UML-based Web Engineering ist eine umfangreich in mehreren Arbeiten [31] [33] [32] [30] [26] vorgestellte Methode zur Entwicklung von Webapplikationen. Der Fokus von UWE liegt darauf, den strukturierten und durch CASE-Tools gestützten Entwicklungprozess im Rahmen des Software Engineering auch bei der Entwicklung von Webapplikationen umzusetzen.

Bestandteil der Methode ist eine Modellierungssprache, mit der Daten einer Domäne, sowie Navigations-, Präsentations- und Interaktionsaspekte modelliert werden können. Das Metamodell der UWE-Sprache ist ein UML-Profil und enthält Spezialisierungen der UML-Metamodellelemente in Form von Stereotypen, die in Profilpaketen strukturiert sind. Attribute von Metamodellelementen und Beziehungen zwischen diesen werden als Tagged Values bzw. Metaassoziationen erfasst, sodass sie zum Zweck der Generierung in Templates referenziert werden können.

Die Profile des UWE-Profilpaketes bauen hierarchisch aufeinander auf und wurden im Laufe der Entwicklung von UWE restrukturiert und auf wesentliche Elemente reduziert. Im Folgenden wird der aktuelle Forschungsstand nach Koch [31] beschrieben.

Das UWE-Metamodell besteht aus den Profilen *content*, *navigation*, *presentation*, *process* und *requirements*. Die Elemente des Profils *content* dienen der Modellierung von statischen Datenstrukturen der Webapplikation. Dies umfasst allgemeine Entitätsklassen, Klassen für Benutzer und Rollen, Geschäftsklassen und Klassen für Systemmetadaten. Das Profil *navigation* dient der Beschreibung der Navigationsmöglichkeiten in der Webapplikation, die als Abfolge von Knoten und Kanten aufgefasst wird. Das Profil navigation wird durch das Profil *process* verfeinert, mit

Abbildung 9: Ausschnitt aus dem UWE-Metamodell

dem anstatt von Klassendiagrammen auch Aktivitätsdiagramme als Diagrammtyp zur Modellierung von Navigationsstrukturen verwendet werden können. Das Profil *presentation* umfasst Elemente für die Modellierung der Benutzerschnittstelle, die als die Repräsentation der Navigation aufgefasst wird. Dabei können für ein Navigationsmodell mehrere Präsentationsmodelle formuliert werden, um unterschiedliche Benutzerschnittstellen für verschiedene Geräteplattformen wie z. B. Terminalrechner und Mobilgeräte definieren zu können. Die Profile content, navigation, process und presentation bauen aufeinander auf und hängen entsprechend voneinander ab. Das Profil *requirements* dient der Modellierung von Anforderungen in Form von stereotypisierten Use-Case-Diagrammen.

Im Folgenden werden die in Abbildung 9 dargestellten Profile content, navigation und presentation erläutert. Die restlichen Profile besitzen für die Generierung eines Prototyps im Rahmen dieser Arbeit keine Relevanz.

Profil Content

Das Metamodell von UWE für Datenmodelle entspricht dem Metamodell für UML-Klassenmodelle. Die Elemente *Class* und *Property* des UML-Metamodells werden im Profil *content* durch die Definition von Stereotypen spezialisiert. Bei der Erstellung eines UWE-Datenmodells werden diese Stereotypen verwendet, um Typsicherheit bei Assoziationen mit anderen UWE-Modellelementen herzustellen.

Andere UML-Elemente wie z. B. das Element *Operation* werden in UWE nicht spezialisiert. Da Operationen im Konzeptmodell deklariert werden können, ist es möglich, im Contentmodell innerhalb derselben Klasse nicht nur Datenhaltungsattribute und Datenzugriffsoperationen zu deklarieren, sondern auch Geschäftslogik. Falls aufgrund der Wahl der Systemarchitektur wie z. B. der 4-Schichtenarchitektur die Trennung von Datenhaltung und Geschäftslogik gewünscht wird, liegt es am Modellierer, dies entsprechend zu modellieren. Formal wird dies vom Metamodell jedenfalls nicht gefordert.

Profil Navigation

Das Profil *Navigation* stellt Metamodellelemente zur Modellierung der Navigationsstruktur bereit, mit denen die Abfolge der Seiten der Webapplikation abgebildet werden kann. Das Metamodell des Navigationsmodells wird weitestgehend von Kraus u. Koch [33] und Koch [31] beschrieben.

Wesentliche Elemente des Navigationsprofils sind die abstrakten Stereotypen *Node* und *Link*, welche Spezialisierungen der Sprachelemente *Class* und *Association* aus dem UML-Klassenmodellkernel sind. Ausgehend von den Navigationsknoten geben

die Links die möglichen Navigationspfade im System an und münden in einen oder mehrere Navigationsknoten. Die Möglichkeit der Angabe mehrerer Zielknoten stellt eine Abweichung von der Spezifikation von Hyperlinks in HTML dar, die nur eine Ressource adressieren können. Eine vorstellbare Abbildung in HTML ist, dass mit dem Klick auf einen Link sowohl die im Anchor stehende Seite geladen wird als auch mit dem Klickereignis des DOM-Tree der Seite weitere Seiten durch Javascript geladen werden. Als instanziierbare Knotentypen stehen im Navigationsmodell die Stereotypen NavigationClass, Menu und AccessPrimitive zur Verfügung:

NavigationClass ermöglicht eine datensatzorientierte Navigation im System, indem jeder Navigationsklasse genau eine Contentklasse zugeordnet wird. Die Navigationsklasse verbindet das Contentmodell mit dem Navigationsmodell über die Zuordnung von *Navigationsattributen* zu *Contentattributen*. Wenn eine Contentklasse z. B. Kundendatensätze beschreibt und diese Contentklasse einer Navigationsklasse zugeordnet ist, dann ist die Navigationsklasse ein Knoten zur Visualisierung oder Bearbeitung der Kundendatensätze. Anhand der Navigationsattribute kann dem Navigationsknoten eine geordnete Untermenge von Contentattributen zugeordnet werden. Falls Contentattribute wie z. B. IDs verborgen werden sollen, kann dies erfolgen, indem sie keinem Navigationsattribut zugeordnet werden.

Die AccessPrimitive *Index*, *Query* und *GuidedTour* dienen im Gegensatz zur *NavigationClass* dem Auslesen bzw. Auflisten mehrerer Datensätze.

Im Fall des *Index* wird durch die Assoziation zwischen *Index* und *NavigationProperty* festgelegt, welche Navigationsattribute zur Benennung der Elemente des Index auf der Präsentationsebene zur Verfügung stehen. Z. B. kann ausgedrückt werden, dass bei einem Index für Personen die Listenelemente durch den Vor- und Nachnamen beschrieben werden.

Im Fall der *Query* kann angegeben werden, welche Felder die Suchmaske für die Suche nach Datensätzen besitzen soll. Wenn z. B. ein Navigationsattribut mit der Bezeichnung *Name* der Query zugeordnet ist, kann es auf der Webseite als Feld des Suchformulars angezeigt werden.

Profil Presentation

Das Profil *Presentation* enthält Metamodellelemente zur Modellierung von Webseiten auf Basis eines Navigationsmodells. Der Stereotyp *Page* entspricht einer Seite der Webapplikation und besteht aus Benutzerschnittstellenelementen, die vom abstrakten Stereotyp *UIElement* abgeleitet sind. Die Elemente sind z. B. Texte, Bilder, Formulare und Listen und sind in Anlehnung an HTML strukturiert. Seiten können über den Stereotyp *PresentationGroup* gruppiert werden. Die Gruppierungsmöglich-

keit ist in vollem Umfang nur in der ursprünglichen Metamodellversion enthalten [33], ist aber trotzdem in Abbildung 9 eingezeichnet, weil sie zweckmässig ist.

Präsentationsgruppen können weitere Gruppen enthalten, sodass Gruppenhierarchien gebildet werden können. Seiten bzw. Präsentationsklassen können auf jeder Ebene der Gruppenhierarchie eingefügt werden. Dies ermöglicht die Strukturierung von Präsentationsklassen anhand ihrer Bedeutung in der Gruppenhierarchie. Z. B. kann eine Gruppe mit Seiten für die Verwaltung von Kundenadressen eine Untergruppe in einer allgemeineren Gruppe für die Verwaltung von Kundendaten sein. Allgemeine Seiten zur Auflistung von Kunden würden dabei auf einer höheren Gruppenebene liegen als Seiten zur Prüfung einer Adresse.

Die Menge der Elemente der Benutzerschnittstelle weist eine Ähnlichkeit mit den in HTML enthaltenen Tags auf und ist in Containerelemente und atomare Elemente untergliedert. Im Gegensatz zu Gruppenelementen sind Einzelelemente einfach strukturiert und können keine Subelemente enthalten. Beispiele dafür sind Bilder und Textfelder. Containerelemente können sowohl Container- als auch Einzelelemente enthalten. Z. B. kann das Containerelement *Form* Eingabefelder, Textfelder und Buttons enthalten.

Jeder Navigationsknoten im Navigationsmodell kann einen oder mehrere Präsentationsklassen referenzieren, die ihn z. B. für verschiedene Arten von Endgeräten visualisieren. Navigationslinks werden auf der Präsentationsebene z. B. durch Hyperlinks oder Aktionen von Formularbuttons repräsentiert.

Präsentationselementen können als Datengrundlage Präsentationsattribute zugeordnet werden, die wiederum Navigationsattribute referenzieren. Damit können z. B. Formulare mit Werten bestehender Datensätze befüllt werden oder es kann der Titel von Webseiten mit Datensatzwerten belegt werden. Links im Navigationsmodell werden auf Hyperlinks bzw. an Buttons gebundene Aktionsanweisungen in HTML-Formularen abgebildet.

Bewertung

Vorteilhaft an UWE ist die konsequente Trennung der Belange (separation of concerns), die eine strukturierte Modellierung der Webapplikation durch die Unterstützung von Daten-, Navigations- und Präsentationsmodellen ermöglicht. Des Weiteren wird auch der Entwicklungsprozesses strukturiert, indem die Anforderungsanalyse, die Entwurfsphase und die Realisierungsphase methodisch durch UWE unterstützt werden.

Probleme bei der Nutzung von UWE für die MDD ergeben sich im Detail. So können durch die kontextfreie Gruppierung von UIElementen im Paket Presenta-

tion Strukturen modelliert werden, die nicht sinnvoll implementiert bzw. generiert werden können. Ein Beispiel dafür ist das Element *TextInput*, das nur im Kontext des Elements *Form* Sinn ergibt.

Auch wäre es wünschenswert, explizit eine getrennte Stereotypisierung von Klassen für die Datenhaltung und Klassen für die Geschäftslogik vornehmen zu können. Die explizite Kennzeichnung von Geschäftsklassen würde den Informationsgehalt des Modells erhöhen und die Erstellung von typsicheren Transformationsdefinitionen vereinfachen.

Der zentrale Kritikpunkt an UWE betrifft das Präsentationsprofil, dessen Sprachelemente einen geringen Abstraktionsgrad besitzen und somit sowohl die Modellierung als auch die Entwicklung von Templates verkomplizieren.

5 Web Application Modeling Language

Die Web Application Modeling Language (WAML) ist eine im Rahmen dieser Arbeit erstellte Sprache zur Modellierung von Webapplikationen. Sie soll als Basis für die Implementierung einer prototypischen Webapplikation dienen und stellt einen Kompromiss zwischen den bereits vorgestellten Sprachen dar. Das Metamodell von WAML ähnelt grob dem Metamodell von UWE, im Unterschied zu UWE sind aber die Sprachelemente für Präsentationsaspekte grobgranularer und ähneln den Units aus WebML. Außerdem werden Navigationsmodelle in WAML in Form von stereotypisierten Aktivitätsdiagrammen formuliert.

Das in Abbildung 10 dargestellte Metamodell von WAML ist ähnlich zu UWE in drei Schichten in Form der drei Profile *content*, *control* und *view* untergliedert, die hierarchisch voneinander abhängen. In diesen werden im Sinne der separation of concerns die Aspekte der Datenhaltung und Geschäftslogik, die Aspekte der Navigation und operationalen Funktionalität und die Aspekte der Darstellungsfunktionalität erfasst. Die Profilaufteilung von WAML orientiert sich an der Model-View-Controller-Architektur (MVC) [46], in der GUI-Aspekte in das Modell, den Controller und die View unterteilt werden. Bei MVC beinhaltet das Modell die Anwendungsdaten und Geschäftslogik. Die Views stellen in einer Schnittstelle zum Benutzer die Anwendungsdaten und Geschäftslogik dar. Der Controller steuert die Views und verarbeitet Benutzeraktionen und eingegebene Daten, indem er die entsprechenden Methoden der Geschäftslogik im Modell aufruft.

Modelle in WAML müssen so aussagekräftig sein, dass aus ihnen vollständig automatisiert Webapplikationen erzeugt werden können, ohne nach der Generierung manuelle Anpassungen vornehmen zu müssen. Die dazu bereitgestellte Ausdruckskraft wird in den folgenden Kapiteln profilweise erläutert. Dabei wird nur das Metamodell beschrieben, ein exemplarisches WAML-Modell ist in Kapitel 6.1 dargestellt.

5.1 Profil Content

Das Profil *content* enthält Stereotypen für die Modellierung von Entitäts- und Geschäftsklassen, also für die Modellierung des Backends. Entitätsklassen werden in WAML durch den Stereotyp *ContentClass* ausgezeichnet und besitzen den Tagged Value *stringName* vom Typ UML-Property. Diesem kann ein Attribut der Contentklasse zugewiesen werden, das zur Benennung der Contentklasse im laufenden System, z. B. in einer *toString*-Methode, verwendet wird. Diese wird in der Präsentation z. B. bei Drop-Down-Boxen für die Benennung der selektierbaren Elemente verwendet.

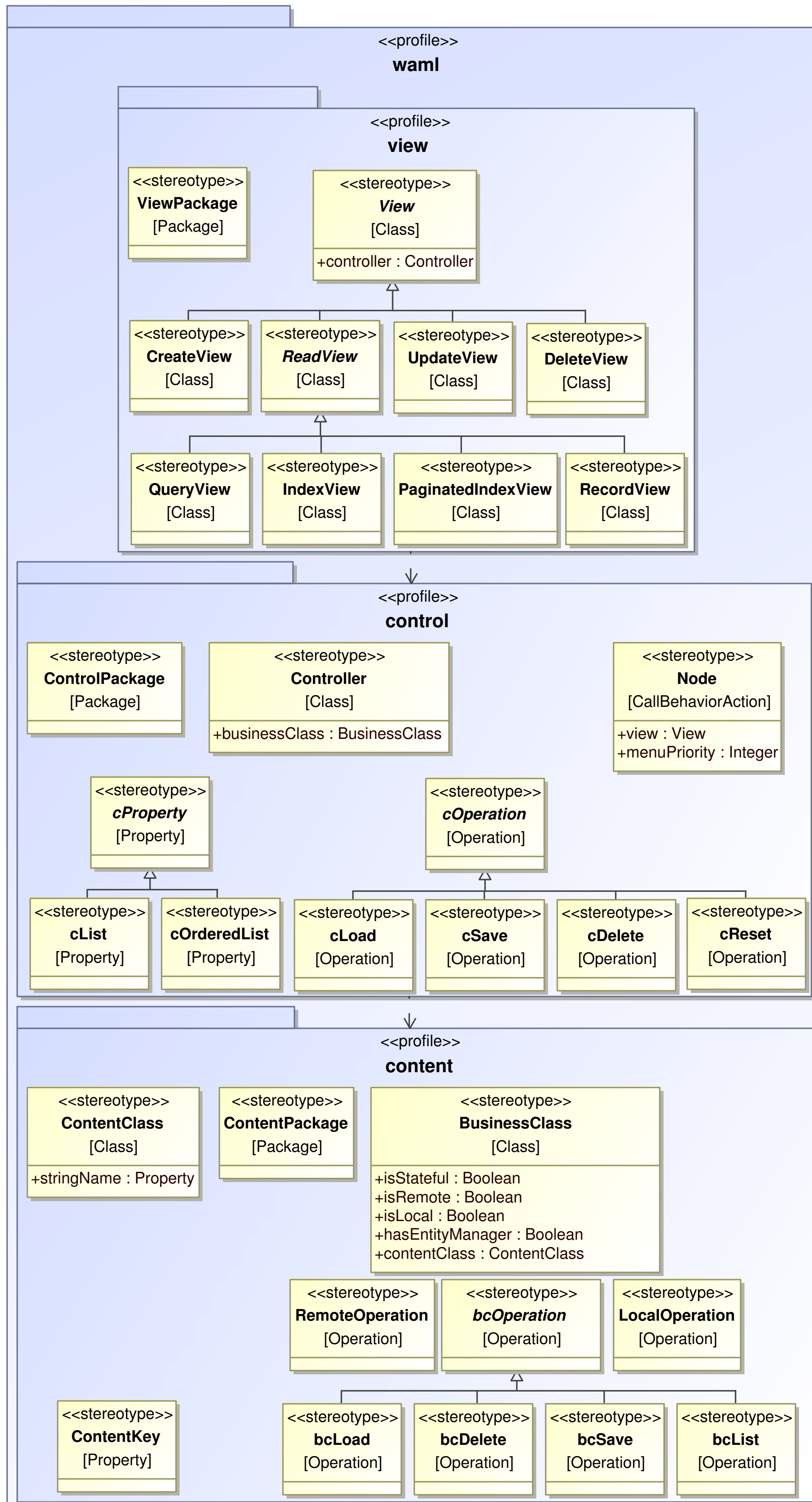


Abbildung 10: Metamodell der Web Application Modeling Language

Durch den Stereotyp *ContentKey* können Attribute von Contentklassen als IDs für die Entityklasse deklariert werden. Pro Contentklasse sollte nur genau ein Attribut dieser Contentklasse diesen Stereotyp besitzen und vom Typ *UML-Integer* sein. Damit ist sichergestellt, dass IDs generell numerisch sind und die Templates vereinfacht werden, da sie keine zusammengesetzten Schlüssel generieren müssen.

Geschäftsklassen werden durch den Stereotypen *BusinessClass* annotiert und enthalten Tagged Values, mit denen festgelegt werden kann, ob bei einer Transformation in z. B. Session Beans diese einen Zustand haben, ob sie nur innerhalb bzw. außerhalb des Servers ansprechbar sind und ob sie einen Entitymanager besitzen. Durch den Tagged Value *contentClass* wird einer Geschäftsklasse die Contentklasse zugeordnet, die durch die Geschäftsklasse primär verwaltet werden soll. Bei Operationen von Geschäftsklassen kann mit den Stereotypen *LocalOperation* und *Remote-Operation* festgelegt werden, ob sie innerhalb bzw. außerhalb eines Applikationsservers aufgerufen werden können. Falls eine Operation als RemoteOperation gekennzeichnet wird, muss sinnvollerweise auch deren Geschäftsklasse Remote-fähig sein. Standardoperationen für das Laden, Löschen, Speichern und Auflisten von Elementen der verwalteten Contentklasse können durch die Stereotypen *bcLoad*, *bcDelete*, *bcSave* und *bcList* ausgezeichnet werden. Auf Basis der modellierten Methodensignaturen und der Parameter der Geschäftsklasse können vollständig automatisiert auch die Methodenrümpfe der Operationen generiert werden. Die Operationen orientieren sich an den bei datenorientierten Systemen standardmäßig vorhandenen Operationen Create, Read, Update und Delete (CRUD). Die Funktionalitäten Create und Update fallen dabei in dem Stereotyp *bcSave* zusammen.

Content- und Geschäftsklassen werden in Contentpaketen zusammengefasst, die durch den Stereotyp *ContentPackage* ausgezeichnet werden. Durch die Deklaration von UML-Paketen als Contentpakete wird der Modellierungskontext der Content- und Geschäftsklassen expliziert. Diese Explikation des Kontexts wird auch bei der Modellierung mit den Sprachelementen aus den Profilen *control* und *view* vorgenommen. Dies hat den Vorteil, dass die Templates so definiert werden können, dass sie je nach Kontext nur die Elemente des jeweiligen, dem Kontext zugeordneten Profils transformieren. Falls also in einem Contentpaket Modellelemente mit Stereotypen des View-Pakets liegen, werden diese vom Template ignoriert. Die disjunkte Behandlung von Profilen durch die Templates zwingt den Modellierer bei der Modellierung zu der separation of concerns und somit zu einer klaren Struktur des Modells.

5.2 Profil Control

Im Profil *control* sind Sprachelemente für die Modellierung von Navigationsaspekten und Controllern enthalten. Mit dem Stereotyp *Controller* können Klassen modelliert werden, welche die operationale Funktionalität der GUI durch die Verarbeitung von MVC-Modelldaten vornehmen. Jedem Controller ist mit dem Tagged Value *businessClass* eine primäre Geschäftsklasse zugeordnet, auf die der Controller zugreift. Da Geschäftsklassen Contentklasse besitzen, sind auch Controllern indirekt Contentklassen zugeordnet. Ähnlich zu Operationen von Geschäftsklassen werden auch die Operationen von Controllern durch Stereotypen ausgezeichnet, um ihre Methodenrümpfe vollständig generieren zu können. Auch diese Stereotypen sind an der üblichen CRUD-Funktionalität orientiert. Mit dem Stereotyp *cReset* können Operationen ausgezeichnet werden, welche den Zustand der Contentklasse des Controllers auf einen Standardwert zurücksetzen. Navigationsstrukturen können mit dem Stereotyp *Node* modelliert werden, mit dem Aktionen in UML-Aktivitätsdiagrammen ausgezeichnet werden können. Knoten entsprechen in der Webapplikation Webseiten, die durch eine zugeordnete View um Darstellungsfunktionalität angereichert werden. Im Vergleich zum Controller für operative Funktionalität muss ein Controller zur Steuerung der Navigationsabläufe auf Basis der Knoten nicht explizit modelliert werden, da dieser Controllertyp bereits durch Frameworks wie JavaServer Faces (JSF) bereitsgestellt wird. Aus dem Aktivitätsdiagramm muss für diese Frameworks eine Datei generiert werden, welche den Controller des Frameworks hinsichtlich der Navigationsstrukturen der Webapplikation konfiguriert.

5.3 Profil View

Im Profil *view* werden Stereotypen für die Auszeichnung von Klassen mit standardisierter Darstellungsfunktionalität bereitgestellt. Durch den Stereotyp *View* bzw. dessen Spezialisierungen werden Präsentationselemente wie z. B. Links und Tabellen zu Funktionseinheiten aggregiert. Dies hat den Vorteil, dass die Modelle der View abstrakt und aussagekräftig sind.

Als Standardviews sind die *CreateView*, *ReadView*, *UpdateView* und *DeleteView* vorgesehen. Diese repräsentieren Darstellungsfunktionalität für die formularbasierte Eingabe einzelner Datensätze, für die Ausgabe von einem oder mehreren Datensätzen, für die formularbasierte Veränderung einzelner Datensätze und für die Abfrage vor der Löschung einzelner Datensätze. Die abstrakte View*ReadView* wird durch die Views *QueryView*, *IndexView*, *PaginatedIndexView* und *RecordView* spezialisiert. Darin kommt zum Ausdruck, dass alternative Darstellungsformen von Daten

ermöglicht werden sollen. Mit der *QueryView* wird eine formularbasierte Suche und Anzeige von Datensätzen aggregiert. Die *IndexView* und *PaginatedIndexView* sind beide tabellenorientierte Auflistungen von Datensätzen mit dem Unterschied, das die zweite View eine seitenweise Darstellung mit einer festen Anzahl von Datensätzen pro Seite ermöglicht. Mit der *RecordView* kann die Darstellung eines einzelnen Datensatzes modelliert werden. Die Hierarchisierung der Views sorgt für Modularität und ermöglicht die projektbezogene Erweiterung des Profils um weitere Views. Z. B. kann die Bibliothek für *ReadViews* um spezifischere Views erweitert werden, bei denen z. B. die Form der Darstellung parametrisiert werden kann.

Indem Views durch Navigationsknoten referenziert werden, können sie flexibel auf mehreren Seiten bzw. Knoten wiederverwendet werden. Außerdem kann die Entwicklung der Navigation zeitlich versetzt zur Entwicklung der Darstellungsfunktionalität erfolgen.

5.4 Meta-Assoziationen

Im Unterschied zu UWE werden Referenzen in WAML nicht durch Assoziationen im Metamodell ausgedrückt, sondern durch Tagged Values. Dies ist technisch darin begründet, dass Profile, die Assoziationen enthalten, bei der Verwendung von MagicDraw 14.0 nicht in XMI-Dateien exportiert werden können. Auf dieses Problem wird von MagicDraw beim Export zudem nur am Rande in den Log-Dateien hingewiesen, was zu Diffizilitäten im Entwicklungsprozess führen kann.

In UML werden Referenzierungen von Modellelementen primär durch Attribute, Assoziationen und Tagged Values ausgedrückt. Referenzen durch Tagged Values haben den Vorteil, dass Typsicherheit durch MagicDraw bei der Belegung mit Werten hergestellt werden kann. Zudem können sie in Templates unter ihrem Namen direkt referenziert werden und die Multiplizität bereits im Metamodell festgelegt werden. Im Vergleich zu stereotypisierten Assoziationen haben Tagged Values den Nachteil, dass sie nicht graphisch im Modell annotiert werden. Außerdem existiert bei Magic Draw 14.0 bei Tagged Values mit Referenzen anstatt von Primitiven das Problem, dass diese nicht in XMI 2.0-Dateien exportiert werden. Bei der Serialisierung in XMI 1.0-Dateien werden spezifikationgemäß nur String-basierte Namen der referenzierten Modellelemente exportiert. Dies ist aber in den Templates zu lösen und wird in Kapitel 6.4.5 erläutert. Bei WAML werden aufgrund der Vorteile für Modellelementreferenzen Tagged Values benutzt.

5.5 Selektion eines Navigationsdiagrammtyps

Wie bereits erläutert werden Navigationsmodelle in WAML in Form von Aktivitätsdiagrammen dargestellt. Für die Auswahl eines UML-Diagrammtyps zur Modellierung der Navigationsaspekte in WAML ist es von Relevanz, dass der Diagrammtyp ein Verhaltensmodell aus der UML ist, durch UML-Werkzeuge unterstützt wird und die Abbildung von Graphen ermöglicht. Im Fall von MagicDraw 14.0 werden die Verhaltensdiagrammtypen Sequenzdiagramm, Kommunikationsdiagramm, Anwendungsfalldiagramm, Zustandsautomatendiagramm und Aktivitätsdiagramm unterstützt und im folgenden auf ihre Eignung für die Navigationsmodellierung evaluiert. [43] [22]

Mit Sequenzdiagrammen kann der Austausch von Nachrichten zwischen Kommunikationspartnern modelliert werden. Der Fokus liegt dabei auf der Repräsentation eines einzelnen Kommunikationsablaufes auf Basis von Nachrichten. Graphisch wird der Ablauf auf Lebenslinien dargestellt, die ausgehend von den Objekten der Kommunikationspartnern als Anker für Nachrichten dienen. Insgesamt steht also nicht die Abbildung eines Netzes von Navigationspfaden im Vordergrund.

Ähnlich zum Sequenzdiagramm steht auch im Kommunikationsdiagrammtyp der Nachrichtenaustausch zwischen Kommunikationspartnern im Fokus, es greifen also die selben Kritikpunkte.

Anwendungsfalldiagramme zeigen eine Sicht auf das erwartete Verhalten eines Systems, indem Anwendungsfälle mit Akteuren in Verbindung gesetzt werden. Da sie keine Ablaufbeschreibung ermöglichen, unterstützen auch sie nicht angemessen den Netzwerkcharakter von Navigationsmodellen.

Diagramme für Zustandsautomaten stellen endliche Automaten als einen Ablauf von Zuständen dar, die ein Objekt im Lauf seines Lebens einnehmen kann. Die Zustände können durch Kanten miteinander in Verbindung gesetzt werden, sodass ein Graph erstellt werden kann. Semantisch sollen in WAML Webseiten durch Knoten repräsentiert werden, deren Darstellungsfunktionalität durch Views konkretisiert wird. Im Mittelpunkt bei der Modellierung von Webseiten steht also nicht der Aspekt des Zustandes der Webapplikation während der Darstellung einer bestimmten Seite, sondern der Aspekt der (Inter-)Aktionsmöglichkeiten auf den Seiten. Zwar können in Diagrammen für Zustandsautomaten UML-Aktivitäten verwendet werden, im Vordergrund stehen aber dennoch Zustände, die durch die Aktivitäten verändert werden. Somit wird die geforderte Semantik nicht optimal durch Zustandsdiagramme abgebildet, auch wenn die Form der Visualisierung durch einen Graphen Netzwerkcharakter besitzt und unabhängig von der Semantik für die Modellierung

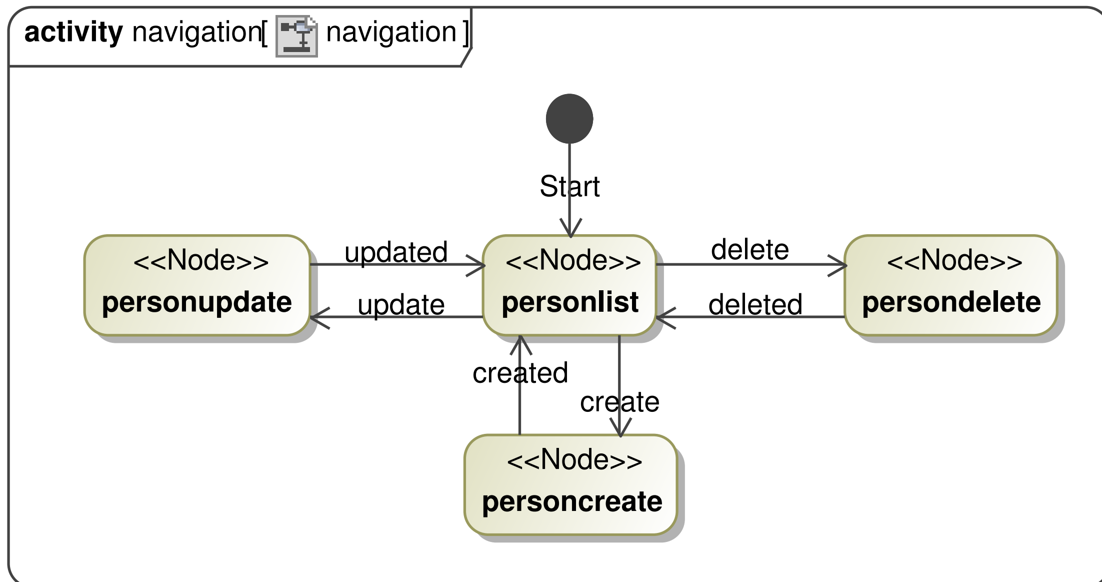


Abbildung 11: Navigationsmodell als Aktivitätsdiagramm

eines Navigationsmodells verwendet werden kann.

Mit Aktivitätsdiagrammen können Aktivitäten durch die Abfolge einzelner Aktionen modelliert werden. Aktionen repräsentieren Schritte in der Aktivität und werden durch Knoten visualisiert. Sie können durch Kanten in eine Ablaufreihenfolge gebracht werden und besitzen im Gegensatz zu Diagrammen für Zustandsautomaten keinen Zustandscharakter. Z. B. ist in Abbildung 11 ein Navigationsmodell auf Basis eines Aktivitätsmodells dargestellt, welches die Navigation einer einfachen Webapplikation zur Bearbeitung von Personendaten beschreibt. Die Knoten repräsentieren Webseiten bzw. deren Interaktionsaspekte und die Kanten definieren die Verlinkung der Seiten untereinander. Da Aktivitätsdiagramme die gewünschte Semantik grundlegend repräsentieren und sich durch Graphen visualisieren lassen, werden sie für Navigationsmodelle in WAML favorisiert.

6 Entwicklung einer prototypischen Webapplikation

Auf der Basis von WAML wird im Folgenden die modellgetriebene Entwicklung einer prototypischen Webapplikation im Sinne eines Proof of Concept vorgestellt. Dazu wird anhand der Beispieldomäne einer Bibliothek eine Webapplikation modelliert, die Zielplattform beschrieben und es werden die Templates für die Zielplattform erläutert.

6.1 Modelle

Der Webapplikation liegt entsprechend der Bibliotheksdomäne ein in Abbildung 12 dargestelltes Datenmodell zugrunde, das Klassen für Personen, Bücher und Ausleihen umfasst. Um verschiedene Kardinalitäten von Assoziationen darzustellen, existiert zwischen den Klassen für Bücher und Personen eine unidirektionale Assoziation mit der Kardinalität 1:*, dagegen zwischen den Klassen für Ausleihen und Bücher eine unidirektionale Assoziation mit der Kardinalität *:*.

Zu den Contentklassen existieren die korrespondierenden Geschäftsklassen, die stereotypisierte Methoden zum Speichern, Laden, Löschen und Auflisten ihrer jeweiligen Contentklassen enthalten. Für sämtliche Geschäftsklassen ist konfiguriert, dass sie Entitymanager referenzieren sollen, über die auf die Persistenzschicht der Webapplikation zugegriffen werden kann.

Im Paket *control* sind Klassen für die Datencontroller enthalten, welche ähnlich zu den Geschäftsklassen Methoden zum Laden, Speichern, Löschen, Auflisten und Zurücksetzen von Contentklassen enthalten. Die Methoden der Controller und Geschäftsklassen sind absichtlich inkonsistent benannt, um zu zeigen, dass die Operationen nicht durch Stringvergleiche der Namen identifizieren werden, sondern über die Stereotypen. So trägt z. B. die Methode zum Speichern von Personen im Controller den Namen *savePerson*, in der Geschäftsklasse dagegen den Namen *persistPerson*. Die Controller verweisen auf die korrespondierenden Geschäftsklassen und machen als Schicht zwischen der View und den als MVC-Model interpretierten Contentklassen die Daten und Funktionalitäten der Klassen des Contentpaketes für die Viewklasssen verfügbar.

Das Paket *view* umfasst Klassen für Darstellungsfunktionalitäten wie z. B. die Auflistung, Eingabe und Löschung von Personendatensätzen. Die Views sind mit den entsprechenden Controllern parametrisiert, die die operationale Funktionalität bereitstellen.

Das in Abbildung 13 dargestellte Navigationsmodell in Form eines Aktivitätsdiagramms enthält stereotypisierte Aktionen, die durch Kanten verbunden sind, welche

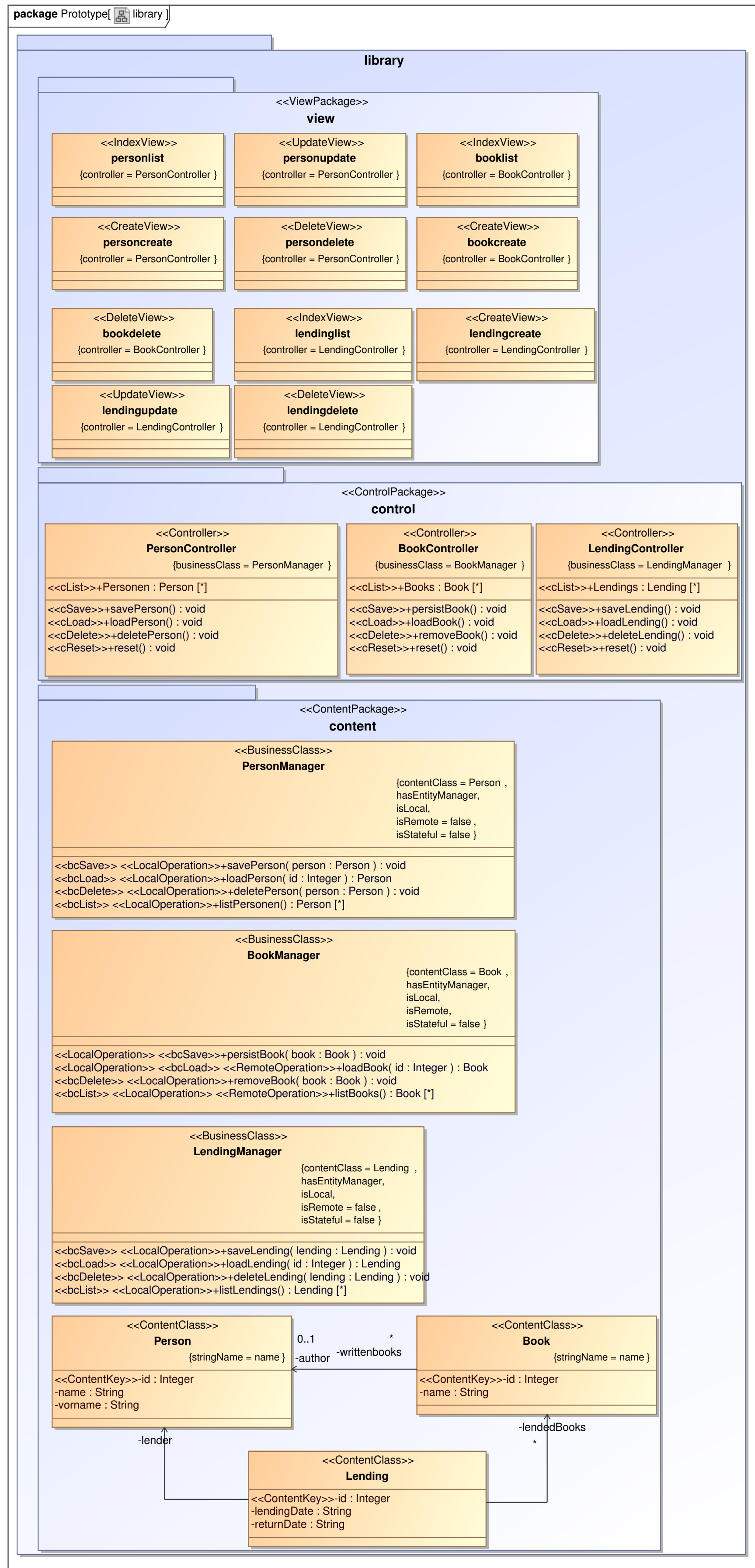


Abbildung 12: Klassendiagramm des Prototyps

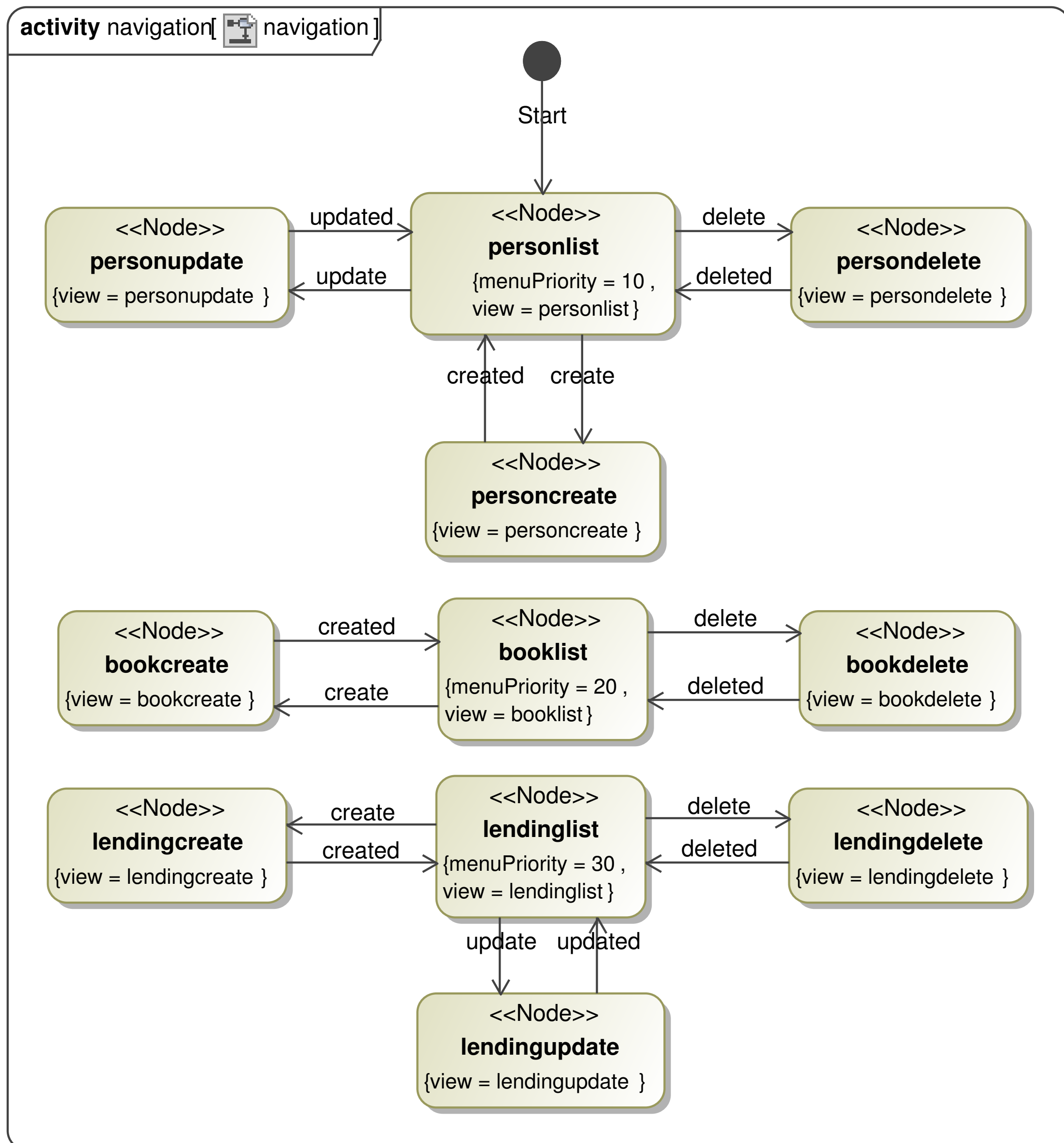


Abbildung 13: Navigationsdiagramm des Prototyps

die Navigationspfade repräsentieren. Die Knoten *personlist*, *booklist* und *lendinglist* werden durch die Angabe von Menüprioritäten für das Menü registriert. Prinzipiell müssten von jedem Knoten entsprechende Kanten zu diesen drei Knoten verlaufen, da über das Menü von jedem Knoten aus die drei Knoten erreicht werden können. Zugunsten der Klarheit des Modells wird auf die Modellierung von menübasierten Kanten verzichtet und mit den Menüprioritäten eine alternative Abbildungsform gewählt.

Den Knoten wird die Darstellungsfunktionalität zugewiesen, indem sie mit den entsprechenden Views parametrisiert sind. Z. B. ist modelliert, dass in der Personenliste bei den Personendatensätzen die Option zum Löschen von Datensätzen auf

einer weiteren Seite generiert werden soll, indem eine Kante zwischen dem Knoten der Personenliste und dem Knoten der Personenlöschung modelliert ist. Der Löschknoten ist somit im Kontext des Listenknotens vorhanden und parametrisiert auf diese Weise die View des Listenknotens.

6.2 Zielplattform

Als Zielplattform wird mit dem JBoss Application Server in der Version 4.2.2 eine Implementierung eines Java EE Application Servers benutzt.

Für die Persistierung der Contentklassen und die Abbildung der Geschäftslogik wird Enterprise JavaBeans in der Version 3.0 verwendet. In dieser Version existiert im Gegensatz zu älteren Versionen die Möglichkeit, Metainformationen zur Parametrisierung der Entity- und Geschäftsklassen direkt in den Klassen zu annotieren. Anhand der Annotationen wird z. B. der Entitymanager bzgl. der Abbildung der Objekte auf relationale Strukturen in der Datenbank konfiguriert. Die Verwendung von Annotationen reduziert den Aufwand bei der Definition der Templates, da sowohl die Modelle als auch die Annotationen im Quelltext deskriptiven Charakter besitzen und somit direkt aufeinander abgebildet werden können.

Als Framework für die Realisierung der MVC-Architektur wird JavaServer Faces (JSF) benutzt. JSF basiert auf JavaServer Pages bzw. Servlets und enthält abstrakte Komponenten für die Erstellung von Benutzerschnittstellen sowie für die Definition von Navigationsstrukturen. Die Navigationsstrukturen werden durch eine Descriptoren-Datei beschrieben. Für die Unterstützung von Ajax-orientierten Views wird das JSF-kompatible Ajax-Framework ICEFaces verwendet, das für die Komponentenbibliothek von JSF entsprechende Ajax-basierte Implementierungen bietet. Ein wichtiges Details ist zudem, dass Seiten bei ICEFaces XML-konform sind. Bei JSF wird gefordert, dass die Komponentenbibliotheken von JSF in jeder JSF-Seite konform zu JSP durch Deklarativen der Form <%@ [...] %> eingebunden werden. Bei ICEFaces geschieht dies XML-konform durch die Definition von XML-Namensräumen. Dies ermöglicht die Verwendung des XML-Beautifiers von oAW, der als Eingabe wohlgeformte XML-Dateien fordert. Nachteilig an ICEFaces ist die steigende Abhängigkeit von Frameworks. Da die Komponenten von ICEFaces aber den Komponenten von JSF ähneln, kann eine Modifikation der Templates für einen Wechsel von ICEFaces auf JSF ohne strukturelle Veränderungen an den Templates vorgenommen werden.

6.3 Projektressourcen

6.3.1 Struktur

Die Ressourcen zur Generierung der Webapplikation umfassen das Metamodell, das Modell der Webapplikation, Xpand-Templates, Xtend-Bibliotheken und einen Workflow, der den Generator steuert. Die Ressourcen sind in einem Eclipseprojekt zusammengefasst, dessen Klassenpfad so konfiguriert ist, dass sämtliche genannten Projektressourcen darin liegen. Zudem müssen die oAW-Klassen im Klassenpfad des Projekts enthalten sein, die vereinfachend in Form eines Eclipse-Plugin-Projektes zusammengefasst sind und als solches durch eine Eclipse-Plugin-Dependency zum Klassenpfad hinzugefügt sind.

Der Zielordner der zu generierenden Artefakte muss sich innerhalb des Klassenpfades eines Eclipse-Projektes befinden, um in Eclipse Syntaxhervorhebungen für Java und Meldungen über Fehler im Java-Quelltext nutzen zu können. Generell ist es sinnvoll, die Generate in einem separaten Projektordner zu speichern, der einen eigenen Klassenpfad besitzt.

Durch die Aufteilung auf mehrere Projekte können die Ressourcen anhand ihrer Abstraktionsebenen getrennt werden. oAW-Ressourcen liegen im Sinne von Metadaten auf einer höheren Abstraktionsebene als die generierten Daten. Durch die Trennung werden die Fehlermeldungen zum Quelltext von Eclipse projektbezogen ausgegeben. Falls ein Team mit der Entwicklung der oAW-Ressourcen beschäftigt ist und ein weiteres Team mit den generierten Quelltexten arbeitet, werden zudem Namenskonflikte zwischen oAW-Dateien und Javaklassen vermieden. Auch können Projekterweiterungen von Eclipse genutzt werden, die z. B. das Deployment auf einen J2EE-Server unterstützen oder im Fall des oAW-Projektetyps Eclipse um Syntaxhervorhebungen von oAW-spezifischen Sprachen wie Xpand erweitern.

Für die Generierung des Prototyps wird aus dem Generatorprojekt *PrototypeGenerator* Quelltext erzeugt und in die Ordner der Zielprojekte *PrototypeBackend* und *PrototypeWeb* gespeichert. Das Projekt *PrototypeWeb* enthält JavaServer Pages, Controllerklassen, statische Inhalte und Deployment Descriptoren. Das Projekt *PrototypeBackend* umfasst Klassen für Entity Beans und Session Beans, Deployment Descriptoren und ein Ant-Script, welches aus den Ressourcen der beiden Projekte ein Enterprise Application Archive (EAR) generiert und dieses auf den JBoss Appication Server ausliefert.

Die Templates und Funktionsbibliotheken sind im Generatorprojekt ähnlich zu den Profilen im Metamodell von WAML in entsprechenden Ordnern zusammengefasst.

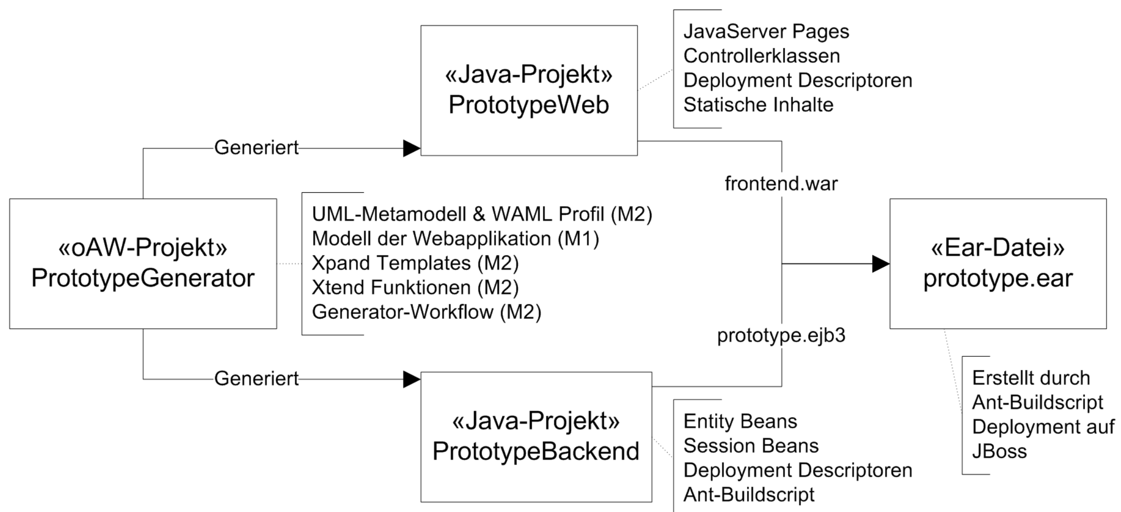


Abbildung 14: Projektressourcen

- Im Hauptordner *templates/* befinden sich Templates und Bibliotheken für die Erstellung von generische Javaklassen aus UML-Klassen, also ohne Bezug auf das WAML-Metamodell.
- Im Ordner *templates/waml/* liegen verteilt auf die Unterordner *content/*, *control/* und *view/* Dateien für die Generierung WAML-spezifischer Modellelemente. Die Namen der Templateordner entsprechen den Namen der Subprofile im Metamodell von WAML. Die Abhängigkeiten der Templates und Funktionen sind unidirektional gestaltet, sodass das Profil View auf den Profilen Content und Control aufbaut, und das Profil Control mit Ausnahme des Sprachelements *Node* ausschließlich das Profil Content voraussetzt. Die Beschreibung der Projektressourcen in diesem Kapitel orientiert sich bottom-up an dieser Abhängigkeitsstruktur.

Jeder Templateordner enthält eine Datei *Root.xpt*, in der Templates mit dem Namen *Root* für die im Templateordner behandelten Metamodellelemente enthalten sind. Die Transformation wird durchgeführt indem der oAW-Workflow generator.oaw das Metamodell und das Modell lädt und mit dem geladenen Modell das primäre Root-Template in der Datei *template/Root.xpt* aufruft. Dieses unterscheidet nach den Typen der Modellelemente und initiiert mittels Polymorphie entweder die Ausführung von Templates im Ordner *template/*, oder startet spezifischere Root-Templates für die WAML-Modellelemente.

Im Folgenden werden mit dem Projekt PrototypeGenerator die relevanten oAW-Ressourcen beschrieben. Zur besseren Übersicht und Platzersparnis sind irrelevante Codefragmente ausgelassen, sodass nicht sämtliche dargestellten Templates lauffähig

sind. Die lauffähigen Versionen befinden sich in der Anlage.

6.3.2 Workflow

Der Generatorprozess von oAW wird durch eine Workflowdatei im XML-Format beschrieben, die auch kurz Workflow genannt wird. Der Workflow kann entweder durch Eclipse oder durch Ant-Scripte initialisiert werden. [42, S. 193] Zweiters ist relevant, um Workflows in auf Ant basierende Buildprozesse zu integrieren, die bei größeren Projekten Anwendung finden, um Inkonsistenzen zu vermeiden [42, S. 136 f.] und die Entwicklungsgeschwindigkeit zu erhöhen. Innerhalb von Workflows wird der Ablauf und die Konfiguration der Generierung beschrieben, indem Komponenten von oAW sequentiell aufgerufen werden.

```
1  <?xml version="1.0"?>
2  <workflow>
3    <bean class="oaw.uml2.Setup" standardUML2Setup="true"/>
4    <component class="oaw.emf.XmiReader">
5      <modelFile value="model/Prototype.uml2"/>
6      <outputSlot value="modelslot"/>
7    </component>
8    <component id="generator" class="oaw.xpand2.Generator" skipOnErrors="false">
9      <metaModel id="MM_EMF" class="oaw.type.emf.EmfMetaModel">
10       <metaModelPackage value="org.eclipse.emf.ecore.EcorePackage"/>
11     </metaModel>
12     <metaModel id="MM_UML2" class="oaw.uml2.UML2MetaModel"/>
13     <metaModel id="MM_Profile_Waml" class="oaw.uml2.profile.ProfileMetaModel">
14       <profile value="model/waml.profile.uml2"/>
15     </metaModel>
16     ...
17     <expand value="templates::Root::Root␣FOR␣modelslot"/>
18     <outlet path="src">
19       <postprocessor class="oaw.xpand2.output.JavaBeautifier"/>
20     </outlet>
21     <outlet name="backendClasses" path="../PrototypeBackend/src" append="false"
               overwrite="true">
22       <postprocessor class="oaw.xpand2.output.JavaBeautifier"/>
23     </outlet>
24     ...
25     <prSrcPaths value="../PrototypeBackend/src,../PrototypeWeb/src,../PrototypeWeb/
               classes,../PrototypeWeb/WEB-INF"/>
26     ...
27   </component>
28 </workflow>
```

Listing 9: Workflow

Im Workflow in Listing 9 werden die Schritte des Einlesens des Modells und der Transformation beschrieben. Das zu transformierende und in XMI serialisierte Modell soll durch die Komponente XmiReader von oAW eingelesen und in eine Slot

genannte Variable *modelslot* geschrieben werden. Die Generatorkomponente wird mit Angaben zu Metamodellen, Templates und Ausgabeordnern, Postprozessoren, geschützten Regionen und Zeichenkodierungen konfiguriert. Konkret wird angegeben, dass das EMF und das UML2-Metamodell bzw. dessen Profile benutzt werden sollen. Zur Transformation des im Slot *modelslot* liegenden Modells wird das primäre Xpand-basierten Roottemplate vorgegeben. Die in den Xpand-Templates enthaltenen FILE-Befehle verweisen auf Outlets, die so konfiguriert sind, dass die Ausgabedateien in den beiden Zielprojekten gespeichert werden. Outlets für Javaklassen werden für Postprozessoren markiert, die den Javaquelltext für eine bessere Lesbarkeit optimieren. Ordner mit Dateien, die geschützte Regionen enthalten, müssen als solche kommasepariert vermerkt werden.

6.4 Basistemplates und -funktionen

6.4.1 Primäres Roottemplate

Das primäre Root-Template in Listing 10 wird mit einem UML-Modell initialisiert und expandiert die folgenden sekundären Root-Templates für die Elemente des Modells. Die Metaoperation *this.ownedElement()* gibt eine Liste der im Modell enthaltenen Elemente auf der höchsten Hierarchiestufe zurück, also z. B. Klassen und Pakete, aber nicht die in den Paketen enthaltenen Elemente. Alternativ dazu gibt die Metaoperation *this.allOwnedElements()* eine Liste aller Elemente des Modells zurück, also z. B. auch Klassen, die in Paketen enthalten sind. Mit der Beendigung der FOREACH-Schleife im primären Root-Template wird die Verarbeitung der Templates beendet und der aufrufende Workflow fortgesetzt.

Anhand der Polymorphie wird von Xpand automatisch das passende Template selektiert und expandiert. Dabei wird das Model nach Paketen bzw. Pakethierarchien analog zur Tiefensuche rekursiv nach Aktivitäten, ContentPackages und ControlPackages durchsucht. Falls ein Element vom Typ Aktivität gefunden wird, wird das entsprechende Root-Template für Waml-Navigationsmodelle geladen. Bei ContentPackages bzw. Controlpackages wird das entsprechende Root-Template geladen. Sämtliche weiteren Modellelemente mit anderen Typen als den im Template genannten werden durch das letzte Root-Template für UML-Elemente abgefangen und nicht weiter bearbeitet. Dieses generische Template wird von Xpand gefordert, da sonst die Verarbeitung sämtlicher Typen durch inklusionsbasierte Polymorphie nicht sichergestellt werden kann.

```
«DEFINE Root FOR uml::Model»
  «EXPAND Root FOREACH this.ownedElement»
«ENDDEFINE»

«DEFINE Root FOR uml::Package»
  «EXPAND Root FOREACH this.ownedElement»
«ENDDEFINE»

«DEFINE Root FOR content::ContentPackage»
  «EXPAND templates::waml::content::Root::Root»
«ENDDEFINE»

«DEFINE Root FOR control::ControlPackage»
  «EXPAND templates::waml::control::Root::Root»
«ENDDEFINE»

«DEFINE Root FOR uml::Activity»
  «EXPAND templates::waml::control::Root::Root»
«ENDDEFINE»

«DEFINE Root FOR uml::Element»«ENDDEFINE»
```

Listing 10: primäres und sekundäre Root-Templates

6.4.2 Templates für Klassen

Als Basis für Klassenkostrukte ist in Listing 11 ein Template definiert, mit dem einfache Javaklassen bzw. Klassen für Plain Old Java Objects (POJO) aus UML-Klassen erstellt werden können. POJO-Klassen sind Javaklassen, die nur an Restriktionen der Sprache Java gebunden sind. Sie implementieren weder Interfaces, noch spezialisieren sie Oberklassen oder enthalten sie Annotationen. Das Template ist für den Typ UML-Klasse definiert und generiert eine Klassendatei mit dem Klassennamen. Aufgaben der Namensfindung werden an Xtend-Funktionen delegiert. Nach dem Schreiben von Attributen und Operationen in die Klassendatei durch eigens dafür definierte Templates wird grundsätzlich für jedes Attribut eine Get- und Set-Methode generiert. Die Zugriffsmethoden müssen nicht als solche in der UML-Klasse expliziert werden, sondern werden aus den Attributen abgeleitet. Daraus folgt, dass Attribute generell über die Zugriffsmethoden modifiziert werden können. Aus Gründen des Umfangs der Listings wird auf die Generierung abstrakter und finaler Klassen verzichtet.

```
«DEFINE Root FOR uml::Class»
  «FILE getClassPath(this)»
    package «getPackageName(this.package)»;
    public class «this.name»{
    «EXPAND Property FOREACH this.ownedAttribute»
    «EXPAND Operation FOREACH this.ownedOperation»
    «EXPAND Getter FOREACH this.ownedAttribute»
```

```
8     «EXPAND Setter FOREACH this.ownedAttribute»
9     }
10  «ENDFILE»
11 «ENDDEFINE»
```

Listing 11: Klassentemplate

Die Templates für die Generierung von Attributen in Listing 12 und deren Zugriffsmethoden in Listing 13 nehmen Modellelemente vom Typ uml::Property entgegen. Falls ein UML-Attribut die Eigenschaft derived besitzt, also ein von anderen Attributen oder Operationen abgeleitetes Attribut ist, wird dieses nur in Form eines Kommentars im Attributsblock vermerkt, da das Attribut durch die Get-Methode konkretisiert wird. Falls das Attribut nicht abgeleitet ist, wird es unter Beachtung der Multiplizität, Sichtbarkeit und des Typen in die Klasse geschrieben. In der Auswahl des Attributstyps durch den Funktionsaufruf *getTypeName(this)* wird überprüft, ob die UML-Multiplizität des Attributs einfach oder mehrfach ist. Bei mehrfacher Multiplizität wird das Attribut in eine generische Arrayliste parametrisiert mit dem Typ des Attributs transformiert.

Entsprechend wird auch bei der Deklaration der Get- und Set-Methoden die Multiplizität beachtet. Falls das Attribut abgeleitet ist, wird in der Get-Methode eine geschützt Region eingefügt, in der manuell die Berechnung des Attributwerts definiert werden kann. Andernfalls wird die Klassen- bzw. Membervariable zurückgegeben. Dabei wird dem Namen des Attributs kein *this* vorangestellt, weil in Folge dessen eine Fallunterscheidung zwischen Member- und Klassenvariablen eingeführt werden müsste, um alternativ bei Klassenvariablen den Bezug auf die Klasse zu setzen. Für die Identifikation der geschützten Region wird eine ID der Form PAKETNAME_SUBPAKETNAME_ KLASSENNAME_ATTRIBUTNAME erstellt, d. h. bei einer Veränderung der Paketstruktur bleibt der Inhalt der geschützten Region nicht erhalten, weil sich die ID ändert. Dieses paketbasierte ID-Muster findet sich im Folgenden auch bei geschützten Regionen in Methoden etc. wieder.

Im Rumpf der Set-Methode wird bei nicht-abgeleiteten Attributen der Parameterwert in die Klassen- bzw. Membervariable geschrieben. Bei abgeleiteten Attributen wird generell eine Exception geworfen, da der Wert abgeleiteter Attribute nicht sinnvoll auf direktem Wege modifiziert werden kann. Alternativ könnte generell auf die Generierung von Set-Methoden für abgeleitete Attribute verzichtet werden. Wie bei dem Template für Get-Methoden wird auf die explizite Referenzierung der Klasse durch *this* verzichtet, da dies zu ein Fallunterscheidung für Klassenvariablen führt. Zur Vermeidung von Namenskonflikten zwischen Parametern und Attributen infolge der indirekten Referenzierung wird der Parameter mit einem Präfix versehen.

```
«DEFINE Property FOR uml::Property»
  «IF isDerived»
    // derived property «this.name»
  «ELSE»
    «IF !isMultiple(this)»
      «getVisibility(this)» «getStatic(this)» «getTypeName(this)» «getAttributeName(
          this)»;
    «ELSE»
      «getVisibility(this)» «getStatic(this)» «getTypeName(this)» «getAttributeName(
          this)» = new java.util.ArrayList<«getQualifiedTypeName(this.type)»>()
          ;
    «ENDIF»
  «ENDIF»
«ENDDEFINE»
```

Listing 12: Template für Attribute in Klassen

```
«DEFINE Getter FOR uml::Property»
  public «getStatic(this)» «getTypeName(this)» «getGetterName(this)»(){
  «IF this.isDerived»
    «PROTECT CSTART "/* " CEND " */" ID getGetterId(this)»
      // TODO: calculate return-value
      return null;
    «ENDPROTECT»
  «ELSE»
    return «getAttributeName()»;
  «ENDIF»
  }
«ENDDEFINE»

«DEFINE Setter FOR uml::Property»
  public «getStatic(this)» void «getSetterName(this)»(«getTypeName(this)»
        p_«getParameterName(this)»){
  «IF isDerived»
    throw new UnsupportedOperationException("setter of a derived property should not
        be called.");
  «ELSE»
    «getAttributeName()» = p_«getParameterName()»;
  «ENDIF»
  }
«ENDDEFINE»
```

Listing 13: Templates für Zugriffsmethoden auf Attribute in Klassen

Im Template in Listing 14 wird bei der Generierung von Methodendeklarationen bzw. Operationsprototypen überprüft, inwieweit diese sichtbar sind, ob sie statisch sind und welchen Rückgabetyp sie besitzen bzw. ob dieser *void* ist. Die Parameter werden getrennt durch Kommata in einer Schleife expandiert und können eine einfache oder mehrfache Multiplizität haben. Falls die Operation keinen Rückgabeparameter hat, wird eine geschützte Region generiert. Andernfalls wird eine Rückgabevariable *result* vom Typ des Rückgabetyps generiert, für die in einer geschützten

Region manuell definiert werden kann, wie ihr Wert berechnet werden soll.

```
1  «DEFINE Operation FOR uml::Operation»
2    «getVisibility(this)» «getStatic(this)» «getTypeName(this)» «this.name»(«EXPAND
             Parameter FOREACH getParameterList(this) SEPARATOR ","»){
3    «IF !isVoid(this)»
4      «getTypeName(this)» result;
5    «ENDIF»
6    «PROTECT CSTART "/* " CEND " */" ID getOperationId(this)»
7      «IF !isVoid(this)»
8        // TODO: write method body
9        result = null; // «type.name»
10     «ENDIF»
11   «ENDPROTECT»
12   «IF !isVoid()»
13     return result;
14   «ENDIF»
15   }
16 «ENDDEFINE»
17
18 «DEFINE Parameter FOR uml::Parameter»
19   «getTypeName(this)» «getParameterName(this)»
20 «ENDDEFINE»
```

Listing 14: Template für Methoden in Klassen

6.4.3 Funktionen für Klassen

Die in den Templates verwendeten Funktionen sind gesondert in Xtend-Dateien zusammengefasst, die entsprechend den Aspekten einer Klasse strukturiert sind.

Die Bibliothek für Paketfunktionen in Listing 15 enthält Funktionen zur Bestimmung von Paketnamen für verschiedene Anwendungskontexte. Die Namen werden mit Hilfe der Funktion *getPackageConcat(uml::Package p)* rekursiv ermittelt, indem bei Paketen ermittelt wird, ob sie sich in einer Pakethierarchie befinden und die Namen der Super- und Subpakete durch Rauten getrennt konkateniert werden. Die Rauten werden je nach Anwendungskontext für Javaquelltexte durch Punkte, für Dateisystemreferenzen durch Slashes und für IDs durch Unterstriche ersetzt.

```
1  String getPackageName(uml::Package p) :
2    getPackageConcat(p).replaceAll("#", ".");
3  String getPackagePath(uml::Package p) :
4    getPackageConcat(p).replaceAll("#", "/");
5  String getPackageId(uml::Package p) :
6    getPackageConcat(p).toUpperCase().replaceAll("#", "_");
7  String getPackageConcat(uml::Package p) :
8    hasSuperPackage(p.nestingPackage) ? getPackageConcat(p.nestingPackage)+"#"+p.name :
             p.name;
9  Boolean hasSuperPackage(uml::Package p) :
10   p.nestingPackage != null;
```

Listing 15: Paketfunktionen

Die Bibliothek für Funktionen zur Verarbeitung von UML-Klassen in Listing 16 umfasst Funktionen zur Bestimmung von Klassennamen, Klassendateipfaden und IDs von Klassen. Diese nutzen Funktionen aus der Paketbibliothek und erstellen die Bezeichner analog zu den Funktionen in der Paketbibliothek durch Konkatenation.

```
String getClassName(uml::Class c) :
  c.name;
String getPackagedClassName(uml::Class c) :
  getPackageName(c.package) + "." + getClassName(c);
String getClassPath(uml::Class c) :
  getPackagePath(c.package) + "/" + getClassName(c) + ".java";
String getClassId(uml::Class c) :
  getPackageId(c.package) +  "_" + getClassName(c).toUpperCase();
```

Listing 16: Klassenfunktionen

In der Bibliothek für Attributsfunktionen in Listing 17 stehen Funktionen zur Bestimmung von Attributsnamen, IDs, Sichtbarkeiten und Art der Referenzierung bereit. Die IDs für Get- und Set-Methoden sind Erweiterungen der IDs ihrer Attribute. Analog werden die Namen der Zugriffsmethoden durch Konkatenation der Strings *get* und *set* mit dem Namen des Attributs gebildet.

```
String getAttributeName(uml::Property p) :
  p.name.toFirstLower();
String getAttributeId(uml::Property p) :
  getClassId(p.class) + "_" + p.name.toUpperCase();
String getVisibility(uml::Property p) :
  p.visibility.toString();
String getStatic(uml::Property p) :
  p.isStatic ? "static" : "";
String getGetterName(uml::Property p) :
  "get" + p.name.toFirstUpper();
String getSetterName(uml::Property p) :
  "set" + p.name.toFirstUpper();
String getGetterId(uml::Property p) :
  getAttributeId(p) + "_GETTER";
String getSetterId(uml::Property p) :
  getAttributeId(p) + "_SETTER";
```

Listing 17: Attributsfunktionen

Die Bibliothek für Funktionen auf Operationen in Listing 18 umfasst sowohl Funktionen für die Bestimmung von OperationIds, Sichtbarkeiten und Referenzierungsarten als auch Funktionen für Operationsparameter. Bei der Bestimmung der Sichtbarkeit wird diese nur als String zurückgegeben, falls die Sichtbarkeit nicht vom Wert *package* ist, da der Wert *package* der Standardwert in Java ist. Die Parameterliste wird ermittelt, indem diejenigen Operationsparameter ausgewählt werden, die keine Rückgabeparameter im Sinne der UML-Variable ParameterDirectionKind sind.

```
String getOperationId(uml::Operation op) :
   getClassId(op.class) +  "_" + op.name.toUpperCase();
String getVisibility(uml::Operation o) :
  o.visibility.toString() != "package" ? o.visibility.toString() : "";
String getStatic(uml::Operation o) :
  o.isStatic ? "static" : "";
List[Parameter] getParameterList(uml::Operation op) :
  op.ownedParameter.select(e|!isReturn(e));
String getParameterName(uml::Property p) :
  p.name;
String getParameterName(uml::Parameter p) :
  p.name;
Boolean isVoid(uml::Operation op) :
  op.type.name == "void";
Boolean isReturn(uml::Parameter p) :
  p.direction == uml::ParameterDirectionKind::return;
```

Listing 18: Funktionen auf Operationen

6.4.4 Funktionen für Typen

Die Funktionen zur Bestimmung von Typnamen und Typabbildungen sind in der Typ-Funktionsbibliothek in Listing 19 zusammengefasst. Bei der Bestimmung von Typnamen für Eigenschaften, Operationen und Parameter wird unterschieden, ob die Multiplizität des Typs einfach oder mehrfach ist. Eine mehrfache Multiplizität wird im Typnamen durch eine generische Collection repräsentiert.

```
String getTypeName(uml::Property p) :
  isMultiple(p) ? "java.util.Collection<" + getQualifiedTypeName(p.type) + ">" :
            getQualifiedTypeName(p.type);
String getTypeName(uml::Operation op) :
  isMultiple(op) ? "java.util.Collection<" + getQualifiedTypeName(op.type) + ">" :
            getQualifiedTypeName(op.type);
String getTypeName(uml::Parameter p) :
  isMultiple(p) ? "java.util.Collection<" + getQualifiedTypeName(p.type) + ">" :
            getQualifiedTypeName(p.type);
```

Listing 19: Funktionen auf Typnamen

Die Multiplizität ist im UML-Metamodellelement *MultiplicityElement* in den Eigenschaften lower und upper definiert.[8, S. 95] Mit dem Feld lower vom Typ Integer wird die untere Grenze der Multiplizität angegeben, mit dem Feld upper vom Typ UnlimitedNatural entsprechend die obere Grenze. Der Eigenschaftswert * für eine unrestringierte obere Grenze wird in der Serialisierung durch XMI durch die Zahl -1 repräsentiert, auch wenn dies im Sinne des Typen UnlimitedNatural nicht erlaubt ist. Falls im Modell z. B. einem Attribut eine Multiplizität 1..* zugeordnet ist, wird diese in der XMI-Serialisierung des Modells durch die Werte lower=1 und upper=-1

dargestellt. Die Multiplizität wird funktional bestimmt, indem die UML-Eigenschaft upper auf den Wert -1 überprüft wird, die untere Grenze ist also irrelevant.

Für Operationen muss die Überprüfungsfunktion der Multiplizität explizit definiert werden, da Operationen weder direkt noch indirekt Spezialisierungen vom MultiplicityElement sind. Generell kann der Return-Parameter einer Operation ermittelt werden und mit diesem die Funktion isMultiple() für MultiplicityElement ausgeführt werden. Das UML-Metamodell bietet aber eine Vereinfachung an, indem die Multiplizität aus den Parametern einer Operation im Sinne des OCL-Statement

```
1 upper = if returnResult()->notEmpty() then returnResult()->any().upper else Set{}
          endif
```

abgeleitet wird. [8, S. 105]

```
1 Boolean isMultiple(uml::MultiplicityElement p) :
2   p.upper == -1;
3 Boolean isMultiple(uml::Operation o) :
4   o.upper == -1;
```

Listing 20: Funktionen zur Bestimmung der Multiplizität

Typabbildungen werden mittels Listing 21 vorgenommen, um von den Instanzen des Metamodellelements *PrimitiveTypes* die entsprechenden Java-Basisdatentypen abzuleiten. Die im UML-Metamodell enthaltenen UML-Primitive sind *Boolean*, *Integer*, *String* und *UnlimitedNatural*. [8, S. 610 – 615] Mit der Funktion *isPrimitive* kann für einen UML-Typ bestimmt werden, ob er ein primitiver Typ ist. Dies ist z. B. für die Art der Präsentation auf einer Webseite relevant. Falls weitere Primitive aus Metamodellerweiterungen für UML, wie z. B. die bei MagicDraw enthaltene Metamodellerweiterung, verwendet werden, müssen die Funktionen für Typabbildungen entsprechend adaptiert werden.

Bei der Bestimmung von Javatypnamen zu UML-Typnamen wird der vollqualifizierte Javatypname zurückgegeben, um bei Klassentemplates keine Javapakete importieren zu müssen sowie Typnamenskonflikte zu vermeiden und somit die Templates zu vereinfachen. Da Java über keinen vorzeichenlosen Ganzzahlentyp verfügt [47], wird der vorzeichenlose UML-Datentyp *UnlimitedNatural* durch den Javatypen *Integer* repräsentiert. Falls der abzubildende Datentyp nicht bekannt ist, wird sein vollqualifizierter Javaname durch Konkatenation generiert.

```
1 Boolean isPrimitive(uml::Type type) :
2   switch(type.name){
3     case "Boolean" : true
4     case "Integer" : true
5     case "String" : true
6     case "UnlimitedNatural" : true
7     default : false
```

```
8   };
9 String getQualifiedTypeName(uml::Type type) :
10  switch (type.name) {
11    case "void" : "void"
12    case "Boolean" : "java.lang.Boolean"
13    case "Integer" : "java.lang.Integer"
14    case "String" : "java.lang.String"
15    case "UnlimitedNatural" : "java.lang.Integer"
16    default: getPackageName(type.package) + "." + type.name
17  };
```

Listing 21: Funktionen zur Typabbildung

6.4.5 Funktionen für Tagged Values

Mit Tagged Values können die Metamodellelemente von UML um Eigenschaften erweitert werden, die anhand von Bezeichnern bzw. Tags in oAW als Bestandteil des Metamodells referenziert werden können. In UML1 sind Tagged Values generell vom Typ *String* [8, S. 647]. Tagged Values in UML2 können als Metaattribute beliebige Typen haben und müssen entsprechend in XMI als Basisdatenwerte oder Modellelementreferenzen serialisiert werden. Bei Primitiven geschieht die Serialisierung in XMI, indem dem attributierten Modellelement ein Tag mit dem Namen und Wert des Tagged Value zugewiesen wird. Beispielsweise wird eine Geschäftsklasse mit dem Tagged Value hasEntityManager=true folgendermaßen serialisiert:

```
1 <content:BusinessClass xmi:id="[...]" [...] hasEntityManager="true"/>
```

Bei Tagged Values mit Referenzen auf Modellelemente muss entsprechend anstelle des Basisdatenwerts eine Referenz-ID angegeben werden. Wie beschrieben bereitet dies mit dem zur Modellierung des Protoypen eingesetzten Modellierungswerkzeug MagicDraw UML 14.0 Probleme. Tagged Values mit Referenzen werden in XMI2 nicht serialisiert und somit können diese Tagged Values in Templates zwar referenziert werden, enthalten aber während der Generierung nie Werte. Bei der Serialisierung in XMI1 wird entsprechend der Konvention von UML1 die Referenz in Form des Namens des referenzierten Modellelements dargestellt. Im Beispiel ist dies der Name *Prototype::library::functionality::bookindex*.

```
1 <node xmi:type="uml:CallBehaviorAction" xmi:id="[...]" name="booklist" [...] >
2   <eAnnotations xmi:id="[...]" source="appliedStereotypes">
3     <contents xmi:type="navigation_0:navigation__Node" xmi:id="[...]" functionality="
              Prototype::library::functionality::bookindex"/>
4   </eAnnotations>
5 </node>
```

Um Tagged Values mit Referenzen nutzen zu können wird deshalb zur Serialisierung XMI1 gewählt. Zur Ermittlung von Referenzen aus den Namen von Modellelementen

stehen die Funktionen aus Listing 21 bereit.

```
1  uml::Class getClass(String name, uml::Model m) :
2    m.allOwnedElements().typeSelect(uml::Class).select(e|e.qualifiedName == name).first
           ();
3  List[uml::Class] getClasses(String name, uml::Model m) :
4    m.allOwnedElements().typeSelect(uml::Class).select(e|e.qualifiedName == name);
5  List[uml::Class] getClassesFromList(List[String] names, uml::Model m) :
6    getClassesFromList(names, m, 0);
7  List[uml::Class] getClassesFromList(List[String] names, uml::Model m, int position) :
8    names.size > position + 1 ? getClasses(names.get(position), m).addAll(
           getClassesFromList(names, m, position+1)) : getClasses(names.get(position
           ), m);
9  uml::Property getProperty(String name, uml::Model m) :
10   m.allOwnedElements().typeSelect(uml::Property).select(e|e.qualifiedName == name).
           first();
```

Listing 22: Funktionen zur Ermittlung von Referenzen aus Elementnamen

Referenzen auf in Modellen enthaltene UML-Klassen werden ermittelt, indem die Namen sämtlicher Elemente eines Modells überprüft werden. Die Selektion der Modellelemente geschieht unabhängig von Verschachtelungen in Pakethierarchien durch den Befehl *allOwnedElements()*. Die Menge der Modellelemente wird reduziert auf Elemente vom Typ *uml::Class*, sodass Xtend automatisch den Rückgabetyp auf *uml::Class* setzt. Aus der Menge der Klassen werden diejenigen ausgewählt, deren Name dem gesuchten entspricht. Sinnvollerweise existiert nur eine einzige solche, die durch *first()* selektiert und zurückgegeben wird. Analog zu der Funktion *getClass* können mit der Funktion *getProperty* referenzierende Attribute aufgelöst werden.

Für Tagged Values mit einer mehrfachen Multiplizität ist es nötig, aus mehreren Namen referenzierter Elemente eine Liste mit Referenzen zu erstellen. Dies geschieht, indem eine Liste von Namen per Rekursion durchlaufen wird und während der Rekursion eine Liste mit Referenzen erstellt wird. Pro Rekursionsschritt wird mit der Funktion *getClasses* eine einelementige Liste erstellt, welche die Referenz enthält und diese an die Ergebnisliste konkateniert. Dies geschieht solange, bis die Liste der Namen durchlaufen ist.

Eine Referenz auf das Modell kann von jedem UML-Element mit der EOperation *uml::Element.getModel()* erhalten werden, die Bestandteil der EMF-basierten Implementierung des UML2-Metamodells in oAW ist.

6.5 Profil Content

In diesem und den folgenden Kapiteln werden Templates und Funktionen beschrieben, welche die Transformation von Waml-Modellelementen- bzw. konstrukten in Javaquelltext definieren.

6.5.1 Templates für Contentklassen

Die Datenhaltung der Webapplikation des Prototyps wird mit Entity Beans im Sinne von Enterprise JavaBeans 3.0 (EJB) realisiert. Entity Beans sind Java Beans, die eine spezielle Art von POJOs sind und sich durch folgende Konventionen von POJOs unterscheiden:

- Java Beans implementieren das Interface java.io.Serializable und erben nicht von anderen Klassen außer von der Klasse Object.
- Der Konstruktor besitzt die Sichtbarkeit public und ist nicht parametrisiert.
- Die Klassenattribute besitzen die Sichtbarkeit private und sind durch Get- und Set-Methoden mit der Sichtbarkeit public gekapselt.

Entity Beans werden durch das Waml-Modellelement *ContentClass* repräsentiert und durch das Template in Listing 23 in Javaquelltext transformiert.

```
1  «DEFINE Root FOR content::ContentClass»
2    «FILE getClassPath(this) backendClasses»
3      package «getPackageName(this.package)»;
4      import javax.persistence.*;
5      @Entity
6      public class «this.name» implements java.io.Serializable{
7        private static final long serialVersionUID = 1L;
8      «EXPAND templates::Class::Property FOREACH this.ownedAttribute»
9      «EXPAND Getter FOREACH this.ownedAttribute»
10     «EXPAND templates::Class::Setter FOREACH this.ownedAttribute»
11     «EXPAND ToString»
12     }
13   «ENDFILE»
14 «ENDDEFINE»
```

Listing 23: Template für Contentklassen

Das Template ähnelt dem Klassentemplate für POJOs in Listing 11 und nutzt einige Basistemplates, unterscheidet sich aber in für Entity Beans relevanten Punkten. Klassendateien von Entity Beans werden in das Outlet *backendClasses* geschrieben, in dem Klassen gesammelt werden, die zusammen in den EJB-Container auf den J2EE-Applikationsserver geladen werden. In der Klassenschnittstelle wird der Namensraum *javax.persistence* importiert, mit dessen Elementen Annotationen durchgeführt werden. Die Entity Bean wird mit *@Entity* annotiert, das die Klasse als ein eindeutig identifizierbares Domänenobjekt markiert, das auf die Datenbank abgebildet und durch einen EntityManager verwaltet werden soll. [41, S. 228] Die Entity Bean implementiert das Interface *java.io.Serializable* und enthält das Attribut *serialVersionUID*, mit dem die Klasse der Entity Bean versioniert werden kann. [1]

Attribute und Set-Methoden werden wie bei POJOs expandiert, dagegen bei Getmethoden ein spezielles Template aus Listing 24 expandiert. Auf die Transformation von Operationen wird verzichtet. Für eine angepasste toString-Methode wird das Template ToString expandiert.

```
1  «DEFINE Getter FOR content::ContentKey»
2    @Id
3    @GeneratedValue(strategy = GenerationType.AUTO)
4    «EXPAND templates::Class::Getter»
5  «ENDDEFINE»
6
7  «DEFINE Getter FOR uml::Property»
8    «IF this.association != null»
9      «getMultiplicityAnnotation(this)»(fetch=FetchType.EAGER)
10   «ENDIF»
11   «EXPAND templates::Class::Getter»
12 «ENDDEFINE»
```

Listing 24: Template für Getmethoden in Entity Beans

Bei der Transformation von Getmethoden für Eigenschaften mit Primärschlüsselcharakter wird dies mit *@Id* annotiert und durch *GeneratedValue* vorgegeben, dass Schüsselwerte automatisch generiert werden sollen. Falls Eigenschaften ohne Primärschlüsselcharakter Mitglied einer Assoziation sind [8, S. 124], wird dies über der entsprechenden Get-Methode durch eine Annotation für den Beziehungstyp vermerkt. [41, S. 239 f.] Der Beziehungstyp kann z. B. die Form *@OneToMany* besitzen.

Mit dem Ausdruck *(fetch=FetchType.EAGER)* wird die Strategie zum Laden von Daten aus der Datenbank festgelegt. Der Wert *EAGER* hat zur Folge, dass der Entitymanager beim Laden einer Contentklasse auch sämtliche assoziierten Contentklassen lädt. Diese Strategie hat den Nachteil, dass bei großen Assoziationsgraphen und vielen Datensätzen unter Umständen große Datenmengen aus der Datenbank geladen werden. Dies kann mit der alternativen Strategie *LAZY* verhindert werden, bei der nur bei Bedarf assoziierte Contentklassen aus der Datenbank geladen werden. Nachteilig an der Strategie *LAZY* ist, dass eine Exception geworfen wird, sobald assoziierte Contentklassen nachträglich vom Client bzw. aus dem Webcontainer angefordert werden, weil die Sitzung des Entitymanagers abgelaufen ist. Zum Zweck der einfacheren Generierung wird deshalb die Strategie *EAGER* gewählt.

6.5.2 Funktionen für Contentklassen

Funktionen für Contentklassen sind in der Bibliothek in Listing 25 zusammengefasst. Die Bestimmung des Strings für den Beziehnungstyp erfolgt, indem sowohl die Multiplizität der Eigenschaft als auch die Multiplizität der Eigenschaft auf der gegenüberliegenden Seite der Assoziation bestimmt wird. Die beiden Multiplizitäten

werden durch die Schlüsselworte *One* und *Many* repräsentiert, die zum Annotationsstring konkateniert werden. Zur Ermittlung des ContentKeys bzw. der ID einer Contentklasse werden die Attribute der Contentklasse nach dem entsprechenden Stereotypen durchsucht. Falls die Contentklasse mehrere Schlüssel besitzt, wird nur der zuerst gefundene gewählt. Da sowohl komplexe als auch primitive Attribute im UML-Metamodell den Typ uml::Property besitzen, wird die Unterscheidung von primitiven und komplexen Eigenschaften anhand der in der Typ-Funktionsbibliothek definierten Funktion *isPrimitive* durchgeführt.

```
1 String getMultiplicityAnnotation(uml::Property p) :
2   "@"+ (isMultiple(getOpposite(p)) ? "Many" : "One") + "To" + (isMultiple(p) ? "Many"
            : "One");
3 uml::Property getOpposite(uml::Property p) :
4   p.association.memberEnd.select(e|e!=p).first();
5 content::ContentKey getContentKey(content::ContentClass c) :
6   c.ownedAttribute.typeSelect(content::ContentKey).first();
7 List[uml::Property] getComplexAttributes(content::ContentClass c) :
8   c.ownedAttribute.select(e|!e.type.isPrimitive());
```

Listing 25: Funktionen für Contentklassen

6.5.3 Templates für Geschäftsklassen

Geschäftslogik wird in Enterprise JavaBeans 3.0 (EJB) in Session Beans abgebildet, die durch das Waml-Modellelement *BusinessClass* repräsentiert werden und durch das Template in Listing 26 in Javaquelltext transformiert werden. Bei der Expandierung wird überprüft, ob das Modellelement der Businessklasse ein Remote- oder ein Localinterface implementiert. Diese Informationen werden in den Tagged Values *BusinessClass.isLocal* und *BusinessClass.isRemote* als Primitive vom UML-Typ Boolean gespeichert. Durch die beiden Interfaces wird dem EJB-Container mitgeteilt, ob auf eine Session Bean lokal auf dem Server oder clientseitig außerhalb des Servers zugegriffen werden kann und welche Methoden der Session Bean aufgerufen werden können. [41, S. 47 f.]

```
1 «DEFINE Root FOR content::BusinessClass»
2   «EXPAND BusinessClass»
3   «IF this.isRemote»
4     «EXPAND RemoteInterface»
5   «ENDIF»
6   «IF this.isLocal»
7     «EXPAND LocalInterface»
8   «ENDIF»
9 «ENDDEFINE»
```

Listing 26: Verzweigungstemplate für Businessklassen und Interfaces

Das Template für die Generierung des Quelltextes der Businessklasse in Listing 27 speichert analog zu Entity Beans generierte Session Beans im Outlet *backendClasses*. Der Tagged Value *isStateful* drückt aus, ob der EJB-Container sicherstellen soll, dass die Session Bean zwischen Methodenaufrufen ihren clientspezifischen Zustand der Instanzvariablen behält. [41, S. 50 ff.] Dies wird entsprechend am Klassenkopf durch *@Stateful* oder *@Stateless* annotiert. Da der Zustand von zustandsbehafteten Session Beans zwischen den Methodenaufrufen gespeichert werden muss, implementieren Session Beans das Interface *java.io.Serializable*. Mit dem Tagged Value *hasEntityManager* wird festgelegt, ob ein EntityManager für Zugriffe auf die Persistenzschicht benötigt wird. Dieser bildet die Objekte auf ein relationales Schema ab und wird durch die Annotation *@PersistenceContext* in die Variable *em* injiziert. Falls ein EntityManager gefordert ist, wird das entsprechende annotierte Attribut generiert. Die der Session Bean zugeordneten Attribute und deren Zugriffsmethoden sowie die Operationen werden durch die Basisklassentemplates expandiert.

```
1  «DEFINE BusinessClass FOR content::BusinessClass»
2    «FILE getClassPath(this) backendClasses»
3      package «getPackageName(this.package)»;
4      «IF this.hasEntityManager»
5        import javax.persistence.*;
6      «ENDIF»
7      import javax.ejb.*;
8      @SuppressWarnings("serial")
9      «IF this.isStateful»
10       @Stateful
11     «ELSE»
12       @Stateless
13     «ENDIF»
14     public class «getClassName(this)» implements java.io.Serializable
15     «IF this.isRemote»
16       , «getRemoteInterface(this)»
17     «ENDIF»
18     «IF this.isLocal»
19       , «getLocalInterface(this)»
20     «ENDIF»
21     {
22     «IF this.hasEntityManager»
23       @PersistenceContext(unitName="em")
24       private EntityManager em;
25     «ENDIF»
26     «EXPAND templates::Class::Property FOREACH this.ownedAttribute»
27     «EXPAND Operation FOREACH this.ownedOperation»
28     «EXPAND templates::Class::Getter FOREACH this.ownedAttribute»
29     «EXPAND templates::Class::Setter FOREACH this.ownedAttribute»
30     }
31   «ENDFILE»
32 «ENDDEFINE»
```

Listing 27: Template für Businessklassen

Die lokalen Interfaces werden durch die Templates in Listing 28 generiert. Im Wesentlichen werden bei lokalen Interfaces die lokalen Operationen der Businessklasse selektiert und entsprechend im Interface Methoden deklariert. Sämtliche lokalen Operationen müssen als solche im Modell mit dem Stereotyp *LocalOperation* kenntlich gemacht werden.

```
1  «DEFINE LocalInterface FOR content::BusinessClass»
2    «FILE getLocalInterfacePath(this) backendClasses»
3      package «getPackageName(this.package)»;
4      @javax.ejb.Local
5      public interface «getLocalInterface(this)»
6      {
7      «EXPAND Accessors FOREACH this.ownedAttribute.typeSelect(content::LocalProperty)»
8      «EXPAND Operation FOREACH this.ownedOperation.typeSelect(content::LocalOperation)
           »
9      }
10   «ENDFILE»
11 «ENDDEFINE»
12
13 «DEFINE Accessors FOR uml::Property»
14   public «getTypeName(this)» «getGetterName(this)»();
15   public void «getSetterName(this)»(«getTypeName(this)» «getParameterName(this)»);
16 «ENDDEFINE»
```

Listing 28: Templates für Interfaces von Businessklassen

Die Remoteinterfaces werden strukturgleich zu den lokalen generiert. Auch wird auf die Darstellung der trivialen Funktionen zur Bildung von Namens- und Pfadstrings von Interfaces verzichtet.

Für die Stereotypen zur Auszeichnung von Methoden in Geschäftsklassen sind in Listing 29 entsprechende Templates definiert, mit denen die Methoden vollständig inklusive der Methodenrümpfe generiert werden können. Die Methode zum Speichern übergibt dem Entitymanager den ersten Parameter der Operation zur Persistierung. Die Löschmethode sucht anhand der übergebenen ID die zu löschende Contentklasse und fordert vom Entitymanager die Löschung an. Die Methode zum Auflisten erstellt eine Query zur Selektion der Instanzen einer Contentklasse und gibt diese als Ergebnisliste zurück.

```
1  «DEFINE Operation FOR uml::Operation»
2    «EXPAND templates::Class::Operation»
3  «ENDDEFINE»
4
5  «DEFINE Operation FOR content::bcLoad»
6    «getVisibility(this)» «getTypeName(this)» «this.name»(«EXPAND templates::Class::
          Parameter FOREACH getParameterList(this) SEPARATOR ","»){ return this.em.
          find(«getTypeName(this)».class, id); }
7  «ENDDEFINE»
8
9  «DEFINE Operation FOR content::bcSave»
```

```
10    ...{em.merge(«getParameterList().first().name»);}
11  «ENDDEFINE»
12
13  «DEFINE Operation FOR content::bcDelete»
14    ...{this.em.remove(this.em.find(«getTypeName(getParameterList(this).first())».class
              , «getParameterList(this).first().name».getId()));}
15  «ENDDEFINE»
16
17  «DEFINE Operation FOR content::bcList»
18    ...{Query q = this.em.createQuery("FROM «this.type.name»"); return («getTypeName(
              this)») q.getResultList();}
19  «ENDDEFINE»
```

Listing 29: Templates für Operationen von Geschäftsklassen

6.5.4 Funktionen für Geschäftsklassen

Die Funktionsbibliothek für Geschäftsklassen enthält unter anderem Funktionen zur Suche nach stereotypisierten Operationen wie z. B. Ladeoperationen. Die Funktion *getControlledContentClass* ermittelt die Contentklasse einer Geschäftsklasse anhand des Tagged Value *contentClass* und mit der Funktion zur Umwandlung von Stringreferenzen in Modellelemente aus der Funktionsbibliothek für Tagged Values. Die Funktion *getPrimaryBusinessClass* ermittelt aus allen Geschäftsklassen des Modells diejenige für eine Contentklasse, die sie verwaltet. Da eine Contentklasse durch mehrere Geschäftsklassen referenziert werden kann, wird die zuerst gefundene zurückgegeben.

```
1  content::bcLoad getOperationBcLoad(content::BusinessClass c) :
2    c.ownedOperation.typeSelect(content::bcLoad).first();
3  content::ContentClass getControlledContentClass(content::BusinessClass c) :
4    (content::ContentClass) getClass(c.contentClass, c.getModel());
5  content::BusinessClass getPrimaryBusinessClass(content::ContentClass c) :
6    c.getModel().allOwnedElements().typeSelect(content::BusinessClass).select(e|e.
             contentClass.getClass(c.getModel()) == c).first();
7  ...
```

Listing 30: Funktionen für Geschäftsklassen

6.6 Profil Control

Durch das MVC-Schema werden im Frontend Darstellungsaspekte, Steuerungsaspekte und Aspekte der Anwendungslogik bzw. Datenhaltung getrennt behandelt. [46, S. 106] Steuerungsaspekte werden in WAML durch die Sprachelemente *Node* und *Controller* modelliert, deren Templates im Folgenden erläutert werden.

6.6.1 Templates für Controller

Controller sind im Sinne von WAML Klassen, die Daten aus Geschäftsklassen abrufen und für die darstellenden *Views* aufbereiten. Die Funktionalität ist beschränkt auf Datentransformationen wie z. B. Typumwandlungen. Zudem ist die Lebensdauer des Controllerzustandes auf die Laufzeit der Benutzersitzung begrenzt, eingegebene Daten müssen also gesondert gespeichert werden. Eine dauerhafte Persistierung und Abbildung der Geschäftslogik sollte ausschließlich durch Geschäftsklassen angeboten werden.

Jedem Controller ist in WAML explizit eine Geschäftsklasse zugeordnet, der wiederum explizit eine Contentklasse zugeordnet ist. Jeder Controller verwaltet somit eine implizit im WAML-Modell zugeordnete Contentklasse, für deren Verarbeitung er Methoden anbietet. In den Templates wird die implizite Zuordnung expliziert, indem der Controllerklasse eine Referenz auf die Contentklasse hinzugefügt wird. Auf der referenzierten Contentklasse wird durch stereotypisierte Methoden Funktionalität definiert, mit der z. B. die Contentklasse persistiert werden kann. Die Methoden werden durch Views aufgerufen.

Im Controller-Template in Listing 31 wird zuerst ein Template für die implizit zugeordnete Contentklasse expandiert. Für Attribute und Operationen der Controllerklasse werden anhand von Stereotypen passende Templates ausgewählt und expandiert, um unter Verzicht auf geschützte Bereiche auch die Methodenrümpfe komplett zu generieren. Um Assoziationen in Views ausgeben zu können, müssen diese für JSF in SelectItems verpackt werden. Ein Anwendungsfall dafür ist die Auswahlmöglichkeit eines Autors für ein Buch mit einer Drop-Down-Box. Um die Drop-Down-Box mit Werten zu befüllen und den ausgewählten Autor zu speichern, müssen zur Auflistung und Selektion die Objekte der Personen-Contentklassen zuvor vom Controller in SelectItem-Instanzen verpackt werden. Diese Funktionalität wird vom Template *SelectItemAccessors* generiert. Um Methoden von Session Beans bzw. Geschäftsklassen aufrufen zu können, müssen diese vorab im JBoss-Server über das Java Naming and Directory Interface (JNDI) lokalisiert werden. Die Methoden zur Lokalisierung werden durch das Template *SessionBeanFinder* generiert.

```
1  «DEFINE Root FOR control::Controller»
2    «FILE getClassPath(this) warClasses»
3      package «getPackageName(this.package)»;
4      public class «this.name»{
5        «EXPAND ControlledContentClass FOR getControlledContentClass(this)»
6        «EXPAND Property(this) FOREACH this.ownedAttribute»
7        «EXPAND Operation(this) FOREACH this.ownedOperation»
8        «EXPAND SelectItemAccessors FOREACH getControlledContentClass(this).
                     getComplexAttributes()»
```

```
9         «EXPAND SessionBeanFinder FOREACH getAllBusinessClasses(this.getModel())»
10      }
11    «ENDFILE»
12  «ENDDEFINE»
```

Listing 31: Template für Controller

Das Template *ControlledContentClass* in Listing 32 generiert eine automatisch initialisierte Objektvariable für die referenzierte Contentklasse. Zudem werden Get- und Set-Methoden für die Objektvariable generiert. Bei der Modellierung sollte beachtet werden, dass jede Contentklasse bzw. deren Geschäftsklasse nur von einem Controller referenziert wird, um eine eindeutige Zuordnung zu erzielen. Während der Laufzeit werden die Controllerklassen durch JSF instanziiert und in der Benutzersitzung gespeichert. Indem automatisch bei der Instanziierung der Controllerklassen die assoziierten Contentklassen instanziiert werden, muss dies nicht mehr durch JSF vorgenommen werden. Da die Instanziierung von Klassen, wie in Kapitel 6.6.4 beschrieben, durch JSF in den Deployment Descriptoren von JSF verwaltet wird, kann somit die Generierung der Deployment Descriptoren vereinfacht werden.

```
1  «DEFINE ControlledContentClass FOR content::ContentClass»
2    private «getPackagedClassName(this)» «this.name» = new «getPackagedClassName(this)»
           ();
3    public «getPackagedClassName(this)» get«this.name.toFirstUpper()»(){
4      return this.«this.name»;
5    }
6    public void set«this.name.toFirstUpper()»(«getPackagedClassName(this)» «this.name»)
           {
7      this.«this.name» = «this.name»;
8    }
9  «ENDDEFINE»
```

Listing 32: Template für Contentklasse eines Controllers

Die Funktionalität für die Auflistung sämtlicher Contentklassen eines Typs wie z. B. Personen wird durch Controllerattribute mit dem WAML-Stereotyp *cList* repräsentiert. Für Attribute mit diesem Stereotyp werden zwei Get-Methoden generiert, die von der zugeordneten Geschäftsklasse die Listenmethode mit dem Stereotyp *bcList* aufrufen. Um die Listenmethode der Geschäftsklasse aufrufen zu können, muss diese vorab durch einen JNDI-Lookup lokalisiert werden. Diese Funktionalität ist in Methoden mit dem Namensmuster *getSessionBean<Geschäftsklassenname>* ausgelagert.

In der ersten Methode in Zeile 2 wird die Lookup-Funktion ausgewählt und das Ergebnis als Collection zurückgegeben. Durch die Funktion *getOperationBcList* wird die Listenmethode der Session Bean ermittelt, indem nach einer Operation mit dem Stereotyp *bcList* gesucht wird.

In der zweiten Methode in Zeile 5 wird ähnlich der ersten Methode von der Session Bean eine Liste von Contentklassen eines Typs abgerufen und diese als Liste von Elementen vom Typ *javax.faces.model.SelectItem* zurückgegeben. Die Transformation geschieht in einer for-Schleife. Die SelectItems enthalten eine ID und einen Bezeichner, die in der Präsentationsschicht z . B. bei Drop-Down-Boxen ausgegeben werden. Der Bezeichner dient der Benennung der Elemente der Drop-Down-Boxen, mit der ID kann bei der Selektion eines Wertes die entsprechende Contentklasse ermittelt werden. JSF hat den Nachteil, dass es keine bei GUI-Frameworks üblichen Drop-Down-Boxes anbietet, bei denen die auszuwählenden Objekte direkt referenziert werden. Die Referenz muss generell über den typunsicheren Umweg von primitiven IDs ermittelt werden. Das ID-Attribut der Contentklassen wird durch die Funktion *getContentKey* ermittelt.

```
1  «DEFINE Property(control::Controller c) FOR control::cList»
2    public «getTypeName(this)» get«this.name.toFirstUpper()»(){
3      return this.getSessionBean«getClassName(getControlledBusinessClass(c))»().
               «getOperationBcList(getControlledBusinessClass(c)).name»();
4    }
5    public java.util.Collection<javax.faces.model.SelectItem> get«this.name.
              toFirstUpper()»AllAsSelectItems(){
6      java.util.Collection<javax.faces.model.SelectItem> result = new java.util.
               ArrayList<javax.faces.model.SelectItem>();
7      «getTypeName(this)» records = this.getSessionBean«getClassName(
               getControlledBusinessClass(c))»().«getOperationBcList(
               getControlledBusinessClass(c)).name»();
8      for(«getQualifiedTypeName(this.type)» record: records){
9        result.add(new javax.faces.model.SelectItem(record.«getGetterName(getContentKey
                 (getControlledContentClass(c)))»(), record.toString()));
10     }
11     return result;
12   }
13 «ENDDEFINE»
```

Listing 33: Template für Listenoperation eines Controllers

Analog zu Attributen kann in WAML die für Operationen von Controllern vordefinierte Funktionalität durch Stereotypen annotiert werden. Die Methoden sind generell parametrisiert mit einem Ereignis vom Typ *javax.faces.event.ActionEvent*, da dies von JSF gefordert wird.

Die Methode zum Laden einer Instanz einer Contentklasse ermittelt aus dem *FacesContext* die mit dem HTTP-Seitenaufruf verbundenen GET und POST Parameter und wählt aus diesen den Parameter mit dem Namen *id* aus. Mittels der ID wird von der bcLoad-Methode der Session Bean des Controllers der entsprechende Datensatz abgerufen und in die Objektvariable des Controllers geschrieben. Im Template in Listing 34 wird die Contentklasse bzw. deren Session Bean anhand des

Tagged Value *Controller.businessClass* durch die Funktion *getControlledContent-Class* ermittelt. Der Rückgabetyp der Methode wird anhand des Rückgabetyps der cLoad-Operation ermittelt und muss mit dem Typ der Contentklasse übereinstimmen.

```
1  «DEFINE Operation(control::Controller c) FOR control::cLoad»
2    «getVisibility(this)» «getStatic(this)» «getTypeName(this)» «this.name»(javax.faces
             .event.ActionEvent event){
3      javax.faces.context.FacesContext facesContext = javax.faces.context.FacesContext.
               getCurrentInstance();
4      java.util.Map params = facesContext.getExternalContext().getRequestParameterMap()
               ;
5      int id = Integer.parseInt(params.get("id").toString());
6      this.«getClassName(getControlledContentClass(c))» = this.
               getSessionBean«getClassName(getControlledBusinessClass(c))»().
               «getOperationBcLoad(getControlledBusinessClass(c)).name»(id);
7    }
8  «ENDDEFINE»
```

Listing 34: Template für Ladeoperation eines Controllers

Die Methode zum Speichern von Contentklassen wird nach Listing 35 generiert, indem die Session Bean der Contentklasse ermittelt wird und auf ihr die bcSave-Operation mit der aktuell geladenen Contentklasse des Controllers als Parameter aufgerufen wird. Nach der Speicherung wird der Zustand des Controllers bereinigt, indem der Wert der Contentklasse zurückgesetzt wird. Dies verhindert die ungewollte und unerwartete Ausgabe der gespeicherten Daten in einem anderen Seitenkontext. Die cReset-Methode muss im Modell explizit an den Controller modelliert werden und als solche mit einem Stereotypen annotiert werden. Alternativ wäre es auch möglich, die Reset-Methode zu generieren, ohne sie im Modell zu explizieren. Dies hätte eine Reduktion des Informationsgehaltes des Modells zur Folge und ist nicht sinnvoll, da die öffentliche Reset-Methode einen zu dokumentierenden Aussagegehalt besitzt und keine rein technische und private Hilfsmethode des Controllers ist.

```
1  «DEFINE Operation(control::Controller c) FOR control::cSave»
2    «getVisibility(this)» «getStatic(this)» «getTypeName(this)» «this.name»(javax.faces
             .event.ActionEvent event){
3      this.getSessionBean«getClassName(getControlledBusinessClass(c))»().
               «getOperationBcSave(getControlledBusinessClass(c)).name»(this.
               «getClassName(getControlledContentClass(c))»);
4      this.«getOperationCReset(c).name»(null);
5    }
6  «ENDDEFINE»
```

Listing 35: Template für Speicheroperation eines Controllers

Die Methode zum Löschen von Contentklassen wird nach Listing 36 so generiert, dass die Session Bean einer Contentklasse ermittelt wird und deren bcDelete-Methode

mit der Contentklasse als Parameter aufgerufen wird. Nach dem Aufruf wird im Controller die Contentklasse zurückgesetzt.

```
1 «DEFINE Operation(control::Controller c) FOR control::cDelete»
2   «getVisibility(this)» «getStatic(this)» «getTypeName(this)» «this.name»(javax.faces
          .event.ActionEvent event){
3     this.getSessionBean«getClassName(getControlledBusinessClass(c))»().
            «getOperationBcDelete(getControlledBusinessClass(c)).name»(this.
            «getClassName(getControlledContentClass(c))»);
4     this.«getOperationCReset(c).name»(null);
5   }
6 «ENDDEFINE»
```

Listing 36: Template für Löschoperation eines Controllers

Die Generierung der Reset-Methode erfolgt entsprechend Listing 37, indem die Contentklasse ermittelt wird und ihre Objektvariable im Controller durch eine neue Instanz überschrieben wird. Dieser und den vorigen Operationen ist gemein, dass die modellierten Methoden generell den Rückgabetyp *void* besitzen müssen, da die Templates für Methodenrümpfe keine Wertrückgaben vorsehen. Diese Restriktion muss gesondert außerhalb der Templates durch Constraints erfasst werden.

```
1 «DEFINE Operation(control::Controller c) FOR control::cReset»
2   «getVisibility(this)» «getStatic(this)» «getTypeName(this)» «this.name»(javax.faces
          .event.ActionEvent event){
3     this.«getControlledContentClass(c).name» = new «getPackagedClassName(
            getControlledContentClass(c))»();
4   }
5 «ENDDEFINE»
```

Listing 37: Template für Resetoperation eines Controllers

Da der Prototyp auf den JBoss-Server in zwei verschiedene Container getrennt ausgeliefert wird, müssen Session Beans im Controller vor dem Aufruf lokalisiert werden. Der Methodenname wird nach dem Schema *getSessionBean*$<$*Geschäftsklassenname*$>$ generiert. Die Sessionbeanfinder-Methoden werden entsprechend Listing 31 durch das Template in Listing 38 für sämtliche Geschäftsklassen im Modell erzeugt. Da der Geschäftsklassenname nur den Klassennamen ohne Beachtung von Paketen enthält, treten Kollisionen bei der Benennung der Methoden auf, falls zwei gleichnamige Geschäftsklassen in zwei verschiedenen Paketen liegen. Generell werden die Methoden in jeder Controllerklasse neu generiert. Diese Redundanz im Zielsystem könnte vermieden werden, indem die Methoden in einer dedizierten Klasse zusammengefasst werden, wodurch aber die Templates verkompliziert werden. Ob die Metaprogrammierung oder die Artefakte der Zielsprache vereinfacht werden sollen, ist eine Frage, die projektbezogen bewertet werden muss. Da der Prototyp vollständig automatisiert und ohne geschützte Regionen erzeugt wird, wird hier Ersteres gewählt.

```
1 «DEFINE SessionBeanFinder FOR content::BusinessClass»
2   private «getPackagedClassName()»Local getSessionBean«getClassName(this)»() throws
            javax.naming.NamingException{
3     javax.naming.Context ic = new javax.naming.InitialContext();
4     «getPackagedClassName()»Local sb = («getPackagedClassName()»Local) ic
5       .lookup("prototype/«getClassName(this)»/local");
6     return sb;
7   }
8 «ENDDEFINE»
```

Listing 38: Template für Sessionbeanfinder

6.6.2 Funktionen für Controller

Funktionen mit Bezug zu Controller-Sprachelementen sind in einer eigenen Funktionsbibliothek entsprechend Listing 39 zusammengefasst. Die Funktion *getControlledBusinessClass* zur Referenzierung der Geschäftsklasse eines Controllers greift auf die Funktionen für komplexe Tagged Values in Listing 22 zurück. Aus dem String-basierten Tagged Value wird mit *getClass* eine Referenz auf die entsprechende Klasse ermittelt und als Geschäftsklasse gecastet. Typfehler können beim Casting nicht auftreten, weil der Tagged Value bei der Modellierung in Magic Draw typsicher nur mit Geschäftsklassen belegt werden kann.

Die durch den Controller kontrollierte Contentklasse wird durch die Funktion *getControlledContentClass* indirekt über die Geschäftsklasse des Controllers ermittelt.

Bei der Funktion *getPrimaryController* wird für eine Contentklasse diejenige Controllerklasse zurückgegeben, welche die Contentklasse verwaltet. Dazu wird das Modell nach sämtlichen Controllerklassen durchsucht und aus den Controllerklassen diejenige selektiert, deren Geschäftsklasse mit der primären Geschäftsklasse der Contentklasse übereinstimmt. Somit ist diese Funktion invers zu der Funktion *getControlledContentClass*. Die Navigation durch das Modell ist in diesem Fall so kompliziert, weil die Tagged Values in WAML unidirektional top-down definiert sind, d. h. bei einer Navigation in der Richtung bottom-up muss nach externen Referenzen von Geschäftsklassen auf die Contentklasse gesucht werden. Benötigt wird die Funktion *getPrimaryController* insbesondere bei Templates für Views, um Drop-Down-Boxen zu befüllen. Dies wird z. B. in Kapitel 6.7.1 erläutert.

Exemplarisch für Funktionen zum Finden von stereotypisierten Attributen und Operationen sind die Funktionen *getOperationCSave* und *getPropertyCList* definiert, die unter den Operationen bzw. Attributen eine Selektion aufgrund des Stereotyps vornehmen. Ein Cast ist nicht notwendig, weil die typeSelect-Funktion automatisch

eine Liste mit Werten vom gesuchten Typ zurückgibt. Die Liste muss auf das erste Element reduziert werden, d. h. dass ein Controller jeweils nur eine Operation eines Stereotyps besitzen sollte. Dies ist generell ein Problem von Stereotypen, da die Anzahl von durch Stereotypen ausgezeichneten Elementen nicht strukturell durch das Metamodell restringiert werden kann. Als Lösung müssen entweder Constraints formuliert werden, welche die Anzahl prüfen, oder es werden unter Verzicht auf Stereotypen nur Tagged Values benutzt, bei denen im Metamodell die Multiplizität festgelegt werden kann. So würde beim zweitgenannten der Controller Tagged Values für Operationen besitzen, die eine einfache Multiplizität haben. Nachteilig ist bei Tagged Values generell, dass referenzierende Tagged Values in der Modellierungsumgebung nicht graphisch annotiert werden. Zudem müsste durch ein Constraint festgelegt werden, dass die durch den Tagged Value referenzierte Operation zu der Controllerklasse des Tagged Value gehört.

```
1  content::BusinessClass getControlledBusinessClass(control::Controller c) :
2    (content::BusinessClass) getClass(c.businessClass, c.getModel());
3  content::ContentClass getControlledContentClass(control::Controller c) :
4    (content::ContentClass) getClass(getControlledBusinessClass(c).contentClass, c.
             getModel());
5  control::Controller getPrimaryController(content::ContentClass c) :
6    c.getModel().allOwnedElements().typeSelect(control::Controller).select(e|e.
             businessClass.getClass(c.getModel()) == getPrimaryBusinessClass(c)).first
             ();
7  control::cSave getOperationCSave(control::Controller c) :
8    c.ownedOperation.typeSelect(control::cSave).first();
9  control::cList getPropertyCList(control::Controller c) :
10   c.ownedAttribute.typeSelect(control::cList).first();
```

Listing 39: Controllerfunktionen

6.6.3 Templates für Navigationsknoten

Navigationspfade von Webapplikationen werden in WAML durch Aktivitätsdiagramme modelliert. Navigationsknoten werden durch Webseiten repräsentiert, zwischen denen navigiert werden kann, falls dies durch Kanten im Aktivitätsdiagramm kenntlich gemacht ist. Für Navigationsknoten ist in Listing 40 ein Template definiert, das die Grundstruktur von Webseiten der Webapplikation beschreibt. Dem Knoten wird mit dem Tagged Value *view* im Modell, wie in Kapitel 5 erläutert, Darstellungsfunktionalität zugewiesen, die durch das entsprechende View-Template generiert wird. Die verschiedenen Formen von View-Templates werden in Kapitel 6.7 behandelt. Falls der Knoten keine View referenziert, wird eine geschützte Region in die Seite eingefügt, in die manuell anstelle der View Seiteninhalte eingefügt werden können.

```
1  «DEFINE Root FOR control::Node»
```

```
2    «FILE getNodeJspxFilename(this) warJsp»
3      «EXPAND Header::Header»
4      <h1>«this.name»</h1>
5      «IF this.view.length > 0»
6        «EXPAND templates::waml::view::Root::Root(this) FOR getView(this)»
7      «ELSE»
8        «PROTECT CSTART "<!-- " CEND " -->" ID getNodeId(this)»
9        «ENDPROTECT»
10     «ENDIF»
11     «EXPAND Footer::Footer»
12   «ENDFILE»
13 «ENDDEFINE»
```

Listing 40: Template für Knoten

Das Template für den Header der Webseiten enthält einleitende HTML-Tags sowie JSP-Tags zu den verwendeten JSF-Taglibraries. Außerdem wird im Headertemplate für jeden Knoten das Template für die Generierung des Menüs entsprechend dem Template in Listing 41 expandiert. Dieses generiert das Menü als eine Liste von Menüelementen, die in einem *JSF-Form* mit *JSF-commandLinks* auf die Seiten verweisen. Die Menüelemente werden aus denjenigen Knoten erstellt, deren Tagged Value *menuPriority* mit einem Zahlenwert belegt ist.

```
1 «DEFINE Root FOR uml::Element»
2   <h:form>
3     <ul>
4     «FOREACH menuEntryNodes(this.getModel()) AS Node»
5       <li><h:commandLink action="menu:«getNodeIfaceId(Node)»" value="«Node.name»"/></
            li>
6     «ENDFOREACH»
7     </ul>
8   </h:form>
9 «ENDDEFINE»
```

Listing 41: Menütemplate

Die Menüelemente können im Modell anhand der Zahlenwerte der Menüpriorität sortiert werden, was einen dezentralen und einfachen Ansatz für die Generierung eines eindimensionalen Menüs darstellt.

```
1 List[control::Node] menuEntryNodes(uml::Model m) :
2   m.allOwnedElements().typeSelect(control::Node).select(e|e.menuPriority > 0).sortBy(
          e|e.menuPriority);
```

Listing 42: Funktion zur Menüelementselektion

Ein besonderer Knotentyp ist der initiale Knoten, der durch das UML-Element *InitialNode* repräsentiert wird. Falls das Navigationsmodell einen initialen Knoten mit einer ausgehenden Kante zu einem WAML-Knoten enthält, wird für den initialen Knoten eine *index.jsp* generiert, die eine Umleitung auf die Webseite des referenzierten Knotens enthält. Dabei ist zu beachten, dass die Weiterleitung nicht wie bei

<jsp:forward> serverseitig vorgenommen wird, sondern der Browser im Header der initialen Webseite per HTTP Code 301 [5] angewiesen wird, die Seite zu wechseln. Dies ist wichtig, da damit ein Wechsel der Seite im Browser einhergeht, bei dem je nach Pfad der Zielseite das Verzeichnis in der Webapplikation gewechselt wird. Falls die referenzierte Webseite relative Links enthielte, würden diese bei einem serverseitigen Seitenwechsel auf das falsche Verzeichnis verweisen. Zu den relativen Referenzen gehört z. B. auch der relative Link auf Stylesheets.

```
1 «DEFINE Root FOR uml::InitialNode»
2   «FILE "index.jsp" warJsp»
3 <% response.sendRedirect(".«getNodeIfaceId(this.outgoing.target.typeSelect(control::
          Node).first())»"); %>
4 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
5 <html><body></body></html>
6   «ENDFILE»
7 «ENDDEFINE»
```

Listing 43: Template für initiale Knoten

6.6.4 Templates für Descriptoren der Navigation

Die Navigation wird in Java ServerFaces (JSF) durch Navigationsregeln erfasst, die in der Descriptoren-Datei *faces-config.xml* gespeichert werden. In JSF können statische sowie dynamische Navigationsstrukturen abgebildet werden. [24] Bei statischer Navigation wird bei dem Klicken auf Links und Formbuttons generell dieselbe Folgeseite geöffnet. Bei dynamischer Navigation wird die Folgeseite vom MVC-Controller je nach dem Ergebnis einer ausgeführten Operation ausgewählt. Z. B. kann nach dem Abschicken eines Formulars vom Controller entschieden werden, ob die übermittelten Formulardaten inkonsistent sind und eine fallweise Umleitung auf eine Korrekturseite initiieren. Technisch wird die dynamsiche Navigation umgesetzt, indem die Controllermethode, welche die Formulardaten verarbeitet, nach der Verarbeitung einen Ergebnisstring zurückgibt, der in der faces-config.xml als Outcome einem Navigationsfall zugeordnet ist. Zu jeder Seite existiert eine Navigationsregel, in der für die Outcomes Zielseiten definiert sind. Je nach dem Ergebnisstring der Controllermethode wird der entsprechende Navigationsfall selektiert und die dort angegebene Seite vom JSF-Controller geladen.

Das Template für die Generierung der faces-config.xml in Listing 44 unterstützt dynamische Navigationsstrukturen, indem der Name einer Kante als Stringwert des Outcome interpretiert wird. Aus den ausgehenden Kanten eines Navigationsknotens werden Navigationsfälle dieses Knotens generiert. Für sämtliche Knoten, die ausgehende Knoten besitzen, wird eine Navigationsregel angelegt. Für jede ausgehende

Kante der Knoten wird ein Navigationsfall mit dem Namen der Kante als Outcome definiert. Falls zudem Knoten von Menüelementen referenziert werden, muss auf jeder Seite für jede Menüreferenz ein Navigationsfall definiert werden, damit von jeder Seite aus das Menü benutzt werden kann. Der Outcome wird bei diesen Navigationsfällen automatisch aus dem Präfix *menu:* und dem Namen der Zielseite der Menüreferenz konkateniert und gleichermaßen in der Generierung des Menüs, wie in Listing 41 gezeigt, angegeben. Alternativ müssten die menübasierten Navigationspfade im Modell als Kanten expliziert werden, was ein unübersichtliches Navigationsmodell zur Folge hätte.

```
1  «DEFINE Root FOR uml::Model»
2    «FILE "faces-config.xml" warWEBINF»<?xml version="1.0" encoding="UTF-8"?>
3      ...
4      <faces-config>
5        «FOREACH this.allOwnedElements().typeSelect(control::Node) AS n»
6          «IF n.outgoing.size > 0 || menuEntryNodes(this.getModel()).size > 0»
7            <navigation-rule>
8              <from-view-id>«getNodeIfaceId(n)»</from-view-id>
9              «FOREACH n.outgoing AS o»
10               <navigation-case>
11                 <from-outcome>«o.name»</from-outcome>
12                 <to-view-id>«getNodeIfaceId(o.target)»</to-view-id>
13               </navigation-case>
14             «ENDFOREACH»
15
16             «FOREACH menuEntryNodes(this.getModel()) AS MenuNode»
17               <navigation-case>
18                 <from-outcome>menu:«getNodeIfaceId(MenuNode)»</from-outcome>
19                 <to-view-id>«getNodeIfaceId(MenuNode)»</to-view-id>
20               </navigation-case>
21             «ENDFOREACH»
22           </navigation-rule>
23         «ENDIF»
24       «ENDFOREACH»
25     </faces-config>
26   «ENDFILE»
27 «ENDDEFINE»
```

Listing 44: Template für die facesconfig.xml

In der faces-config.xml werden neben den Navigationsregeln auch die durch den JSF-Controller verwalteten Managed Beans unter Angaben eines Namens und der Lebenszeit registriert. Managed Beans sind Java Beans, auf die unter ihrem zugewiesenen Namen in Seiten bzw. JSF-Views in der Form *#{beanname.attributsname}* zugegriffen werden kann. Das Template für die faces-config.xml enthält entsprechend Listing 45 einen Abschnitt, in dem jeder WAML-Controller als Managed Bean registriert wird. Die Lebensdauer der Managed Beans erstreckt sich auf die Dauer der Benutzersitzung. Da in dem Controller eine Referenz auf die kontrollierte Content-

klasse existiert und der Controller in der Benutzersitzung seitenübergreifend seinen Zustand behält, wird auf der Präsentationsschicht die Entwicklung von Wizards unterstützt, die seitenübergreifende inkrementelle Modifikationen an der Contentklasse des Controllers vornehmen. Z. B. ist es damit möglich, einen Personendatensatz über mehrere Seiten hinweg zu pflegen, ohne den Datensatz für jeden Seitenaufruf neu laden zu müssen.

Des Weiteren bietet die Speicherung der Contentklasse in der Benutzersitzung Vorteile bzgl. der Datensicherheit. Im Fall eines rein request-basierten Wizards muss die ID des zu bearbeitenden Datensatzes auf jeder Webseite des Wizards zumindest als verstecktes Eingabeelement angegeben werden. Damit befindet sich die Angabe der ID unter der Kontrolle des Clients und kann modifiziert werden. Eine Speicherung des Datensatzes unter einer modifizierten ID könnte z. B. bei fehlerhafter Programmierung Schreibvorgänge auf Datensätze ermöglichen, für die der Client eigentlich nicht autorisiert sein sollte.

Nachteilig ist der erhöhte serverseitige Speicherbedarf zu sehen, da auf dem J2EE-Server unter Hochlast und bei vielen und großen Managed Beans Sitzungsdaten von relevanter Größe und Menge anfallen können.

```
1  ...
2  <faces-config>
3    «FOREACH this.allOwnedElements().typeSelect(control::Controller) AS c»
4      <managed-bean>
5        <managed-bean-name>«getManagedBeanName(c)»</managed-bean-name>
6        <managed-bean-class>«getPackagedClassName(c)»</managed-bean-class>
7        <managed-bean-scope>session</managed-bean-scope>
8        </managed-bean>
9    «ENDFOREACH»
10   ...
11 </faces-config>
12 ...
```

Listing 45: Template für die facesconfig.xml

6.7 Profil View

Knoten in WAML werden in Webseiten expandiert und enthalten, wie in Kapitel 6.6.3 erläutert, einen Tagged Value für die referenzierte View. Durch die Unterscheidung von Nodes, Views und Controllern wird, wie in Kapitel 5 beschrieben, eine Trennung von Navigationsaspekten, Darstellungsfunktionalität und operativer Funktionalität erzielt. Views repräsentieren Darstellungsfunktionalität, die für die Interaktivität der Webapplikation die operationale Funktionalität der Controller benutzt. Generell können Präsentationselemente wie z. B. Formulare und Tabellen ma-

nuell in geschützten Bereichen der Seiten implementiert werden, durch die Standardisierung von Viewtypen kann aber auch die Entwicklung der Präsentationsschicht durch die modellgetriebene Entwicklung unterstützt werden. Im Folgenden werden die ViewTypen *CreateView* und *IndexView* erläutert. Die ausgelassenen restlichen ViewTypen sind strukturell ähnlich und können im Anhang eingesehen werden.

6.7.1 Template für CreateView

Die *CreateView* dient der Generierung von Eingabeformularen. Der View ist indirekt eine Contentklasse zugeordnet, da die View mit dem Tagged Value *controller* einen Controller referenziert, der über seine Geschäftsklasse eine Contentklasse besitzt. Für die CreateView wird durch das Template in Listing 46 ein JSF-Formular generiert, das sämtliche Attribute der Contentklasse geordnet nach der Reihenfolge der Attribute in der Klassendefinition auflistet. Die Werte der Input-Elemente werden so generiert, dass sie auf die im Controller enthaltene Contentklasse in der Form *#{personController.person.name}* verweisen. In der Aktion des Buttons zum Absenden des Formulars muss auf einen Outcome in der faces-config.xml verwiesen werden. In diesem Fall wird statisch auf die erste ausgehende Kante bzw. den durch sie repräsentierten Outcome verwiesen. Der ActionListener referenziert die mit dem Klickereignis verbundene Controllermethode in der Form *#{personController.savePerson}*, das bei einem Klick auf den Button ausgelöst wird und die Speicherung des Datensatzes durch den Controller initiiert.

```
1  «DEFINE Root(control::Node node) FOR view::CreateView»
2    <h:form>
3      <h:panelGrid columns="2" border="1">
4      «FOREACH getContentClass(this).ownedAttribute AS attribute»
5        <h:outputText value="«attribute.name»:" />
6        «IF isPrimitive(attribute.type)»
7          <h:inputText id="«attribute.name»" value="#{«getManagedBeanName(getController
                     (this)) + "." + getManagedBeanName(getContentClass(this)) + "." +
                     getManagedAttributeName(attribute)»}" />
8        ...
9        «ENDIF»
10     «ENDFOREACH»
11     </h:panelGrid>
12     <h:commandButton action="«node.outgoing.first().name»" actionListener="#{
                 «getManagedBeanName(getController(this))».«getManagedOperationName(
                 getOperationCSave(getController(this)))»}" value="Create"/>
13   </h:form>
14 «ENDDEFINE»
```

Listing 46: Template für CreateView

Bei der Generierung von Eingabefeldern von Attributen wird danach unterschieden, ob der Typ des Attributs ein Basisdatentyp oder ein komplexer Typ ist. Attribu-

te mit Basisdatentypen können in stringbasierten Inputfeldern ausgegeben werden und werden bei der Speicherung in einer Managed Bean automatisch von JSF bzgl. ihres Typs konvertiert. [10] Eingabefelder für Attribute mit komplexen Datentypen dagegen werden wie in Listing 47 dargestellt als Drop-Down-Listboxen zugänglich gemacht. Beispielsweise kann bei Büchern der Autor ausgewählt werden, indem vom Buchcontroller die aktuell ausgewählte Person geladen wird und vom PersonenController sämtliche Personendatensätze angefordert werden, um die Listbox zu befüllen. Der Datensatz der selektierten Person wird zur Präsentation nicht direkt aus der Contentklasse ausgelesen, sondern über den Controller, weil der Datensatz vorab in ein SelectItem transformiert werden muss, was durch den Controller durchgeführt wird.

```
1 <h:outputText value="author:"/>
2 <ice:selectOneMenu id="author" value="#{bookController.authorSelectedAsSelectItem}">
3   <f:selectItems value="#{personController.personenAllAsSelectItems}"/>
4 </ice:selectOneMenu>
```

Listing 47: Listbox für die Selektion einer Person

Zu der Fallunterscheidung nach der Komplexität des Attributstypes wird anhand der Multiplizität des Attributes unterschieden, ob eine Listbox mit einfacher oder mehrfacher Auswahlmöglichkeit generiert werden soll. Ja nach Multiplizität müssen auch unterschiedliche SelectItem-Transformatormethoden aufgerufen werden.

```
1  «IF isPrimitive(attribute.type)»
2  ...
3  «ELSE»
4    «LET getManagedBeanName(getPrimaryController((content::ContentClass) attribute.type
           )) + "." + getManagedAttributeName(getPropertyCList(getPrimaryController
           ((content::ContentClass) attribute.type))) + "AllAsSelectItems" AS
           allValues»
5      «IF !isMultiple(attribute)»
6        <ice:selectOneMenu id="«attribute.name»" value="#{«getManagedBeanName(
                 getController(this)) + "." + getManagedAttributeName(attribute)
                 »SelectedAsSelectItem}">
7          <f:selectItems value="#{«allValues»}"/>
8        </ice:selectOneMenu>
9      «ELSE»
10       <ice:selectManyListbox id="«attribute.name»" value="#{«getManagedBeanName(
                 getController(this)) + "." + getManagedAttributeName(attribute)
                 »SelectedAsSelectItems}">
11         <f:selectItems value="#{«allValues»}"/>
12       </ice:selectManyListbox>
13     «ENDIF»
14   «ENDLET»
15 «ENDIF»
```

Listing 48: Template für CreateView

6.7.2 Template für IndexView

Mit dem Template für IndexViews in Listing 49 werden Tabellen generiert, die zeilenweise Datensätze von Contentklassen und spaltenweise deren Attribute darstellen. Die Tabelle wird über die CList-Operation des Controllers der Contentklasse mit Datensätzen befüllt. Neben der Generierung der Spalten für Attribute werden zudem Spalten für Links erzeugt. Dazu werden die ausgehenden Kanten des Knotens der View auf Knoten mit passenden Viewtypen durchsucht. Falls z. B. ein Knoten mit einer IndexView eine ausgehende Kante zu einem Knoten mit einer UpdateView besitzt, wird vom Template in der Liste ein Link mit der Beschriftung *modify* eingefügt. Als Ausnahme wird der Link zum Anlegen eines neuen Datensatzes nur einmalig unterhalb der Tabelle angelegt, ist also nicht Teil der *<ice:dataTable>*.

```
1  «DEFINE Root(control::Node node) FOR view::IndexView»
2    <h:form>
3      <ice:dataTable value="#{«getManagedBeanName(getController(this))».
               «getManagedAttributeName(getPropertyCList(getController(this)))»}" var
               ="record">
4      «FOREACH getContentClass(this).ownedAttribute AS attribute»
5        <ice:column>
6          <f:facet name="header">
7            <ice:outputText value="«attribute.name»"/>
8          </f:facet>
9          <ice:outputText value="#{record.«attribute.name»}"/>
10       </ice:column>
11     «ENDFOREACH»
12     «EXPAND LinkToRecordNode(this) FOREACH node.outgoingNodes().select(e|e.
               hasViewType(view::RecordView))»
13     «EXPAND LinkToUpdateNode(this) FOREACH node.outgoingNodes().select(e|e.
               hasViewType(view::UpdateView))»
14     «EXPAND LinkToDeleteNode(this) FOREACH node.outgoingNodes().select(e|e.
               hasViewType(view::DeleteView))»
15     </ice:dataTable>
16     «EXPAND LinkToCreateNode(this) FOREACH node.outgoingNodes().select(e|e.
               hasViewType(view::CreateView))»
17   </h:form>
18 «ENDDEFINE»
```

Listing 49: Template für IndexView

Bei der Generierung eines Links auf datensatzbezogene Seiten wie z. B. Seiten zum Modifizieren und Löschen von Datensätzen wird ein Parameter *id* übergeben, dessen Wert mit dem Wert der ID des Datensatzes der Tabellenzeile belegt wird. Der Parameter wird vom JSF-Controller an die CLoad-Methode des WAML-Controllers übergeben, die mit ihm den entsprechenden Datensatz zur Bearbeitung lädt. Zur Vereinfachung der Templates sollten Contentklassen deshalb nur eine ID besitzen.

```
1  «DEFINE LinkToUpdateNode(view::IndexView view) FOR control::Node»
2  ...
```

```
3    <h:commandLink action="«this.incoming.first().name»" actionListener="#{
              «getManagedBeanName(getController(view))».«getManagedOperationName(
              getOperationCLoad(getController(view)))»}">
4      <h:outputText value="modify"/>
5      <f:param name="id" value="#{record.id}"/>
6    </h:commandLink>
7  ...
8  «ENDDEFINE»
```

Listing 50: Template für parametrisierten Link in einer IndexView

6.8 Generierte Webapplikation

Die generierte Webapplikation ist ausschnittsweise für die Verwaltung von Personendaten in Abbildung 15 dargestellt. Sie ist unterteilt in die Pakete *content* und *control*. Die Klassen des Pakets *content* werden auf dem Applikationsserver durch einen EJB Container verwaltet, dagegen die Klassen des Pakets *control* und die durch Knoten repräsentierten Webseiten durch einen Web Container.

Die Entity Bean *Person* enthält Zugriffsmethoden für ihre Attribute und wird von der Session Bean *PersonManager* verwaltet. *PersonManager* implementiert das Interface *PersonManagerLocal*, in welchem die lokal auf dem Applikationsserver aufrufbaren Methoden deklariert sind. Die Methoden der Session Bean nutzen für Datenbankzugriffe den Entitymanager mit dem Namen *em*.

Der Controller *PersonController* ist im Vergleich zum Eingabemodell in Kapitel 6.1 um Methoden für den Zugriff auf die direkt assoziierte Entity Bean und für die Lokalisierung der Session Beans angereichert. Er wird durch die Webseiten benutzt, welche die Methoden des PersonControllers aufrufen. Exemplarisch ist die Verwendung der Methode *getPersonen* durch die Seite *personlist.jspx* modelliert.

Die Webseiten verweisen durch Links aufeinander, im Unterschied zum Navigationsmodell in Kapitel 6.1 sind aber zusätzlich die menübasierten Links als gestrichelte Kanten in das Diagramm eingezeichnet. Das mengenmäßige Verhältnis von Kanten menübasierter Links zu Kanten regulärer Links verdeutlicht, dass der Verzicht auf menübasierte Kanten im Navigationsmodell sinnvoll ist.

Die restlichen Klassen und Webseiten werden strukturgleich generiert und können in der Anlage eingesehen werden.

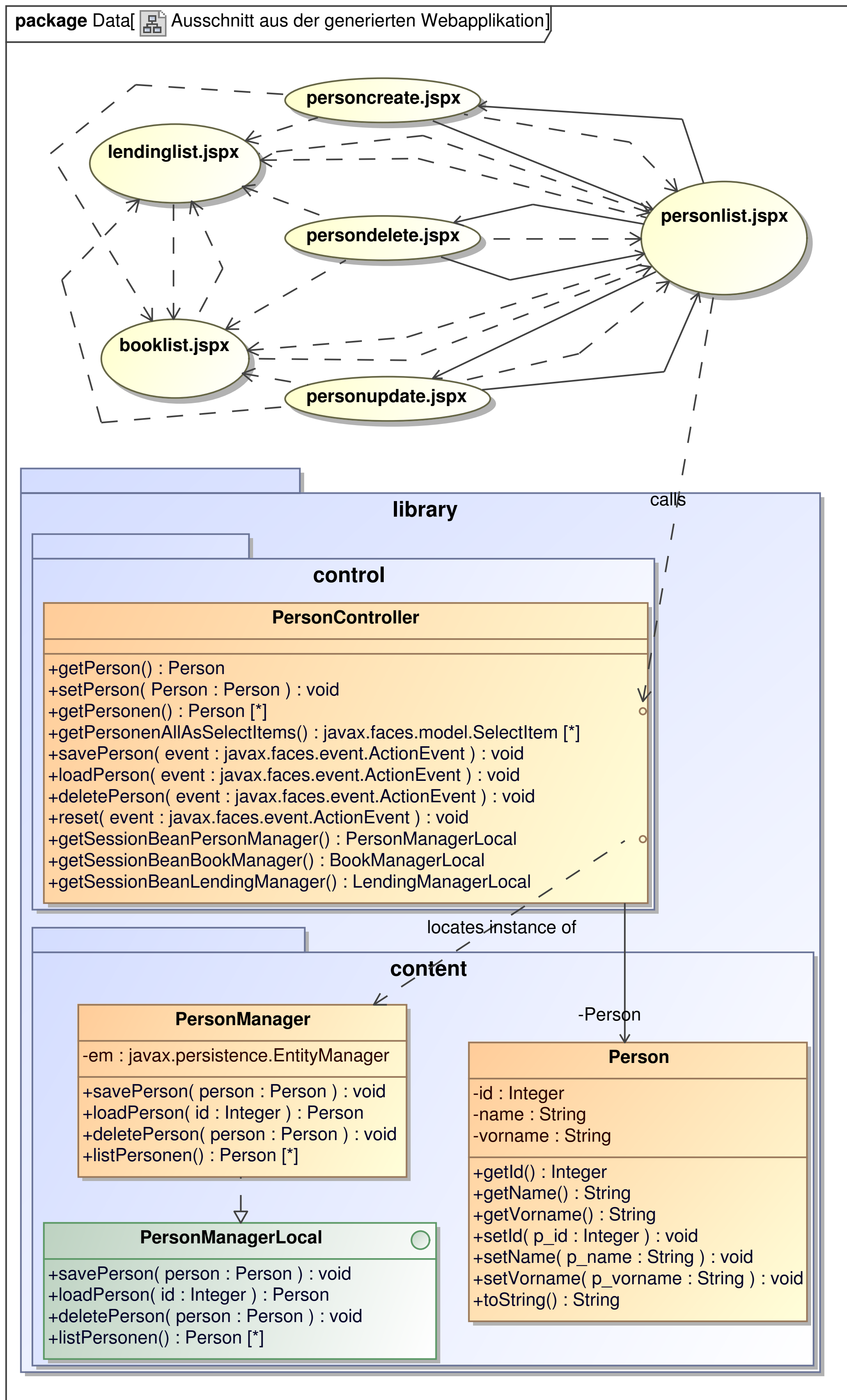


Abbildung 15: Ausschnitt aus der generierten Webapplikation

7 Bewertung und Ausblick

Im Rahmen dieser Arbeit wurde die Vision der modellgetriebenen Entwicklung mit dem status quo verglichen, indem die openArchitectureWare exemplarisch für die Generierung einer prototypischen Webapplikation verwendet wurde. Zur Modellierung von Webapplikationen wurde die Sprache WAML entwickelt, die sich an den Konzepten der vorgestellten Sprachen WAE, WebML und UWE orientiert. Der Explikationsgrad von WAML wurde so gewählt, dass die Modelle zwar hinreichend abstrakt sind, aber dennoch eine präzise Parametrisierung der Templates ermöglichen. Abschließend wurde gezeigt, dass das Ziel der vollautomatisierten Generierung für Standardfunktionalitäten erreicht werden kann.

Die Vision einiger Vertreter der MDA, dass die modellgetriebene Entwicklung die quelltextbasierten Entwicklungsmethoden ablösen wird, wird sich meiner Meinung nach mittel- bis langfristig nicht erfüllen. Sprichwörtlich sieht jedes Problem aus wie ein Nagel, wenn das einzige Werkzeug ein Hammer ist. Die gleiche Herangehensweise droht beim Einsatz des MDD, das meiner Meinung nach nicht optimal für die Entwicklung von kleinen Applikationen mit spezifischen Operationen ist. Es ist nicht sinnvoll bis unmöglich, spezifische Methodenrümpfe vollständig zu generieren. Falls der Schwerpunkt einer zu entwickelnden Software nicht auf objektorientierten Strukturen, sondern auf komplexen Methodenrümpfen oder Funktionen liegt, sind die Einsatzmöglichkeiten des MDD eingeschränkt, da auch die Templates sehr spezifisch sein müssten.

Die Komplexität einer Applikation wird mit der MDD nicht reduziert, sondern nur in Form von Modellen anders repräsentiert und gekapselt. In der objektorientierten Softwareentwicklung findet die Komplexitätsreduktion unter anderem durch Frameworks statt, welche Implementierungen wiederverwendbar und abstrakt anbieten. Ob die Komplexität sich hinter Modellelementen oder hinter Frameworkklassen verbirgt, ist für die Existenz und Problematik der Komplexität irrelevant.

Die betriebswirtschaftliche Effizienz des MDD ist sicherlich in den meisten Fällen kurzfristig nicht gegeben, weil parallel das Zielsystem sowie der Generator von geschulten Programmierern entwickelt werden müssen. Sobald aber mehrere Systeme oder Versionen eines Systems wiederholt durch einen Generator erzeugt werden, kann sich die Investition in den Generator qualitativ, zeitlich und hinsichtlich der Kosten rentieren.

Als Warnung sollen die prominentesten Probleme während der Entwicklung erwähnt werden. Beim XMI-Export von MagicDraw 14.0 werden Meldungen in einem Logging-Fenster ausgegeben. Mit zunehmender Textmenge in diesem Fenster wird

der Export schon nach wenigen Exportvorgängen bis zur Unbrauchbarkeit verlangsamt. Der in JSF verwendete Ansatz der Dependency Injection hat zur Folge, dass die Konsistenz des Quelltextes nicht vor der Laufzeit überprüft wird. Dies hat aufwändige Suchen nach Fehler zur Folge. Bei häufigem Deployment wird zudem ein Speicherleck im JBoss-Server sichtbar, das einen regelmäßigen Neustart des Servers erzwingt.

Offene Aspekte für die Weiterentwicklungen des prototypischen Generators sind, den Generator um weitere Templates für Views anzureichern und eine verfeinerte Parametrisierung der Views zu ermöglichen. Insbesondere bei den Modifikationen von Datensätzen sollten die Views so verfeinert werden, dass sie vom Benutzer Reaktionen auf Fehlermeldungen bzgl. der Sicherung der referentiellen Integrität erfragen. Serverseitig sollten Transaktionen unterstützt werden, um Kollisionen bei der parallelisierten Verarbeitung von Daten kontrolliert zu behandeln. Zur Sicherstellung konsistenter Modelle können Constraints eingeführt werden, mit denen vor der Generierung Modellvalidierungen durchgeführt werden.

Literatur

[1] http://java.sun.com/javase/6/docs/api/java/io/Serializable.html

[2] *Eclipse Public License - v 1.0.* http://www.eclipse.org/legal/epl-v10.html

[3] *Ecore.* http://download.eclipse.org/modeling/emf/emf/javadoc/2.4.0/org/eclipse/emf/ecore/package-summary.html

[4] *Fornax-Plattform.* http://www.fornax-platform.org/

[5] *Hypertext Transfer Protocol – HTTP/1.1.* http://tools.ietf.org/html/rfc2616

[6] *The Open Source Definition.* http://www.opensource.org/docs/osd

[7] *UML Metamodel.* http://download.eclipse.org/modeling/mdt/uml2/javadoc/2.2.0/

[8] *Unified Modeling Language (UML), version 2.1.1.* 2007

[9] Balzert, Helmut: *Lehrbuch der Software-Technik.* Bd. 1. Software Entwicklung. 2. Aufl. Spektrum Akademischer Verlag, 2000

[10] Bergsten, Hans: *JavaServer faces.* Sebastopol, CA : O'Reilly, 2004 http://www.loc.gov/catdir/enhancements/fy0715/2004301572-d.html. – ISBN 0596005393

[11] Bongio, Aldo ; Ceri, Stefano ; Fraternali, Piero ; Maurino, Andrea: Modeling data entry and operations in WebML. In: *Lecture Notes in Computer Science* 1997 (2000), S. 201 – 214

[12] Budinsky, Frank: *Eclipse modeling framework : a developer's guide.* Boston, MA : Addison-Wesley, 2003. – ISBN 0131425420

[13] Cardelli, Luca ; Wegner, Peter: On Understanding Types, Data Abstraction, and Polymorphism. In: *ACM Computing Surveys* 17 (1985), 12, Nr. 4, S. 471–522

[14] Ceri, Stefano: *Designing data-intensive Web applications.* Amsterdam : Morgan Kaufmann Publishers, 2003 http://www.loc.gov/catdir/description/els031/2002114096.html. – ISBN 1558608435

[15] Ceri, Stefano ; Fraternali, Piero ; Bongio, Aldo: Web Modeling Language (WebML): a modeling language for designing Web sites. In: *ACM Journal* 33 (2000), Nr. 1-6, S. 137–157

[16] Conallen, Jim: Modeling Web Application Architectures with UML / Rational Software. 1999. – Forschungsbericht

[17] Conallen, Jim: *Building Web applications with UML.* 2nd ed. Boston : Addison-Wesley, 2003. – ISBN 0201730383 (pbk. : alk. paper)

[18] Dijkstra, Edsger W.: The humble programmer. In: *Commun. ACM* 15 (1972), Nr. 10, S. 859–866. http://dx.doi.org/http://doi.acm.org/10.1145/355604.361591. – DOI http://doi.acm.org/10.1145/355604.361591. – ISSN 0001–0782

[19] Dumke, Reier ; Lother, Mathias ; Wille, Cornelies ; Zborg, Fritz: *Web Engineering.* Addison Wesley, 2003

[20] Efftinge, Sven: Was genau ist eigentlich eine DSL? In: *Javamagazin* (2008), Nr. 1, S. 87 – 90

[21] Efftinge, Sven ; Friese, Peter ; Haase, Arno ; Kadura, Clemens ; Kolb, Bernd ; Moroff, Dieter ; Thoms, Karsten ; Völter, Markus: *openArchitectureWare User Guide, Version 4.2*, 09 2007

[22] Fowler, Martin: *UML distilled : a brief guide to the standard object modeling language.* 3rd ed. Boston : Addison-Wesley, 2004. – ISBN 0321193687 (pbk. : alk. paper)

[23] Frankel, David: *Model driven architecture : applying MDA to enterprise computing.* New York : Wiley, 2003 http://www.loc.gov/catdir/bios/wiley045/2002014914.html. – ISBN 0471319201 (PAPER : alk. paper)

[24] Geary, David M. ; Horstmann, Cay S.: *Core JavaServer faces.* 2nd ed. Upper Saddle River, NJ : Prentice Hall, 2007 http://www.loc.gov/catdir/toc/ecip0711/2007006830.html. – ISBN 9780131738867 (pbk. : alk. paper)

[25] Greiffenberg, Steffen: Methoden als Theorien der Wirtschaftsinformatik. In: *Wirtschaftsinformatik 2003* 2 (2003), S. 947–967

[26] Hennicker, Rolf ; Koch, Nora: Modeling the User Interface of Web Applications with UML / Ludwig-Maximilians-Universität München. 2001. – Forschungsbericht

[27] Hutton, Graham: *Programming in Haskell.* Cambridge, UK : Cambridge University Press, 2007 `http://www.loc.gov/catdir/enhancements/fy0729/2007274987-b.html`. – ISBN 9780521871723 (hbk.)

[28] Kappel, Gerti (Hrsg.) ; Pröll, Birgit (Hrsg.) ; Reich, Siegfried (Hrsg.) ; Retschitzegger, Werner (Hrsg.): *Web Engineering: Systematische Entwicklung von Web-Anwendungen.* dpunkt, 2003. – ISBN 3–89864–234–8

[29] Kleppe, Anneke G. ; Warmer, Jos B. ; Bast, Wim: *MDA explained : the model driven architecture : practice and promise.* Boston : Addison-Wesley, 2003. – ISBN 032119442X (alk. paper)

[30] Koch, Nora: *Adaptive Hypermedia Systems Adaptive Hypermedia Systems Software Engineering for Adaptive Hypermedia Systems*, Ludwig-Maximilian-Universität München, Diss., 2001

[31] Koch, Nora: *The UWE Metamodel.* 2007

[32] Koch, Nora ; Kraus, Andreas: The expressive Power of UML-based Web Engineering / Institut für Informatik, Ludwig-Maximilians-Universität München. 2002. – Forschungsbericht

[33] Kraus, Andreas ; Koch, Nora: A Metamodel for UWE / Institut für Informatik, Ludwig-Maximilians-Universität München. 2003. – Forschungsbericht

[34] Kühn, Harald: Ein Vergleich von Modellierungssprachen für Web-Anwendungen / Universität Wien, Institut für Informatik und Wirtschaftsinformatik. 2001. – Forschungsbericht

[35] Michaelson, Greg: *An introduction to functional programming through Lambda calculus.* Wokingham, England : Addison-Wesley Pub. Co., 1989. – ISBN 0201178125

[36] Miller, Joaquin ; Mukeriji, Jishnu: MDA Guide Version 1.0.1 / OMG. 2003 (1.0.1). – Forschungsbericht

[37] Moreno, Nathalie ; Fraternali, Piero ; Vallecillo, Antonio: A UML 2.0 Profile for WebML Modeling. In: *ACM Journal* 155 (2006), Nr. 4

[38] Moreno, Nathalie ; Fraternali, Piero ; Vallecillo, Antonio: WebML Modeling in UML. In: *IET Software* 1 (2007), Nr. 3, S. 67–80

[39] Murugesan, San ; Deshpande, Yogesh ; Hansen, Steve ; Ginige, Athula: Web Engineering: A New Discipline for Development of Web-Based Systems. In: *Lecture Notes in Computer Science* 2016 (2001), S. 3–13

[40] Naur, P. ; Randell, B.: *Software Engineering.* NATO, Scientific Affairs Division, Brussels, 1969

[41] Panda, Debu ; Rahman, Reza ; Lane, Derek: *EJB 3 in action.* Greenwich, CT : Manning Publications Co., 2007 `http://www.manning.com/EJB3inAction`. – ISBN 9781933988344 (pbk.)

[42] Pietreck, Georg ; Trompeter, Jens ; Niehues, Benedikt ; Kamann, Thorsten ; Holzer, Boris ; Kloss, Michael ; Thoms, Karsten ; Beltran, Juan Carlos F. ; Mork, Steffen: *Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis.* entwickler.press, 2007

[43] Pilone, Dan ; Pitman, Neil: *UML 2.0 in a nutshell.* 1st ed. Beijing : O'Reilly Media, 2005 `http://www.loc.gov/catdir/toc/fy0608/2006530121.html`. – ISBN 0596007957 (pbk.)

[44] Rossbach, Peter ; Stahl, Thomas ; Neuhaus, Wolfgang: Model Driven Architecture. In: *Javamagazin* 9 (2003)

[45] Stahl, Thomas ; Völter, Markus: *Model-driven software development : technology, engineering, management.* Chichester, England : John Wiley, 2006 `http://www.loc.gov/catdir/toc/ecip069/2006007375.html`. – ISBN 0470025700 (pbk. : alk. paper)

[46] Strobel, Claus: *Web-Technologien in E-Commerce-Systemen.* Oldenbourg Verlag, 2003

[47] Ullenboom, Christian: *Java ist auch eine Insel.* 6., aktualisierte und erweiterte Auflage. Bonn : Galileo Computing, 2007 `http://www.galileocomputing.de/openbook/javainsel6/`

[48] Völter, Markus: openArchitectureWare 4.2 Fact Sheet / eclipse. Version: 09 2007. `http://www.eclipse.org/gmt/oaw/`. 2007. – Forschungsbericht

Ich versichere hiermit, dass ich meine Diplomhausarbeit „*Modellgetriebene Entwicklung von Webapplikationen*“ selbstständig und ohne fremde Hilfe angefertigt habe, und dass ich alle von anderen Autoren wörtlich übernommenen Stellen wie auch die sich an die Gedankengänge anderer Autoren eng anlehnenden Ausführungen meiner Arbeit besonders gekennzeichnet und die Quellen zitiert habe.

Münster, den 30.01.2008

Ulrich Wolffgang