Westfälische Wilhelms-Universität Münster

Modellgetriebene Entwicklung daten- und prozessbasierter Webapplikationen

Inauguraldissertation

zur Erlangung des akademischen Grades
eines Doktors der Wirtschaftswissenschaften
durch die Wirtschaftswissenschaftliche Fakultät
der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Ulrich Wolffgang
aus Münster

Münster, Mai 2012

Dekan: Prof. Dr. Thomas Apolte

Erster Gutachter: Prof. Dr. Herbert Kuchen

Zweiter Gutachter: Prof. Dr. Jörg Becker, Professor h.c. (NRU - HSE, Moskau)

Tag der mündlichen Prüfung: 15. Mai 2012

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
ARIS	Architektur integrierter Informationssysteme
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CASE	Computer-Aided Software Engineering
CIM	Computation Independent Model
CMOF	Complete MOF
CRUD	Create, Read, Update, Delete
DAO	Data Access Object
DSL	Domain Specific Language
DTD	Document Type Definition
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
ERM	Entity-Relationship-Modell
GMF	Graphical Modeling Framework
GPML	General Purpose Modeling Language
GPPL	General Purpose Programming Language
HUTN	Human-Usable Textual Notation
JPA	Java Persistence API
JSF	JavaServer Faces
HTTP	Hypertext Transfer Protocol
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
MDWE	Model Driven Web Engineering
MOF	Meta Object Facility
MOFM2T	MOF Model to Text Transformation Language
OCL	Object Constraint Language
oAW	openArchitectureWare
OMG	Object Management Group
OO-H	Object-oriented Hypermedia Method
OOWS	Object Oriented Web Solution
PIM	Platform Independent Model
PSM	Platform Specific Model

POJO	Plain Old Java Object
QVT	Query/View/Transformation
QVTC	QVT Core
QVTO	QVT Operational Mappings
QVTR	QVT Relations
RIA	Rich Internet Application
SoC	Separation of Concerns
TaskUI	Task User Interface
UML	Unified Modeling Language
UWE	UML-based Web Engineering
SQL	Structured Query Language
XMI	XML Metadata Interchange
WAE	Web Application Extension for UML
WASL	Web Application Specification Language
WAR	Web Application Archive
WebML	Web Modeling Language
WWW	World Wide Web
XSD	XML Schema Definition
XSLT	XSL Transformation
YAML	YAML Ain't Markup Language

Zitationsstil der Arbeit

- Eine Literaturangabe wird durch eckige Klammern [...] gekennzeichnet. Metainformationen zur Literaturangabe werden umgebend in runde Klammern (...[...]) eingefasst.

- Ein direktes Zitat wird durch eine Literaturangabe i.V.m. Anführungszeichen „..." gekennzeichnet.

- Ein indirektes Zitat wird durch eine Literaturangabe ohne Verwendung von Anführungszeichen gekennzeichnet. Es gibt sinngemäß die Aussage der zitierten Quelle mit eigenen Worten wieder. Die Textstelle wird nicht im Konjunktiv formuliert. Der Literaturangabe wird kein *vergleiche* (vgl.) vorangestellt. Der Umfang eines indirekten Zitates wird folgendermaßen gekennzeichnet:

    1. Eine Literaturangabe vor einem Punkt in Verwendung als Satzzeichen kennzeichnet den gesamten Satz als indirektes Zitat der Quelle.
    2. Eine Literaturangabe innerhalb eines Satzes kennzeichnet den Satzabschnitt als indirektes Zitat der Quelle.
    3. Eine Literaturangabe am Ende eines Satzes, der auf einen Doppelpunkt endend eine Aufzählung einleitet, kennzeichnet sowohl den Satz als auch die Aufzählung als indirektes Zitat der Quelle.
    4. Eine Literaturangabe hinter dem letzten Punkt eines Absatzes kennzeichnet den gesamten Absatz als indirektes Zitat der Quelle. Falls ein Satz oder Satzabschnitt innerhalb dieses Absatzes durch eine weitere Literaturangabe gekennzeichnet ist, wird in diesem Bereich sinngemäß auf beide Quellen verwiesen.
    5. Falls in einem Absatz auf einen Satz mit einem indirekten Zitat nach Art 1 ein Satz folgt, der durch einen weiterführenden Term wie z. B. *hierbei*, *dabei* oder *hiermit* eingeleitet wird, ist er als Teil des indirekten Zitats des vorhergehenden Satzes zu werten.

- Ein Textabschnitt, welcher durch eine Literaturangabe mit vorangestelltem *vergleiche* (vgl.) annotiert ist, gibt die Literaturstelle nicht zwingend sinngemäß wieder, sondern verweist lediglich auf diese zur weitergehenden Lektüre. Inhaltliche Übereinstimmungen liegen zwar in der Natur des Verweises, der Textabschnitt ist aber eine originäre Aussage des Autors.

1 Einführung

1.1 Motivation und Zielsetzung der Arbeit

Die Entwicklung betrieblicher Anwendungssysteme ist trotz deren Relevanz für das Funktionieren moderner Unternehmen nach wie vor mit Unwägbarkeiten verbunden. So wurden auch in den letzten Jahren für IT-Projekte aus verschiedenen Branchen und Ländern signifikante Abbruch- und Misserfolgsraten [EK08, S. 88][SGR07] ermittelt, die auf relevante Zielabweichungen bei ungefähr einem Viertel bis Drittel der Projekte hindeuten. Auch weisen die resultierenden Softwaresysteme qualitative Mängel, z. B. hinsichtlich der Sicherheit auf. So sind SQL-Injections und Cross-Site Scripting (XSS) nach wie vor dominierende Angriffstechniken [MPM10, S. 14][10.10b, 11.11a], die auf unzureichende Überprüfungen clientseitiger Dateneingaben zurückzuführen sind. Die zugrunde liegenden Sicherheitslücken sind aufgrund ihrer vergleichsweise geringen Komplexität als Resultat ungenügender Programmierpraktiken zu werten und entfalten i.V.m. dem Trend zu global zugänglichen Webapplikationen eine besondere Brisanz. Als Resultat spiegeln sich in der Wirtschaft wiederholt bei einem hohen Prozentsatz der Unternehmen die verschiedenartigen Sicherheitsdefizite in Kompromittierungen der IT-Systeme wider [10.10a, S. 10][Ric11].

Die Problematik scheiternder Softwareentwicklungsprojekte und qualitativ unzureichender Softwaresysteme ist bereits seit den 1960er Jahren unter dem Stichwort der Softwarekrise dokumentiert [NR69, BR70] und führte in Form des Software Engineerings zu einem Bedarf an einem systematischen Vorgehen, erprobten Techniken und generell Abstraktionsmechanismen. Ein prägendes Merkmal der Methodik des Software Engineerings ist die Erstellung und Nutzung von Modellen, die als Artefakte ähnlich zu den Ingenieurwissenschaften primär Planungs- und Dokumentationszwecken dienen.

Die modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) strebt als neues Paradigma die (teil-)automatisierte Generierung von Softwaresystemen aus Modellen anstelle der Entwicklung von Quelltext an. Damit verlagert sich der Fokus der Softwareentwicklung von der Verfassung von Quelltext zu der Erstellung von Modellen und Generatoren. Mit dem erhöhten Automatisierungs- und Abstraktionsgrad wird auf Produktivitäts- und Qualitätssteigerungen abgezielt, indem fachliche Modelle mit Domänenbezug auf technische Implementierungsmuster abgebildet werden. Die Model Driven Architecture (MDA) der Object Management Group (OMG) standardisiert das MDSD und vereinheitlicht die Unterteilung in plattformunabhängige und plattformspezifische Modelle bzw. Perspektiven [Obj03].

Insbesondere im Kontext der Entwicklung Web-basierter Anwendungssysteme bietet die Trennung des Fachkonzepts bzw. der plattformunabhängigen Systemspezifikation von

der Implementierung Vorteile, da Web-Technologien mit einer vergleichsweise hohen Innovationsgeschwindigkeit fortentwickelt werden und entsprechend Änderungen für die technischen Realisierungen induzieren. So wurden innerhalb der letzten Dekade die anfänglichen Techniken zur serverseitigen Generierung dynamischer und personalisierter Webseiten zu solchen für komplexe Webapplikationen weiterentwickelt. Letztere lassen sich beispielsweise über Middleware-Techniken wie Java EE in bestehende IT-Landschaften integrieren und können mittels AJAX (vgl. [Gar05]) ähnlich zu klassischen Desktopanwendungen als Rich Internet Applications (RIAs) mit interaktiven Benutzeroberflächen gestaltet werden.

Im Kontext des Web Engineering als Unterdisziplin des Software Engineering existieren einige modellgetriebene Lösungsansätze für die Spezifikation und Generierung von Webapplikationen. Diesen ist gemein, dass die verwendeten Modellierungssprachen einen tendenziell fachkonzeptionellen Charakter aufweisen und die detaillierte Repräsentation von Implementierungsdetails im Modell nicht im Fokus steht. Damit ist verbunden, dass bislang kein Generatorframework existiert, welches multiple selbstständige Codegeneratoren für die unterschiedlichen Webplattformen bzw. deren Programmiersprachen und Frameworks integriert.

Das Ziel der Arbeit ist die Konstruktion und Evaluation von Generatorframeworks für die modellgetriebene Entwicklung daten- und prozessbasierter betrieblicher Webapplikationen. Die intendierte Funktionalität der Generatorframeworks zielt sowohl auf die automatisierte Erzeugung von Webapplikationen ab, deren Geschäftslogik rein Datenhaltungs-orientiert ist, als auch auf solche, die eine Interaktion mit Geschäftsprozessen ermöglichen. Das Paradigma der modellgetriebenen Softwareentwicklung wird verfolgt, da betriebliche Anwendungssysteme üblicherweise im Kern Geschäftslogik zur Verwaltung von Daten aufweisen. Diese lässt sich mit weniger Aufwand generieren, als vergleichsweise komplexere algorithmische Teile der Geschäftslogik, die als manuell erzeugte Komponenten in das Generat integriert werden können. Das Gestaltungsvorhaben zur Konstruktion der Generatorframeworks ist insofern restringiert, als dass erstens multiple praxisrelevante Webplattformen mit verschiedenen Programmiersprachen in Form von Java EE, Python und PHP zu adressieren sind. Zweitens sind Techniken zu verwenden, die konform zu den Standards der MDA sind, um deren vereinheitlichende Wirkung zu nutzen. Drittens sind die Webapplikationen sowohl plattformunabhängig als auch plattformspezifisch mit Bezug auf möglichst viele Implementierungsdetails zu modellieren (vgl. Kapitel 4.2). Auf diese Weise soll die Unterstützung einer möglichst großen Bandbreite an Typen von Anwendungen und im Speziellen Webapplikationen sichergestellt werden.

1.2 Struktur der Arbeit

Die Arbeit ist untergliedert in sechs Hauptkapitel (vgl. Abbildung 1). Den Ausgangspunkt bilden in Kapitel 2 konzeptionelle und technische Grundlegungen zur modellgetriebenen Softwareentwicklung als Technik zur Generierung von Softwaresystemen. Sie wird mit ihren Charakteristika zunächst vor dem Hintergrund ihres Ursprungs im Software Engineering eingeordnet und vom Computer-Aided Software Engineering abgegrenzt. Ein grundlegendes Merkmal modellgetriebener Softwareentwicklung ist die Definition von Modellierungssprachen mittels Metamodellierungstechniken, auf die zunächst konzeptionell und anschließend in technischer Hinsicht im Kontext der MDA als standardisierendem Ansatz eingegangen wird. Das Eclipse Modeling Framework (EMF) implementiert Konzepte der MDA und wird auszugsweise erläutert. Abschließend wird auf Transformationstechniken zur Generierung von Quelltext und Modellen aus Modellen eingegangen.

In Kapitel 3 werden modellgetriebene Ansätze aus dem Kontext des Web Engineering zur Entwicklung von Webapplikationen vorgestellt. Nach einer detaillierten Beschreibung ihrer Spezifika werden diese vergleichend in einen Klassifikationsrahmen eingeordnet und bzgl. ihrer Konformität zu den Prinzipien und Techniken der MDA evaluiert. Das Ergebnis motiviert die Entwicklung des WASL-Generatorframeworks, das ebenfalls klassifiziert wird.

Auf das WASL-Generatorframework zur modellgetriebenen Entwicklung datenbasierter Webapplikationen wird im vierten Kapitel im Detail eingegangen. Ausgehend von den zu erfüllenden Anforderungen wird ein Überblick über die Schichtenarchitektur des Frameworks gegeben. Das Vorgehen zur Entwicklung der Framework-Komponenten wird in der gestaltungsorientierten Wissenschaft verortet und hinsichtlich der Gestaltungsfreiheiten durch Designprinzipien restringiert. Die resultierenden Metamodelle und Transformationsdefinitionen sind der Kern des Generatorframeworks und werden abschließend detailliert erläutert.

In Kapitel 5 wird der Templategenerator *Metagen* zur Generierung von Modell-zu-Text-Transformationsdefinitionen vorgestellt, mit dessen Hilfe einige der Transformationsdefinitionen des WASL-Generatorframeworks automatisiert erzeugt werden.

Im sechsten Kapitel wird auf das TaskUI-Generatorframework eingegangen, welches der Entwicklung prozessbasierter Webapplikationen dient, die in Form von formularbasierten Web-Benutzerschnittstellen für BPMN-Prozessmodelle implementiert werden. Dazu werden vorab relevante Spezifika der Modellierungssprache BPMN erläutert. Aus der BPMN wird das TaskUI-Generatorframework abgeleitet, das auf Komponenten des WASL-Generatorframeworks verankert ist und hinsichtlich der Architektur, Metamodelle und Transformationsdefinitionen beschrieben wird.

Die Ergebnisse der Arbeit werden in Kapitel 7 einer Evaluation unterzogen. Diese be-

fasst sich mit den Eigenschaften und Limitationen der Generatorframeworks, sowie deren Auswirkung auf die Softwareentwicklungsmethode im Vergleich zu der rein quelltextorientierten Softwareentwicklung.

Die Arbeit schließt in Kapitel 8 mit einer Zusammenfassung der Ergebnisse und einem Ausblick auf weiteren Forschungsbedarf.

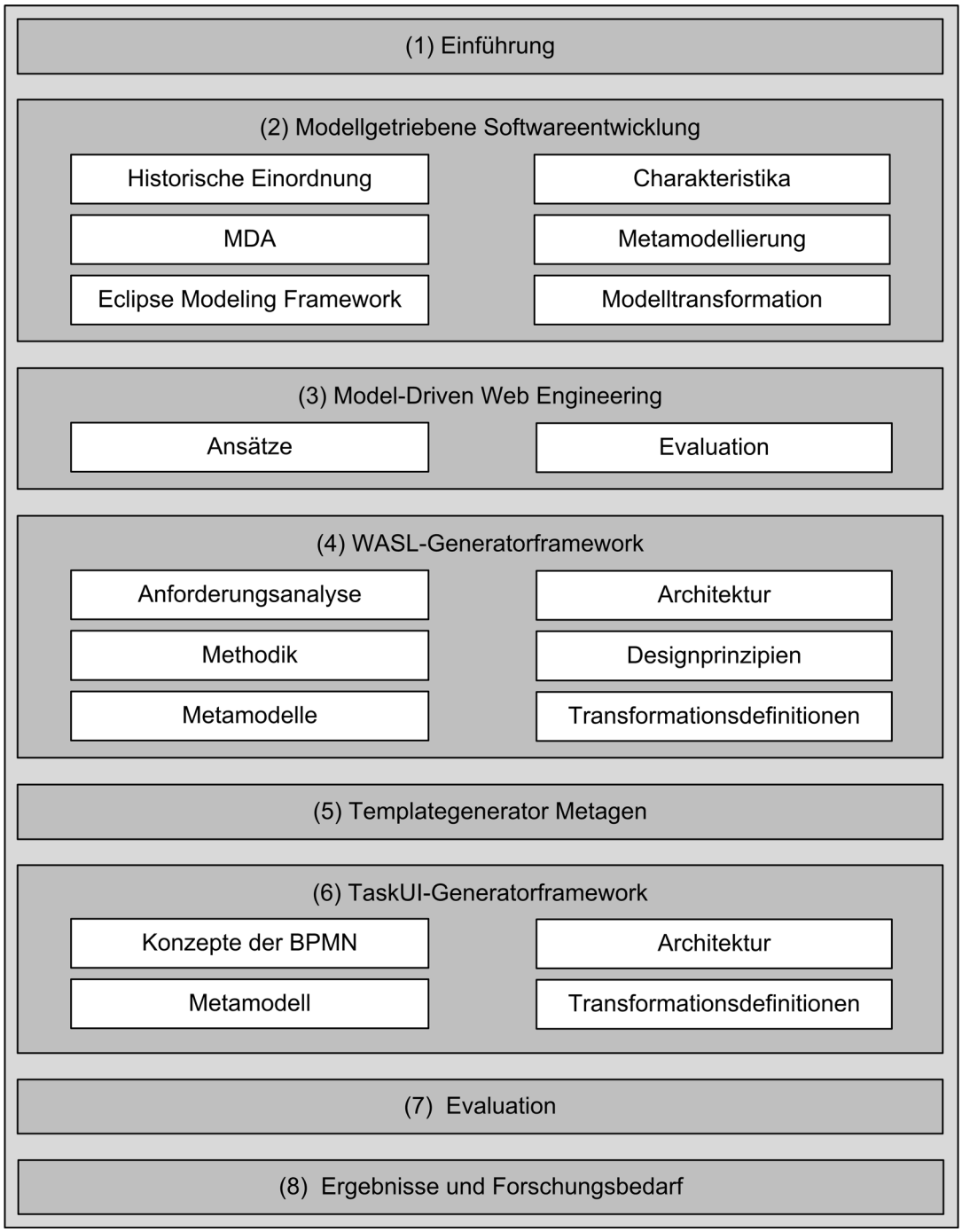

Abbildung 1: Struktur der Arbeit

2 Modellgetriebene Softwareentwicklung

Die in dieser Arbeit vorgestellten Generatorframeworks zur Entwicklung betrieblicher Webapplikationen basieren auf Prinzipien und Techniken der Model Driven Architecture (MDA), die ihrerseits ein präzisierender und standardisierender Ansatz für die modellgetriebene Softwareentwicklung ist. Daher wird als Grundlegung der Arbeit zunächst auf die modellgetriebene Softwareentwicklung im Allgemeinen und anschließend auf die Model Driven Architecture im Speziellen mit ihren technischen Details eingegangen.

2.1 Begriffsbildung

Nach Trompeter et al. [TPB+07, S. 11] befasst sich die modellgetriebene Softwareentwicklung mit der „Automatisierung in der Softwareherstellung“, sodass „möglichst viele Artefakte eines Softwaresystems generativ aus formalen Modellen abgeleitet werden“. Stahl et al. [SVEH07, S. 11] definieren die modellgetriebene Softwareentwicklung als „... Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen“. Die Definition ist weiter gefasst, da zu diesen sowohl die reine Generierung von Code als auch die Interpretation von Modellen zur Laufzeit gezählt werden [SVEH07, S. 12].

Die vielzähligen verschiedenen Ansätze für das MDSD weisen einen hohen Grad an Heterogenität bzgl. der konkreten Realisierungen auf, sodass eine Festlegung auf bestimmte Modellierungssprachen, Standards, Entwicklungswerkzeuge oder anderweitige spezifische technische Details nicht durchgängig möglich ist. Stattdessen ist der Zweck bzw. die Zielsetzung und die Inanspruchnahme von Modellen als zentrale Artefakte für dessen Erreichung hervorzuheben.

Im Folgenden wird die modellgetriebene Softwareentwicklung deshalb als eine Paradigma zur Entwicklung von Softwaresystemen verstanden, welches die Generierung oder den Betrieb von Softwaresystemen auf Basis von Modellen bzw. Diagrammen vorsieht. Modelle werden über die reinen Analyse-, Entwurfs- und Dokumentationsfunktionen hinaus als zentrale Artefakte durchgängig im Fertigungsprozess verwendet. Das Paradigma zielt auf die Vermeidung von Routinearbeiten und die Vermeidung einer Divergenz zwischen Modellen und deren Implementierungen ab. Im Fall der Generierung von Code kommt die Absicht einer Steigerung der Codequalität hinzu. Zur Modellierung können sowohl generische als auch domänenspezifische Modellierungssprachen verwendet werden, welche optional adaptiert oder zur Gänze neu erstellt werden.

2.2 Historische Einordnung

Das MDSD steht als Softwareentwicklungsmethode in der Tradition des Software Engineering, in dessen Rahmen schon vor dem MDSD modellbasierte Ansätze der Softwareentwicklung Gegenstand der Forschung und Praxis waren (vgl. [Boe06]). Da das MDSD von diesen vorgelagerten Herangehensweisen beeinflusst ist, ist für eine Einordnung des MDSD deren Betrachtung sinnvoll.

Das Software Engineering bildete sich als Reaktion auf die Softwarekrise Ende der 1960er Jahre heraus [NR69, BR70]. Gefordert wurde und wird für die Softwareentwicklung ein ingenieurmäßiges Vorgehen, das basierend auf theoretischen Fundierungen die reproduzierbare Entwicklung von Software anstelle von künstlerisch-kreativen Ad-hoc Schaffungsprozessen ermöglichen soll. Der Bedarf nach entsprechenden Methoden resultiert aus dem Phänomen, dass mit der zunehmenden Mächtigkeit der Computersysteme Erwartungen an neue Einsatzgebiete verbunden sind, welche zu steigenden Anforderungen an immer komplexere und gesellschaftlich relevantere Systeme führen [Dij72].

Erste Ansätze zur Verwendung von Modellen fanden sich ab den 1970er Jahren in strukturierten Entwicklungsmethoden wie der strukturierten Analyse (SA) [DeM79] oder dem strukturierten Design (SD) [YC79] zur abstrahierenden Darstellung von Datenstrukturen und -flüssen [GPR06, S. 12 f.]. Diese mündeten unter anderem in die Structured Analysis and Design Technique (SADT) [MM87], welche eine Diagramm-basierte Notation zur Erstellung von Aktivitäts- und Datenmodellen bietet. Der Fokus liegt dabei auf den Analyse- und Entwurfsphasen, die Nutzung in der Implementierungsphase zur Generierung von Code aus Modellen ist nicht von vorrangiger Bedeutung.

In den 1980er Jahren kam mit dem Computer-Aided Software Engineering (CASE) die Leitidee hinzu, die Softwareentwicklungsprozesse bzw. deren Phasen durch Werkzeuge zu begleiten und Quelltexte zumindest teilweise zu generieren [PW07][Sch06, S. 25]. Dazu wurde die graphische Modellierung von Softwaresystemen angestrebt, um diese mit einem höheren Abstraktionsgrad und stärkeren fachlichen Bezug zu spezifizieren, als es mit reinem Quelltext möglich ist. Für diese Zwecke wurde für das CASE eine Vielzahl an Softwarewerkzeugen zur Methodenunterstützung und Automatisierung entwickelt [FPM10, S. 243][Fug93, S. 25]. Die Werkzeugunterstützung reicht von der reinen Erstellung von Diagrammen bis hin zur integrierten Unterstützung des kompletten Softwareentwicklungsprozesses. In der Praxis konnte sich CASE aber nicht durchsetzen, da die verwendeten Modellierungssprachen zu generisch sind, keine geeigneten Ziel- bzw. Middlewareplattformen für die benötigten Codegeneratoren existieren und die verfügbaren Werkzeuge in größeren Entwicklungsprojekten nicht ausreichend skalieren [Sch06] (vgl. Kapitel 2.3.2).

Aufgrund der Verbreitung des objektorientierten Programmierparadigmas (OOP) ab

den 1980er Jahren etablierten sich in den 1990er Jahren im Rahmen des CASE korrespondierende objektorientierte Modellierungssprachen und -methoden. Ideen der Object-modeling technique (OMT) [RBP+91], des Object-oriented Design (OOD) [Boo93] und des Object-oriented Software Engineering (OOSE) [Jac92] mündeten im Rahmen der *Rational Software Corporation* in die Unified Modeling Language [BR95, BRJ96], welche im Jahr 1997 von der Object Management Group (OMG) als Standard akzeptiert wurde [Rat97] und anschließend zu einer verbreiteten Modellierungssprache für die objektorientierte Softwareentwicklung avancierte.

2.3 Charakteristika

Als Weiterentwicklung des CASE rückten ab den 2000er Jahren mit dem Model Driven Software Development (MDSD) bzw. synonym dem Model Driven Engineering (MDE) [FR07] oder dem zeitweise von der OMG als Marke geschütztem [04.04a] Model Driven Development (MDD) Modelle als zentrale Artefakte in den Vordergrund. Im Vergleich zum CASE und dessen schwerpunktmäßigem Einsatz von Modellen für Analyse-, Entwurfs- und Dokumentationszwecke steht beim MDSD die automatisierte Generierung lauffähiger Softwaresysteme aus Modellen im Vordergrund [FR07, S. 6]. Auch sollen Codegeneratoren und Modellierungssprachen neu konstruiert und vergleichsweise flexibel angepasst werden können (vgl. Kapitel 2.3.2).

Zur Realisierung dieser Absichten werden im MDSD je nach Auslegung und praktischer Umsetzung zur Modellierung entweder generische oder domänenspezifische Modellierungssprachen verwendet. Generische Modellierungssprache (General Purpose Modeling Language, GPML) wie z. B. die Unified Modeling Language (UML) bieten sprachlich vordefinierte, aber weitestgehend generische Semantik, welche durch vorgegebene Notationsformen visualisiert wird. Abhängig von der Sprachspezifikation sind Adaptionen durch spracheigene Mechanismen wie z. B. bei der UML durch UML-Profile [Obj10c] vorgesehen. Dagegen wird eine domänenspezifische Modellierungssprache bzw. Domain Specific Language (DSL) von Grund auf für die Bedeutungszusammenhänge einer Domäne erstellt (vgl. Kapitel 2.3.1). Für das MDSD ist im Vergleich zum CASE insbesondere die Nutzung von DSLs charakteristisch (vgl. Kapitel 2.3.2).

Die Entwicklung von DSLs im Rahmen des MDSD geschieht durch die Definition sprachlicher Metamodelle (vgl. Kapitel 2.4). Dazu existiert eine Vielzahl an Metamodellierungsansätzen basierend auf diversen Metamodellierungssprachen (vgl. [KK02]), deren Verwendung charakteristisch für das MDSD ist und von denen die Meta Object Facility (MOF) und Ecore zwei etablierte Varianten sind.

Die Lücke zwischen der abstrakten Beschreibung der zu lösenden Probleme durch Mo-

delle und der lauffähigen Implementierung wird durch Modell-zu-Text-Transformationen (M2T) überbrückt. Im Zuge einer M2T-Transformation werden textuelle Artefakte generiert, welche nicht auf Quelltext beschränkt sind, sondern zudem Konfigurationsdateien, XML-Deployment-Descriptoren, Datenbankskripte, HTML-Seiten, Dokumentationen etc. sein können. Abhängig von der praktischen Umsetzung des MDSD werden Modelle nicht nur in (Quell-)Texte, sondern auch in weitere Modelle transformiert (M2M).

Eine *Transformation* ist in diesem Kontext ein Prozess, in dessen Rahmen ein Modell oder Text aus einem Modell generiert wird, und der durch eine *Transformationsdefinition* formal beschrieben wird (vgl. [KWB03, S. 73]). Die Transformationsdefinition besteht ihrerseits aus *Transformationsregeln*, die spezifizieren, auf welche Weise die Modellelemente im Detail transformiert werden sollen. Abhängig von der konkret eingesetzten MDSD-Technik wird eine Transformationsdefinition auch als Mapping [Obj11b] oder Template [Obj03, S. 3-2 ff.] bezeichnet und eine Transformationsregel als Mapping oder Relation [Obj11b]. Die Transformation ist also die Ausführung einer oder mehrerer Transformationsdefinitionen auf mindestens einem Modell, wobei die Bedeutung des Quellmodells erhalten bleiben soll. Eine Transformation wird von einer *Transformationsengine* durchgeführt, welche Teil eines *Transformationswerkzeuges* mit zusätzlicher unterstützender Bedienfunktionalität sein kann. Ein *Generator* besteht aus einer oder mehreren Transformationsdefinitionen zzgl. unterstützender Komponenten, z. B. zur Validierung von Modellen (vgl. [TPB+07, S. 120 ff.]). Die Ausgabe des Generators nach einer Transformation ist das *Generat* [SVEH07, S. 21]. Manuell erstellte oder automatisiert generierte Modelle sowie generierter Quelltext werden auch als *Artefakte* bezeichnet. Ein Generat besteht somit aus Artefakten, wird aber alternativ auch insgesamt als Artefakt bezeichnet.

Für die Implementierung bzw. Formulierung von M2T-Transformationsdefinitionen werden Techniken vorausgesetzt, als die sowohl generische Programmiersprachen (General Purpose Programming Language, GPPL) wie Java oder C#, als auch spezialisierte Transformationssprachen zum Einsatz kommen [10.10c, S. 24 ff.] (vgl. Kapitel 2.7). Bei der Formulierung von Transformationsdefinitionen mittels generischer Programmiersprachen ist der zu generierende Text auf eine Vielzahl von Schreib-Anweisungen verteilt, was zu einer mangelnden Übersichtlichkeit führt. Templatesprachen sind eine alternative Art von Transformationssprachen, die den zu generierenden Text in kohärenter Form darstellen und durch Inline-Ausdrücke anreichern.

Zur Formulierung von M2M-Transformationsdefinitionen finden analog generische Programmiersprachen oder spezielle Transformationssprachen Verwendung, welche als domänenspezifische Programmiersprachen der deklarativen, prozeduralen oder hybrid der deklarativ-prozeduralen Spezifikation von Modelltransformationen dienen [SK03, S. 5] (vgl.

Kapitel 2.8).

Eine lediglich teilautomatisierte Generierung von Quelltext kann auch beim MDSD nicht ausgeschlossen werden, sodass Mechanismen benötigt werden, die manuell hinzugefügte Implementierungsanteile vor einem Überschreiben durch den Generator schützen. Hierzu unterstützen einige der Templatesprachen die Definition geschützter Bereiche (Protected Regions), deren textuelle Inhalte nicht überschrieben bzw. nach jeder Transformation wiederhergestellt werden.

Aus der heterogenen MDSD-Landschaft sticht als eine spezielle MDSD-Initiative die MDA der OMG [Obj03] hervor, die seit dem Jahr 2000 [Sol00] vorangetrieben wird. Charakteristisch für die MDA ist die Nutzung standardisierter OMG-eigener Techniken wie z. B. der UML und der MOF [Obj08a], um Systeme auf verschiedenen Abstraktionsniveaus durch plattformunabhängige und plattformspezifische Modelle als zentrale Artefakte zu spezifizieren und möglichst automatisiert zu generieren (vgl. Kapitel 2.5).

2.3.1 Domänenspezifische Modellierung

Ein Charakteristikum des MDSD ist die domänenspezifische Modellierung. Diese ist definiert als „die abstrahierende Darstellung von Sachverhalten aus einem konkreten Problemraum unter Verwendung einer Modellierungssprache, welche auf den Problemraum zugeschnitten ist“ [TPB+07, S. 56]. Fowler definiert eine DSL allgemein als eine Programmiersprache von begrenzter Ausdrucksfähigkeit, welche sich auf eine bestimmte Domäne konzentriert[1] [Fow10, S. 27]. Domänenspezifische Sprachen sind allgemein keine Schöpfung des MDSD, sondern finden bereits seit den 1950er Jahren in anderen Teilgebieten der Informatik Anwendung [TPB+07, S. 57]. Beispiele für DSLs außerhalb des MDSD sind die Backus-Naur-Form, die Datenbanksprache SQL und die Programmiersprache R. Die ursprüngliche Nutzung von DSLs zur Programmierung wurde jedoch mit dem MDSD um die Verwendung zu Modellierungszwecken erweitert [TPB+07, S. 58] und die vormals textuelle Form der Modelle um graphische Notationen ergänzt. Entsprechend wird im Kontext des MDSD nach Trompeter et. al. eine DSL definiert als eine „Modellierungs-, Programmier- und/oder Spezifikationssprache, die sich auf die abstrakte Darstellung von Sachverhalten einer fachlichen und/oder technischen Domäne beschränkt und dies mit domänennahen Sprachmitteln effizient ermöglicht“ [TPB+07, S. 59].

DSLs werden im MDSD durch Metamodelle beschrieben, welche die Domänenkonzepte mit ihrer Semantik, ihren Constraints und ihren Beziehungen untereinander spezifizieren [Sch06, S. 21]. Vor dem Einsatz von DSLs sind Vor- und Nachteile abzuwägen, von denen im

[1] Im Original: „Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain.“

Kontext des MDSD folgende besonders hervorzuheben sind (vgl. [Fow10, S. 33-38][FR07]):

Ein Vorteil von DSLs ist, dass bei der Softwareentwicklung im Optimalfall der Personenkreis der Entwickler um den der fachlichen Domänen- bzw. Diskurssystemexperten ergänzt werden kann, da die Kommunikation mit den Domänenexperten oder im Idealfall deren aktive Einbeziehung vereinfacht wird. Eine vollständige Substitution von Softwareentwicklern durch Domänenexperten ist hingegen unrealistisch, da die endgültige Implementierung weiterhin technisch geprägt ist.

Die deklarative Form der Modellierung i.V.m. einem erhöhten Abstraktionsgrad ermöglicht eine kompakte Repräsentation von Semantik im Modell. Indem Sachverhalte im Vergleich zur Modellierung mit GPMLs durch weniger Modellelemente abgebildet werden, sind bei der Modellierung geringere Freiheitsgrade zu beachten, was ein effizienteres Vorgehen unterstützt.

Im Kontext des MDSD werden Modellelemente üblicherweise durch Sprachelemente typisiert. Die Typen repräsentieren Semantik, sodass den typisierten Modellelementen Bedeutung zugeordnet werden kann, welche von Transformationsengines oder Ausführungsumgebungen interpretiert werden kann. Aufgrund der erhöhten Expressivität von DSLs eigenen sich diese daher für das MDSD.

Nachteilig an DSLs sind die Kosten zur initialen Entwicklung und Wartung der Modellierungssprachen und -werkzeuge. Hinzu kommt kontinuierlicher Aufwand für Schulungen der Modellierer und Entwickler, da im Vergleich zu etablierten GPMLs wie z. B. der UML Methodenwissen über spezialisierte DSLs nicht vorausgesetzt werden kann.

2.3.2 Abgrenzung vom Computer-Aided Software Engineering

In der Literatur wird das MDSD als separater Ansatz zum CASE verstanden, in den aber Erfahrungen aus dem Scheitern des CASE eingeflossen sind [Sch06][SVEH07, S. 43 f.][TPB+07, S. 17, S. 244][Zim09]. Grundlegend ist dem CASE und dem MDSD die Zielsetzung einer generativen Softwareentwicklung auf Basis von Modellen gemein. Im Rückblick wurde die Etablierung des CASE in der Praxis aber aufgrund diverser Defizite gehemmt, und mit dem MDSD ein alternativer Lösungsansatz gesucht. Aufgrund der Vielfalt der CASE- und MDSD-Ansätze ist die Grenze zwischen dem CASE und dem MDSD zwar fließend, es lassen sich aber folgende prinzipielle Unterschiede ausmachen:

CASE-Werkzeuge geben in monolithischer Form teilweise bis gänzlich die verwendbaren Modellierungssprachen, Transformationsdefinitionen und Zielplattformen vor, sodass die Wahlmöglichkeiten eingeschränkt sind [SVEH07, S. 43][TPB+07, S. 17]. Das MDSD dagegen betont die Entwicklungsmethodik anstelle von Entwicklungsumgebungen und deren Einheitslösungen, indem Gestaltungsmöglichkeiten zur freien Spezifikation von Modellie-

rungssprachen und Transformationsdefinitionen geboten werden. Dies schließt die Erstellung und Nutzung von DSLs mit ein, sodass Domänenkonzepte mit Relevanz für konkrete Entwicklungsprojekte aufgegriffen werden können und auf diese Weise Abbildungsdefekte des Modells vermieden werden. Als Besonderheit des MDSD wird die Spezifikation von DSLs durch Metametamodelle ermöglicht (vgl. Kapitel 2.4), sodass sie bei der Entwicklung von Entwicklungswerkzeugen nicht vorab bekannt sein müssen, sondern diesen nachträglich durch die Werkzeugnutzer hinzugefügt werden können.

Mit den erweiterten Gestaltungsmöglichkeiten geht einher, dass Anpassungen an bestehenden Transformationsdefinitionen im MDSD vergleichsweise effektiver unterstützt werden, als im CASE. So sind die Transformationsdefinitionen bei CASE-Werkzeugen üblicherweise als Teil des jeweiligen Werkzeugs proprietär und hart in diesem codiert [TPB+07, S. 17 f.]. Im Gegensatz dazu sieht das MDSD adaptierbare Transformationsdefinitionen vor, indem diese durch standardisierte oder zumindest etablierte Transformationssprachen formuliert werden. Die Transformationsdefinitionen sind somit unabhängig vom verwendeten Werkzeug und aufgrund der spezialisierten Ausdrucksmittel der Transformationssprachen zudem übersichtlicher.

Auch weisen beim CASE die Serialisierungsformate für die Artefakte eine mangelnde Standardisierung auf, sodass Vendor-Lockin-Situationen die Folge sind. Die mangelnde Interoperabilität der CASE-Werkzeuge konterkariert die Absicht einer Modell-basierten Entwicklung, da Modelle als spezifische Investitionen zu Abhängigkeiten vom jeweiligen Werkzeughersteller führen. Für das MDSD existiert dagegen mit dem XML Metadata Interchange (XMI) [Obj07a] ein standardisiertes Serialisierungsformat.

Die Komplexität des Generats ist beim MDSD prinzipiell nicht begrenzt, da die Transformationsdefinitionen bedarfs- und domänengerecht angepasst werden können. Dagegen zielen CASE-Werkzeuge üblicherweise nicht auf bestimmte Domänen ab, sodass spezifische Codebestandteile aufgrund des universellen Anspruchs nicht generiert werden können. Bei einigen der UML-Werkzeuge des CASE resultiert dies beispielsweise darin, dass der Generator auf die Erzeugung von Klassenskeletten restringiert ist [SVEH07, S. 43]. Dem MDSD kommt bei der Absicht, optimalerweise die Implementierung vollständig automatisiert aus Modellen zu generieren, indes zugegen, dass die Zielplattformen im Vergleich zum CASE weiterentwickelt sind [Sch06, S. 25 f.]. So sind mittlerweile Programmiersprachen und Frameworks bzw. generell Softwareplattformen verfügbar, die über ein höheres Abstraktionsniveau verfügen, als es zu Zeiten von CASE der Fall war. Middleware-Konzepte wie Transaktionen, Messaging und entfernte Funktionsaufrufe werden bereits durch verfügbare Softwareplattformen bereitgestellt und müssen nicht in Modellen von Grund auf konzipiert werden, was den Abstraktionsgrad der Modelle hebt und die Modellkomplexität

senkt.

2.4 Metamodellierung

Das MDSD sieht im Unterschied zum CASE die Definition von Modellierungssprachen und im Speziellen von DSLs durch die Nutzer der Modellierungswerkzeuge vor. Üblicherweise werden zu diesem Zweck die Syntax und die statische Semantik einer Modellierungssprache durch ein sprachbasiertes Metamodell spezifiziert. Anzumerken ist, dass das Prinzip der Metaisierung zur sprachbasierten Metamodellierung zwar charakteristisch für das MDSD in der Abgrenzung zum CASE ist, in allgemeiner Form aber ebenso im Kontext der Wirtschaftsinformatik, vornehmlich zur Entwicklung von Modellierungssprachen und Vorgehensmodellen Anwendung findet (vgl. [Sch01, S. 5], [FS08, S. 130 ff.]). Im Folgenden liegt der Fokus auf der sprachbasierten Metamodellierung, da in der Arbeit der Einsatz von Metamodellierungstechniken für die Spezifikation von Modellierungssprachen im Vordergrund steht. Prozessbasierte Metamodelle sind zwar für die strukturierte Nutzung der gewonnenen Modellierungssprachen relevant und basieren ebenfalls auf dem Prinzip der Metaisierung [Bro03, S. 83 ff.], sind aber für die Betrachtung aus Sicht der MDA (vgl. Kapitel 2.5) nicht von zentraler Bedeutung.

Die OMG definiert in ihren Spezifikationen zur Metamodellierung eine Vier-Schichtenarchitektur (vgl. Abbildung 2), deren Schichten mit den Abkürzungen M0, M1, M2 und M3 bezeichnet werden ([Obj10b, S. 16 ff.], [KWB03, S. 85 ff.], [Fra03, S. 105 ff.], [MSUW04, S. 41 ff.]). Im Folgenden wird das Konzept der Metamodellierung in der Terminologie der OMG beschrieben, da sich die Arbeit auf die MDA und diese Schichten bezieht.

Schicht M0: Die Schicht M0 ist die Instanzschicht und umfasst die Elemente der Realwelt. Im Kontext des Software Engineering sind diese Elemente beispielsweise Objekte in objektorientierten Systemen zur Laufzeit oder Datensätze in Datenbanken [MSUW04, S. 41][Fra03, S. 107]. Jedes Element steht als Instanz in Beziehung zu mindestens einem Element aus der Schicht M1, das in dieser Beziehung als der Typ des Elements fungiert. Mehrere Elemente können dabei in Beziehung zu demselben Typen stehen.

Zu beachten ist, dass Instanzen in Softwaresystemen Abbildungen korrespondierender Elemente des physischen Teils der Realwelt sein können, also potentiell weitere physische Instanzen beschreiben [KWB03, S. 85]. Dies ist z. B. bei Artikel-Datensätzen der Fall, die physische Artikel repräsentieren.

Schicht M1: Die Schicht M1 umfasst Modelle und deren Elemente, mit denen die Instanzen aus der Schicht M0 typisiert werden. Die Elemente aggregieren relevante Eigen-

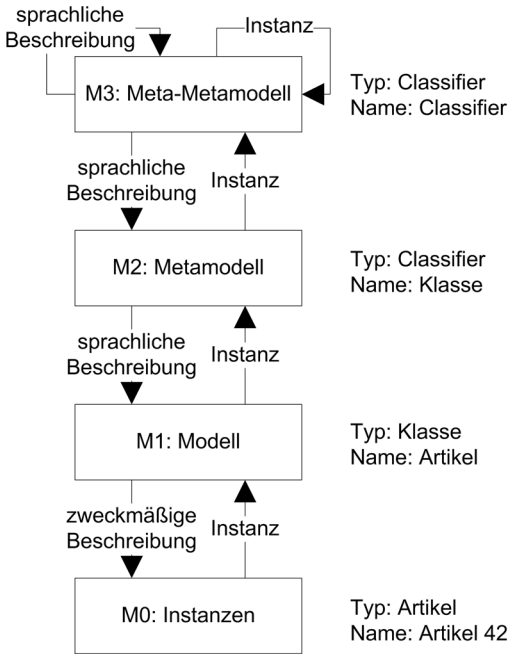

Abbildung 2: Vier-Schichtenarchitektur der MDA

schaften der Instanzen zu einem bestimmten Zweck. Im Kontext objektorientierter Modellierung enthält die Schicht M1 z. B. eine reduzierte und optimalerweise zweckkonforme Repräsentation eines Softwaresystems mit seinen Klassen (vgl. [MSUW04, S. 41]). Im Rahmen der Modellierung relationaler Datenbanken sind dies entsprechend in Schemata spezifizierte Relationen bzw. deren Implementierungen durch Datenbanktabellen. Auf der Schicht M0 werden die Attribute der Elemente der Schicht M1 mit Werten ausgeprägt. Die Wertebereiche der Attribute können durch Constraints beschränkt werden.

So ist z. B. ein UML-Klassendiagramm der Schicht M1 zugeordnet, dessen UML-Klassen korrespondierend zu objektorientierten Klassen typisierte Eigenschaften bzw. Objekt- und Klassenvariablen und damit mögliche Instanzausprägungen spezifizieren. Indem z. B. auf der Schicht M1 eine Klasse mit dem Namen *Artikel* spezifiziert wird, können auf der Schicht M0 die Instanzen der Klasse *Artikel* die Klasse als Typ referenzieren. Auf diese Weise werden Instanzen auf der Schicht M0 durch Modellelemente aus der Schicht M1 typisiert, sodass sich die Instanzen durch Typen klassifizieren lassen [KWB03, S. 85]. Instanzen müssen durch mindestens ein Modellelement typisiert werden, um im Sinne des Modells zulässig zu sein.

Schicht M2: Der Schicht M2 sind Metamodelle zugeordnet, also Modelle von Modellierungssprachen. Die Metamodelle spezifizieren Sprachelemente und Beziehungen zwi-

schen diesen, welche zur Formulierung der Modelle auf der Schicht M1 verwendet werden [KWB03, S. 86 f.]. So werden z. B. die verschiedenen sprachlichen Ausdrucksmittel der UML 2 durch ein Metamodell spezifiziert [Obj07b]. Ein Bestandteil dieser Spezifikation ist das Sprachelement *Class* [Obj07b, S. 86], das in einem UML-Diagramm als Modellelement z. B. mit dem Namen *Artikel* instanziiert werden kann. Analog zu der Beziehung der Schicht M0 zu der Schicht M1 werden die Elemente der Schicht M1 durch solche der Schicht M2 typisiert. Entsprechend lässt sich über die Instanz-Typ-Beziehungen die Konformität des Modells zu der Syntax und statischen Semantik der Modellierungssprache überprüfen

Schicht M3: Die Schicht M3 umfasst Modelle von Modellierungssprachen, mit denen die Metamodelle auf der Schicht M2 formuliert werden. Bezogen auf die Schicht M0 kommen zwei Schichten hinzu, sodass die Modelle auf der Schicht M3 als Metametamodelle bezeichnet werden, und die spezifizierten Sprachen als Metasprachen. Beispielsweise werden im Rahmen der Spezifikation der UML 2 [Obj07b] Sprachelemente wie *Class*, *Property*, *Operation* und *Parameter* als Metamodellelemente spezifiziert, die Instanzen der Metametamodellelemente bzw. Metasprachelemente der Metasprache aus der Schicht M3 sind. Analog zu den Instanz-Typ-Beziehungen zwischen Elementen aus der Schicht M1 und der Schicht M2 existieren zwischen Elementen aus der Schicht M2 und der Schicht M3 Instanz-Typ-Beziehungen.

Von der OMG ist kein Meta-Metametamodell, also ein Modell einer Metametasprache auf einer Schicht M4 vorgesehen, da bereits die von der OMG vorgesehene Metasprache MOF [Obj06b] generisch genug ist, um sich selbst reflexiv zu beschreiben [Obj10b, S. 17]. Dies ist möglich, weil das Metametamodell generisch Elemente und Beziehungen zwischen diesen spezifiziert und somit einen hinreichenden Sprachumfang bietet, um eben solche Elemente und Beziehungen zwischen Elementen zu beschreiben. Damit sind die Elemente des Metametamodells Typen und zugleich Instanzen dieser Typen [Fra03, S. 108 f.].

Die Anzahl der Schichten ist nicht statisch auf eine bestimmte Zahl festgelegt, sondern hängt von der spezifischen Problemstellung ab [Obj06b, S. 8 f.]. Ein Minimum von zwei Schichten wird aber vorausgesetzt, da sonst keine Typ-Instanz-Beziehungen existieren können. Auch die Einordnung in das Benennungsschema der MDA von M0 bis M3 ist beliebig, da im Kontext des Software Engineering die Semantik der Instanzen auf der Schicht M0 nicht festgelegt ist [Fra03, S. 107 f.]. So ist nicht sichergestellt, dass die Elemente aus der Schicht M0 nicht wiederum als Typen einer Schicht M-1 verwendet werden. Eine mögliche differenzierende Charakterisierung der Schicht M0 im Gegensatz zu den Schichten M1 bis M3 ist darin zu sehen, dass die Schicht M0 ausschließlich Elemente der Realwelt umfasst

bzw. abbildet, die Schichten M1 bis M3 dagegen Modellelemente. Da eine Einteilung in die Schichten M0 bis M3 aber in der Praxis zielführend ist, gilt als Konvention aus diesem Grund im Folgenden, dass die Metametamodellschicht konform zum Benennungsschema der OMG als M3 gekennzeichnet wird, die Metamodellschicht als M2, die Modellschicht als M1 und die Instanzschicht als M0.

Weitere etablierte Metametamodelle neben der MOF sind Ecore und XML Schema. Ecore ist Teil des Eclipse Modeling Framework und entspricht weitestgehend der Essential MOF (EMOF) [SBPM08, 524], die eine Teilmenge der MOF ist. XML Schema bietet Sprachelemente zur Definition von Schemata für XML-Dokumente [W3C04a]. Abhängig davon, ob die in einem XML-Dokument enthaltenen Instanzen der Schicht M0 oder der Schicht M1 zugerechnet werden, also Realweltelemente oder Modellelemente sind, kann XML Schema als Metamodell oder als Metametamodell klassifiziert werden.

In der praktischen Anwendung bietet die sprachbasierte Metaisierung den Vorteil, verschiedene Modellierungssprachen und Modelle auf Basis eines standardisierten Metametamodells und damit einer einheitliche Metasprache verarbeiten zu können. Dies ermöglicht sprachunabhängige Entwicklungswerkzeuge für das Sprachdesign, die Modellierung und die Transformation durch sprachunabhängige Transformations- und Generatorengines.

2.5 Model Driven Architecture

2.5.1 Grundlagen

Die Model Driven Architecture (MDA) ist eine im Jahr 2000 erstmals angeregte und im Jahr 2001 gestartete Initiative der Object Management Group [Sol00, Obj01] als Variante zur Realisierung des MDSD-Paradigmas. Das MDSD gibt mit der Absicht zur generativen Erzeugung von Code aus Modellen zwar die Zielsetzung und die generelle Methode vor, legt aber nicht die Ausgestaltung der Techniken fest. So werden abhängig von der Auslegung und Realisierung des MDSD DSLs oder GPMLs zur Modellierung verwendet, Code aus Modellen in einem einzelnen Schritt oder über intermediäre Modelle generiert und diverse Techniken zur Spezifikation und Serialisierung von Modellierungssprachen, Modellen und Transformationsdefinitionen verwendet.

Die MDA sieht daher eine Auswahl standardisierter und etablierter Techniken [Obj03, S. 7-1 f.] und Vorgehensweisen vor, um bei der Entwicklung von Softwaresystemen die drei Ziele *Portabilität*, *Interoperabilität* und *Wiederverwendbarkeit* sicherzustellen [Obj03, S. 2-2]. Zu diesem Zweck integriert die MDA einige der standardisierten Techniken der OMG. Als wichtigste gehören zu diesen die UML [Obj10b, Obj10c] als Modellierungssprache, die MOF [Obj06b] als Metamodellierungssprache zur Definition von DSLs und das Serialisierungsformat XML Metadata Interchange (XMI) [Obj07a] (vgl. [GPR06, S. 31

f.]). Weitere sind die Object Constraint Language (OCL) [Obj10a] zur deklarativen Definition von Invarianten auf Modellen, Query/View/Transformation (QVT) [Obj11b] zur Spezifikation von Modell-zu-Modell-Transformationen und die Human-Usable Textual Notation (HUTN) [Obj04] zur übersichtlichen textuellen Darstellung von Modellen (vgl. auch [FL03]). Die UML ist in der MDA die zentrale Modellierungssprache, alternativ ist aber auch die Nutzung von DSLs abhängig vom Problemkontext vorgesehen. Sowohl die UML als auch DSLs werden durch die OMG-eigene Metasprache MOF spezifiziert.

Die MDA ist Teil des generellen Trends in der Softwareindustrie, zur Komplexitätsreduktion schichtweise von der zugrundeliegende Hardware zu abstrahieren [MF05][Obj03, S. 1-3]. In der MDA wird die Differenzierung der Abstraktionsebenen durch eine Einteilung von Modellen als berechnungsunabhängige Modelle (Computation Independent Model (CIM)), plattformunabhängige Modelle (Platform Independent Model (PIM)) sowie plattformspezifische Modelle (Platform Specific Model (PSM)) vorgenommen [Obj03, S. 2-5 f.]. Das CIM richtet sich als Domänenmodell an Domänenexperten und befasst sich mit den Anforderungen an das System sowie der Umgebung des Systems, ohne auf Details der Systemstruktur und Implementierung einzugehen. Das PIM beschreibt die Funktionalität des Systems und bildet den Teil der Systemspezifikation ab, der sich nicht mit einem Plattformwechsel ändert. Das PSM setzt die Funktionalität des PIM in Verbindung mit technischen Details der Implementierung, indem es spezifiziert, wie das System die Plattformschnittstellen nutzt.

Die Unterscheidung von PIM und PSM anhand der Plattformunabhängigkeit als Kriterium setzt die Identifikation einer Plattform als Referenzpunkt voraus, da Plattformunabhängigkeit an sich ein relatives Konzept ist [Fra03, S. 48]. So kann z. B. das Modell eines Javaprogramms als plattformunabhängig bezeichnet werden, falls die Plattform ein bestimmter Betriebssystemtyp ist, dagegen als plattformspezifisch, falls Java als die Plattform gilt. Nach der OMG ist eine Plattform definiert als eine Menge von Subsystemen und Techniken, die eine zusammenhängende Menge von Funktionalität durch Schnittstellen und bestimmte Nutzungsmuster anbieten, welche von jeder Applikation genutzt werden kann, die durch die Plattform unterstützt wird, ohne die Implementierung der Funktionalität durch die Plattform beachten zu müssen[2] [Obj03, S. 2-3]. Plattformen können somit verschiedenen Abstraktionsebenen zugeordnet sein, wie z. B. im Fall maschinennaher Befehlssätze der Prozessorarchitektur x86, objektorientierter Programmierschnittstellen von Java EE oder prozessorientierter Workflowengines.

[2] Im Original: „A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.“

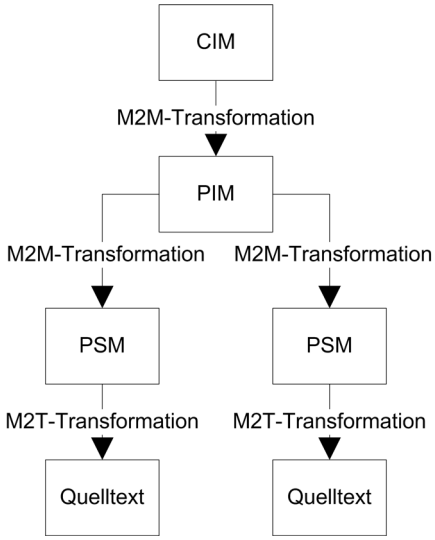

Abbildung 3: Abstraktionsebenen der MDA

Durch die Modellierung von Systemen auf verschiedenen Abstraktionsebenen als CIM, PIM und PSM wird im Idealfall für verschiedene Adressatengruppen eine vollständige Sicht auf das repräsentierte System gegeben. Die Spezifikation der plattformspezifischen Implementierungsdetails eines Systems werden von den fachlich-abstrakteren Beschreibungsformen getrennt, sind aber mit ihnen über Transformationsdefinitionen bzw. Mappings in Beziehung gesetzt [Obj03, S. 3-2 ff.]. Dies soll im Optimalfall eine vollständig automatisierte Transformation eines Modells in ein Modell der nächstniedrigeren Abstraktionsstufe ermöglichen, also z. B. vom PIM zum PSM (vgl. Abbildung 3).

2.5.2 Meta Object Facility

Die Meta Object Facility (MOF) ist ein Standard der OMG zur Beschreibung von Metadaten, insbesondere von Modellierungssprachen durch Metamodelle im Rahmen der sprachbasierten Metamodellierung. Die MOF wurde erstmals im Jahr 1997 durch die OMG ratifiziert [Fra03, S. 97] und in Verbindung mit der UML weiterentwickelt [Obj06b, S. 5]. Die Verbindung resultiert daraus, dass Teile der UML in die MOF integriert sind und für die Zwecke der sprachbasierten Metamodellierung der MOF modifiziert und erweitert werden.

Die aktuelle Version 2.0 der MOF ist unterteilt in die EMOF als Kernmetamodell und die Complete MOF (CMOF) als Metamodell mit erweitertem Umfang und somit höherer Komplexität. Dies ermöglicht ähnlich zu der Aufteilung der UML in einen Sprachkern sowie dessen Erweiterungen die Standardkonformität von Software, welche nur die EMOF implementiert. Ein Beispiel für eine solche Software ist das Eclipse Modeling Framework (EMF), das sich auf die Implementierung der EMOF beschränkt.

Die EMOF ist eine vereinfachte Untermenge der MOF und verbindet das Basispaket der UML (InfrastructureLibrary::Core::Basic, vgl. [Obj10b, S. 91-102]) mit Funktionalität zum Umgang mit Metadaten [Obj06b, S. 31].

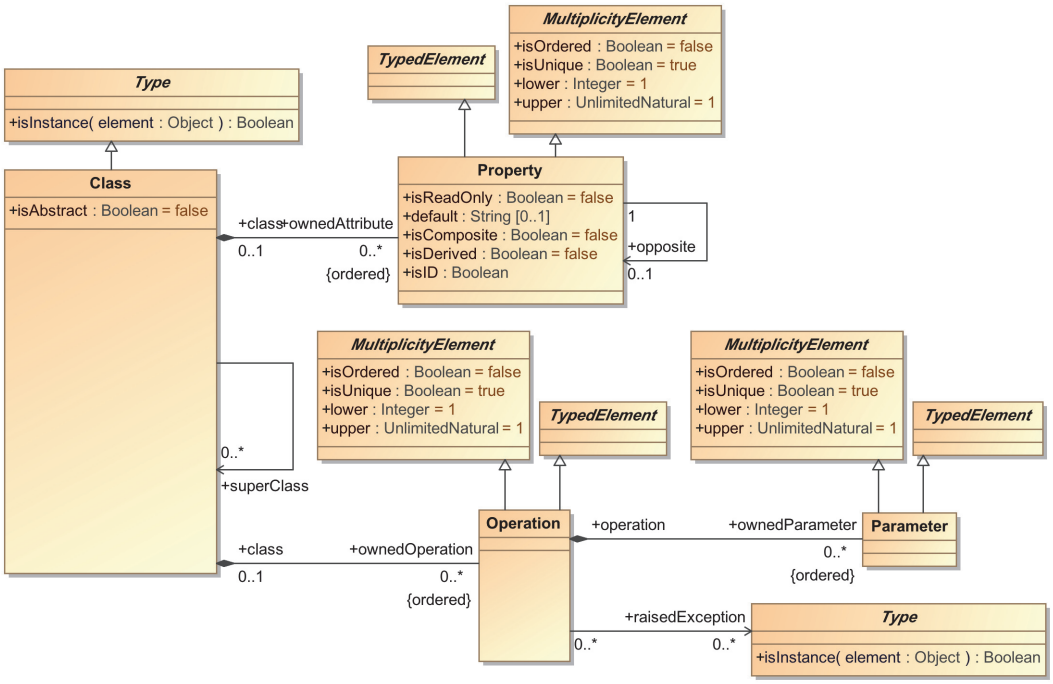

Abbildung 4: MOF Classes (Abbildung nach [Obj06b, S. 33])

Das Metamodell der EMOF spezifiziert im Wesentlichen Sprachelemente für Klassendiagramme zur objektorientierten Informationsmodellierung. Der Kern ist aus dem Paket *Basic* der UML Infrastructure übernommen und umfasst Klassen (*Class*) als Spezialisierung von *Typen*, welche *Properties* und *Operationen* besitzen (vgl. Abbildung 4). Eine *Operation* kann *Parameter* besitzen und Exceptions in Form von *Typen* werfen. *Properties*, *Operationen* und *Parameter* sind typisiert, da sie Spezialisierungen von *TypedElement* sind und besitzen als Spezialisierung von *MultiplicityElement* einfache oder mehrfache Multiplizität.

Neben dem Element *Class* ist *DataType* eine weitere Spezialisierung von *Type* und ermöglicht in weiteren Spezialisierungen die Modellierung von primitiven Datentypen (*PrimitiveType*) sowie von *Enumerationen* (vgl. Abbildung 5). *Enumerationen* umfassen Literale (*EnumerationLiteral*) als vordefinierte Werte. Das Element *Type* erbt von *NamedElement* ein Namensattribut, sodass Modellelemente benannt werden können.

Jeder *Type* kann zudem optional einem *Package* zugeordnet werden, das in einer Hierarchie von *Packages* strukturiert werden kann (vgl. Abbildung 6).

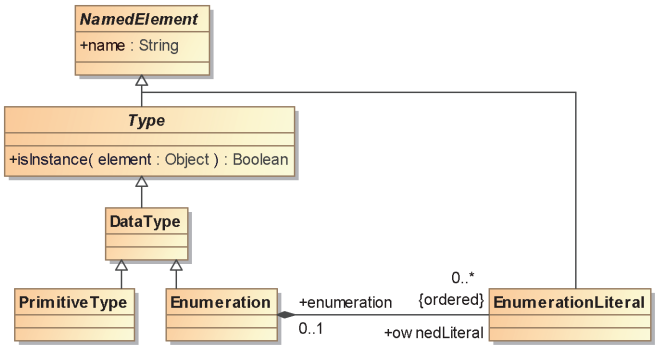

Abbildung 5: MOF Data Types (Abbildung nach [Obj06b, S. 34])

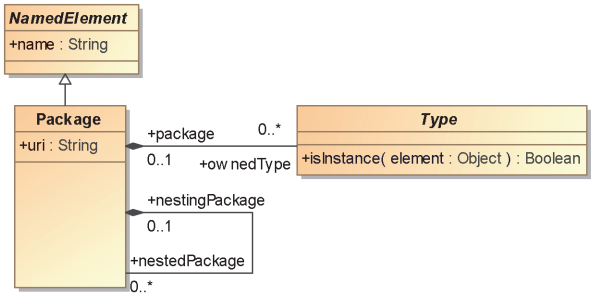

Abbildung 6: MOF Packages (Abbildung nach [Obj06b, S. 34])

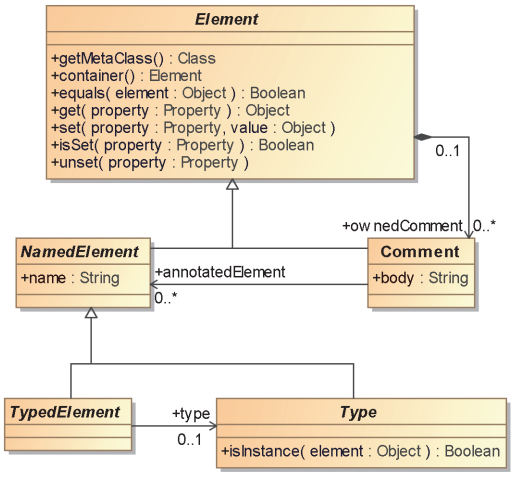

Abbildung 7: MOF Types (Abbildung nach [Obj06b, S. 35])

Sämtliche Metamodellelemente sind direkt oder indirekt Spezialisierungen von *Element* und können aufgrund der Referenz zwischen *Element* und *Comment* textuell annotiert werden (vgl. Abbildung 7). Typisierte Elemente (*TypedElement*) referenzieren zudem ihren *Type*.

Über die Paketzusammenführung (Merge) aus dem Package *Reflection* der MOF kommen einige weitere Metamodellelemente hinzu. Im Wesentlichen wird jedem *Package* eine *Factory* zugeordnet, welche der Instanziierung von Metamodellelementen zu Modellelementen dient. Durch den Merge aus dem *Identifiers* Package von MOF können Klassen durch *Properties* in Form von Schlüsselattributen identifiziert werden, deren Kontext durch einen *Extent* definiert wird, welcher den Gültigkeitsbereich von Schlüsseln definiert. Durch den Merge des Package *Extension* können *Elemente* mit *Tags* annotiert werden, welche sich paarweise aus Namen und Werten zusammensetzen.

Das Metamodell der CMOF ist ausschließlich eine Paketzusammenführung (Merge) aus verschiedenen bereits existierenden Metamodellpaketen und spezifiziert damit keine eigenen Elemente [Obj06b, S. 45]. Der Kern der CMOF besteht aus den Metamodellelementen der EMOF und wird um Pakete aus der UML Infrastructure (*PrimitiveTypes*, *Constructs*) sowie aus der MOF (*CMOFExtension*, *CMOFReflection*) ergänzt. Im Wesentlichen sind die Unterschiede zwischen EMOF und CMOF analog zu den Unterschieden zwischen den UML-Paketen *Basic* und *Constructs*, da die EMOF auf dem Paket *Basic* und CMOF auf dem Paket *Constructs* basieren. Da das UML-Paket *Constructs* deutlich umfangreicher als das Paket *Basic* ist, gilt dies auch im Vergleich von CMOF und EMOF. So umfasst die CMOF beispielsweise Sprachelemente für *Assoziationen* und Namensräume (*Namespace*).

Analog zu der EMOF erweitert die CMOF das zugrunde liegende UML-Paket um Mechanismen für Reflexion. Im Unterschied zu dem von der EMOF verwendeten MOF-Paket *Reflection* besitzt das von der CMOF eingebundene MOF-Paket *CMOFReflection* aber einen mächtigeren Reflexionsmechanismus. So können beispielsweise Modellelemente gelöscht werden, *Argumente* bei der Instanziierung von Metamodellelementen übergeben werden und die *Assoziationen* der CMOF instanziiert werden.

Sowohl die Metamodelle der EMOF als auch der CMOF werden durch die CMOF als Modellierungssprache spezifiziert. Die CMOF ist also selbstbeschreibend bzw. ihr eigenes sprachbasiertes Metamodell. Die EMOF wird sowohl durch die CMOF, als auch durch die EMOF spezifiziert, ist also ebenfalls selbstbeschreibend [Obj06b, S. 11]. Die Spezifikation der EMOF durch die CMOF geschieht auf Basis von Paketzusammenfassungen (Merges) [Obj06b, S. 32], welche in der CMOF als Modellierungsmittel vorgesehen sind. Im Unterschied dazu geschieht die Spezifikation der EMOF durch die EMOF mit den Ausdrucksmitteln des *EMOF Merged Model*, das ein Resultat von Paketzusammenfassungen ist

[Obj06b, S. 33 ff.] und somit einen Bedarf für den Merge-Mechanismus der CMOF umgeht.

2.6 Eclipse Modeling Framework

2.6.1 Grundlagen

Die praktische Umsetzung der OMG-Standards und die Realisierung des theoretischen Architekturkonzepts der MDA setzt entsprechende technische Implementierungen voraus. Für die Entwicklung der in der Arbeit vorgestellten Generatorframeworks ist insofern zum Zweck der Wiederverwendung und Standardkonformität eine solche Implementierung zu wählen.

Das Eclipse Modeling Project[3], das im Rahmen des Eclipse Project[4] unterhalten wird, befasst sich als ein industriell etablierter Rahmen mit der Entwicklung modellgetriebener Werkzeuge und Standardimplementierungen. Es erweitert die Basisfunktionalität der Eclipse IDE um eine integrierte Entwicklungsumgebung für die modellgetriebene Softwareentwicklung und umfasst eine Vielzahl von Entwicklungswerkzeugen, Modellierungsframeworks und Laufzeitumgebungen für diverse Transformationssprachen. Zu diesen zählt das Eclipse Modeling Framework (EMF)[5], das in einem Unterprojekt des Eclipse Modeling Project entwickelt wird und Funktionalität zur Erstellung, Nutzung, Modifikation und Persistierung von Modellen und Modellierungssprachen bietet. Hinzu kommen Generatorfähigkeiten, welche die Entwicklung modellgetriebener Entwicklungswerkzeuge unterstützen.

Im EMF wird konzeptionell zwischen dem Metamodell bzw. dessen Metamodellelementen, Modellen bzw. deren Modellelementen und Instanzen von Modellelementen unterschieden [SBPM08, S. 122], deren Ebenen im Folgenden als *Metamodellebene*, *Modellebene* und *Instanzebene* bezeichnet werden. Modelle werden im EMF auf der Modellebene in der Sprache Ecore erstellt und können mit ihren Modellelementen auf der Instanzebene instanziiert werden. Ecore ist eine in EMF implementierte Modellierungssprache mit Ähnlichkeit zu MOF, die Sprachelemente zur Formulierung von Klassendiagrammen auf der Modellebene bietet und durch ein Metamodell auf der Metamodellebene spezifiziert ist.

Ein Modell kann in EMF abhängig vom Zweck auf diverse Weisen repräsentiert bzw. persistiert werden (vgl. [SBPM08, S. 28-31]), von denen folgende vorrangig sind:

- Die native Repräsentationsform für ein Modell in EMF ist das Ecore-basierte Modell, dessen Modellelemente und Instanzen zur Laufzeit als Java-Objektgraph und serialisiert als XMI-Dokument manifestiert werden. Ein Ecore-basiertes Modell kann

[3] `http://eclipse.org/modeling/`
[4] `http://eclipse.org/`
[5] `http://www.eclipse.org/modeling/emf/`

durch hartcodierte Transformationsdefinitionen in ein UML-Modell, ein XML-Dokument oder in Javaklassen als alternative Darstellungsvarianten transformiert werden. Ebenso können diese Formen in ein Ecore-basiertes Modell überführt werden, sodass die Transformationen bidirektional definiert sind. [SBPM08, S. 122-144]

- Alternativ ist die Repräsentation des Modells durch annotierte Javaklassen möglich. Die Modellierungssprache auf der Metamodellebene ist demnach die Programmiersprache Java, mit der in Form von Javaklassen auf der Modellebene ein Modell erstellt wird, welches zur Laufzeit auf der Instanzebene in Form von Javaobjekten instanziiert werden kann. Die Javaklassen können aus einem Ecore-basierten Modell durch einen in EMF enthaltenen Codegenerator erzeugt werden und enthalten in Form von Annotationen zusätzliche Metainformationen, die z. B. ausdrücken, ob der Wert einer Objektvariable als Schlüsselvariable ein Objekt identifiziert. [SBPM08, S. 163-185]

- Die Repräsentation als UML-Modell ermöglicht den Einsatz von UML-Werkzeugen, indem ein Ecore-basiertes Modell als UML-Modell exportiert werden kann bzw. entgegengesetzt ein UML-Modell importiert werden kann. Eine Instanziierung der Modellelemente wie bei Ecore-basierten Modellen ist nicht vorgesehen. Der unterstützte Sprachumfang von UML ist dabei rudimentär und umfasst lediglich die für Klassendiagramme benötigten Sprachelemente. [SBPM08, S. 145-162]

- Eine weitere Möglichkeit zur Repräsentation des Modells ist die Darstellung als XML-Schemadefinition. Auf der Modellebene entsprechen den Klassen eines Ecore-basierten Modells dabei z. B. komplexe XML-Datentypen [SBPM08, S. 197]. Auf der Instanzebene werden korrespondierend zu den Instanzen eines Ecore-basierten Modells die Instanzen als Elemente eines XML-Dokuments dargestellt, welches valide bzgl. der XML-Schemadefinition sein muss. Sowohl das Modell, als auch die Instanzen können bidirektional transformiert werden. Die Transformation ist insbesondere hilfreich für die im Rahmen dieser Arbeit angewandte automatisierte Generierung Ecore-basierter Modellierungssprachen aus XML-Schemadefinitionen (vgl. Kapitel 5). [SBPM08, S. 186-259]

Für EMF existieren über die sprachlichen Ausdrucksmittel von Ecore hinaus keine Restriktion bzgl. der abzubildenden Semantik, sodass auf der Modellebene mit Ecore durch Klassendiagramme sowohl fachliche Sachverhalte als auch Modellierungssprachen beschrieben werden können. Abhängig davon, ob Ecore der fachlichen Modellierung oder der sprachbasierten Metamodellierung dient, ist Ecore im Sinne der MDA (vgl. Kapitel 2.4) der Schicht M2 oder der Schicht M3 zuzuordnen, was sich entsprechend auf die Zuordnungen der darunterliegenden Schichten auswirkt (vgl. Abbildung 8).

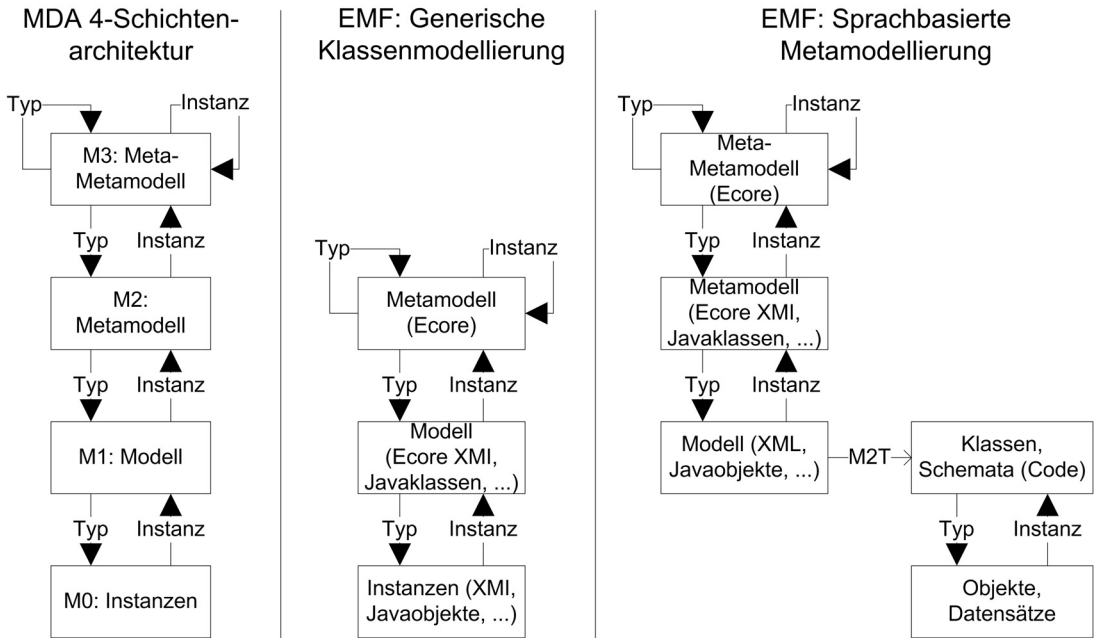

Abbildung 8: Einordnung von EMF in die Vier-Schichtenarchitektur der MDA

Falls Ecore im Rahmen einer sprachbasierten Metamodellierung als Metasprache zur Modellierung einer Modellierungssprache verwendet wird, ist es der Schicht M3 zuzuordnen. Auf der Modellebene im Sinne von EMF liegt demzufolge das Modell der Modellierungssprache (M2), repräsentiert durch eine der beschriebenen Darstellungsformen von Modellen in EMF. Auf Basis der Modellierungssprache können auf der Instanzebene im Sinne von EMF schließlich Modelle erstellt werden (M1). Falls diese Modelle instanziierbare Sachverhalte wie z. B. Klassen beschreiben, sind die Instanzen z. B. dieser Klassen der Schicht M0 im Sinne der MDA zuzuordnen. Da sich EMF auf die drei genannten Ebenen beschränkt, wird die Realisierung der Schicht M0 als vierte Ebene nicht durch EMF abgedeckt. Für den Schritt von der Schicht M1 zu der Schicht M0 ist deshalb die Instanzebene im Sinne von EMF in eine alternative Repräsentationsform zu übersetzen. Im Kontext des MDSD wird dazu z. B. das auf der Instanzebene enthaltene Modell in Quelltext bzw. Code transformiert, der zur Laufzeit zu Instanzen wie z. B. Objekten in objektorientierten Systemen oder Datensätzen in Datenbanken führt. Wie auch bei MOF wird eine Schicht M4 nicht benötigt, da Ecore mit Mitteln von Ecore modelliert wird, also als sein eigenes Metamodell agiert [SBPM08, S. 122].

2.6.2 Ecore

Ecore ist eine Modellierungssprache des EMF, die durch ein gleichnamiges Metamodell implementiert wird und Sprachelemente für Klassendiagramme bietet. Ein verbreiteter und

intendierter Spezialfall ist die Verwendung von Ecore zur Spezifikation von Modellierungssprachen im Sinne der Vier-Schichtenarchitektur der MDA. Ähnlich zu MOF drückt Ecore auf einer abstrakten Ebene aus, dass es Elemente gibt, die Eigenschaften besitzen, in Beziehung zueinander stehen und durch Pakete gruppiert werden. Die wesentlichen Elemente des Metamodells sind *EClass*, *EDataType*, *EAttribute* und *EReference*, die Spezialisierungen von *ENamedElement* sind (vgl. Abbildung 9 sowie [11.11b][SBPM08, S. 122-138] hierzu und im Folgenden):

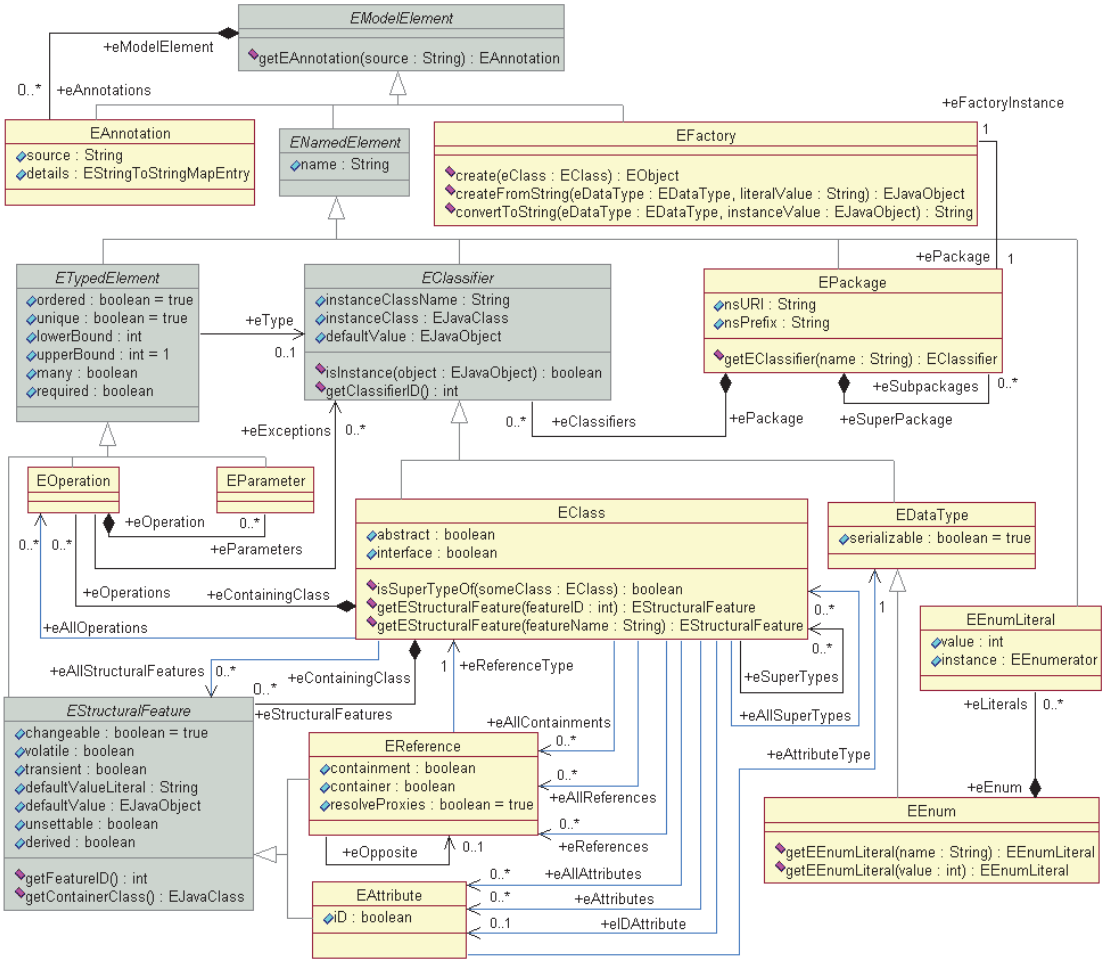

Abbildung 9: Ecore-Metamodell (Abbildung aus [11.11b])

EClass typisiert Klassen. Das Attribut *abstract* drückt aus, ob eine *EClass* abstrakt im Sinne der objektorientierten Programmierung ist. *EClass* erbt über das Element *EClassifier* vom Element *ENamedElement* das Attribut *name* zur Benennung. Eine *EClass* kann über die Referenzierung von *EAttribute* Attribute enthalten und über die Referenz zu *EReference* andere *EClasses* referenzieren. Über die Referenz *eSuper-*

Types ist eine mehrfache Generalisierung durch Oberklassen und damit verbundene mehrfache Vererbung möglich.

Ecore ist selbstbeschreibend, da Ecore (EMM) das sprachbasierte Metamodell von Ecore (EM) ist. So enthält EM z. B. die Elemente *EClass*, *EClassifier*, *EReference* und *EAttribute*, welche den EMM-Typ **EClass** haben. Da der EMM-Typ **EClass** vom EMM-Typ **ENamedElement** das Attribut **name** erbt, ist es z. B. möglich, dem EM-Element *EClassifier* den Namen „EClassifier“ zu geben. Des Weiteren kann es in EM als abstrakt definiert werden, da der EMM-Typ **EClass** das eben genannte Attribut **abstract** besitzt. EM-Elemente mit dem EMM-Typ **EClass** können Attribute besitzen, da der EMM-Typ **EClass** mit der EMM-Referenz **eAttributes** Elemente vom EMM-Typ **EAttribute** referenzieren kann.

Zur Präzisierung wird im Folgenden eine Klasse als EClass, ein Attribut als EAttribute, eine Referenz als EReference, etc. bezeichnet, wobei das Element aus EM *herausgestellt* wird, das typisierende Element aus EMM dagegen nicht.

EPackage gruppiert *EClassifier* und damit unter anderem *EClasses*. Die EClass *EPackage* erbt von der generalisierenden EClass *ENamedElement* das EAttribute *name* zur Benennung. Instanzen von *EPackage*, welche im selben Modell enthalten sind, können denselben Namen haben, da die Unterscheidung durch Namensräume anhand des identifizierenden EAttribute *nsURI* vorgenommen wird [SBPM08, S. 137]. *EPackages* können über die EReference *eSuperPackage* hierarchisch geschachtelt werden.

EDataType typisiert primitive Datentypen, die einem Ecore-basierten Modell als Instanzen hinzugefügt und *EAttributes* zugewiesen werden können.

EEnum spezialisiert *EDataType* zu einer Enumeration, deren Wertmenge im Modell durch die Instanzen von *EEnumLiteral* definiert wird.

EAttribute dient der Attributierung von *EClasses*. Ein *EAttribute* hat aufgrund der indirekten Spezialisierung von *ENamedElement* einen Namen und ist über die EReference *eAttributeType* zu *EDataType* mit einem primitiven Datentyp typisiert.

EReference wird zur Referenzierung von *EClasses* verwendet. Eine *EClass* kann *EReferences* besitzen und über diese *EClasses* referenzieren, welche über die EReference *eReferenceType* vermerkt werden. Bidirektional navigierbare Referenzen werden ausgedrückt, indem zwei *EReferences* durch die EReference *eOpposite* in Beziehung zueinander gesetzt werden. Modelle werden in EMF grundsätzlich durch hierarchische Containment-Beziehungen strukturiert. Eine *EClass* kann beliebig viele *EClasses* ent-

halten, indem sie diese durch eine *EReference* referenziert, deren Attribut *containment* den Wert *true* aufweist.

2.6.3 Vergleich von Ecore und MOF

Die MDA sieht den Einsatz von Standards für das MDSD vor, um die Ziele der Portabilität, Interoperabilität und Wiederverwendbarkeit sicherzustellen. Der folgende Vergleich von Ecore und MOF dient der Feststellung, inwieweit Ecore als standardkonforme Implementierung der EMOF gelten kann und somit den Standardisierungsbemühungen der OMG entspricht.

MOF und Ecore haben sich im Laufe der Weiterentwicklung der Versionen zunehmend angenähert. Sowohl der Standard MOF 1.4 [Obj02] als auch Ecore als dessen implementierungsnahe Interpretation unterstützen die klassenbasierte Modellierung von Systemen und Modellierungssprachen. Im Vergleich weisen sie neben einer Vielzahl an Detailunterschieden aber insbesondere Abweichungen im Hinblick auf Beziehungen zwischen Klassen auf, welche bei MOF 1.4 durch Assoziationen und bei Ecore durch gegenseitige Referenzen realisiert werden [GR03].

Als Weiterentwicklung der MOF 1.4 wurde mit MOF 2.0 das Metamodell in die implementierungsnahe EMOF und die komplexere CMOF unterteilt (vgl. Kapitel 2.5.2). Eine Motivation für die Differenzierung war, Konvergenz zwischen EMOF und EMF bzw. Ecore anzustreben. Als Resultat ähneln sich EMOF und Ecore bzgl. ihrer Struktur weitestgehend, sodass sowohl EMOF als auch Ecore durch EMF unterstützt werden. [SBPM08, S. 56 f.]

Beim Vergleich von Ecore und EMOF (vgl. Abbildungen 9 und 10) sind als wichtigste Unterschiede zu nennen:

Primitive Datentypen In EMOF wird ein primitiver Datentyp durch die Class *PrimitiveType* als Spezialisierung von *DataType* modelliert, in Ecore dagegen fallen diese beiden Metamodellelemente in der EClass *EDataType* zusammen.

EMOF definiert basierend auf der UML [Obj10b, S. 171-175] im Metamodell die primitiven Datentypen *Boolean*, *Integer*, *String* und *UnlimitedNatural* als Instanzen des Meta-Metamodellelements *PrimitiveType*, die zur Typisierung getypter Metamodellelemente dienen. So besitzt z. B. das Metamodellelement *NamedElement* das Attribut *name*, das vom Typ *String* ist und dazu als Typ die PrimitiveType-Instanz *String* referenziert.

Ecore dagegen bietet aufgrund der Implementierung auf Basis von Java eine größere Menge an primitiven Datentypen wie z. B. *EBoolean*, *EInt*, *ELong*, *EFloat* und *EDate*,

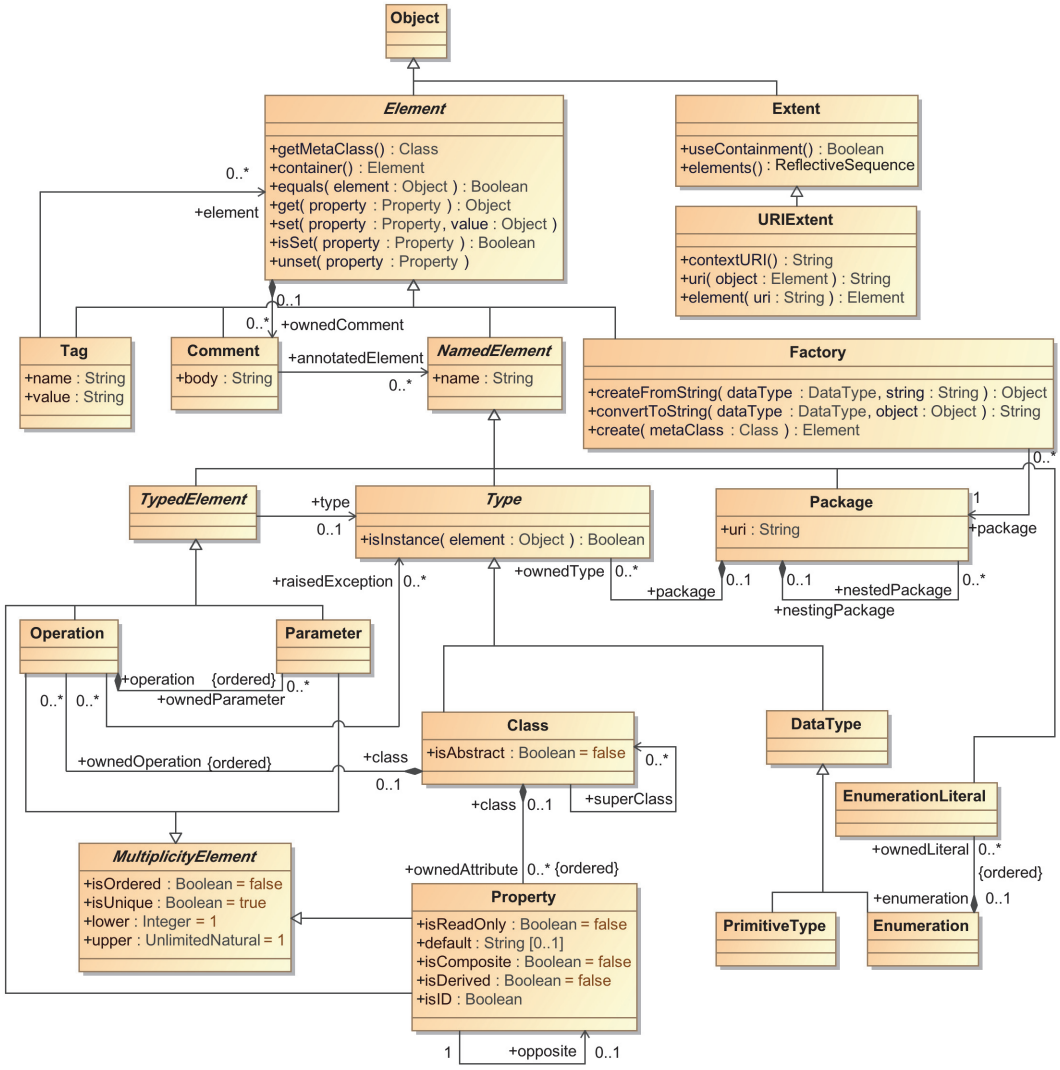

Abbildung 10: EMOF [Obj06b, S. 15-38] mit Anordnung analog zu Ecore (vgl. Abbildung 9)

die Datentypen von Java repräsentieren [SBPM08, S. 143 f.].

Differenzierung von Attributarten In EMOF werden sowohl primitive Attribute als auch Referenzattribute als *Property* modelliert. Die Unterscheidung geschieht auf Basis des Typs, welcher im geerbten Attribut *type* gespeichert wird. Ein primitives Attribut verweist im Typ auf eine Instanz von *DataType*, ein Referenzattribut dagegen auf eine *Class*.

Dagegen werden in Ecore ein primitives Attribut als *EAttribute* und ein Referenzattribut als *EReference* modelliert. Die Unterscheidung ist notwendig, da z. B. nur *EReferences* die Containment-Hierarchie definieren und nur *EAttributes* als ID verwendet werden können. Beide Attributarten werden ähnlich zu EMOF über die EClass *ETypedElement* typisiert, sind aber bzgl. der zuweisbaren Typen auf *EClasses* bzw. *EDataTypes* beschränkt.

Multiplizität Ecore verzichtet auf das *MultiplicityElement* von EMOF und ordnet dessen Attribute dem *TypedElement* zu. Dies ist möglich, da in beiden Metamodellen die Menge der getypten Elemente und die Menge der Elemente mit Multiplizität übereinstimmen.

Insofern ist das Metamodell von Ecore bzw. EMF aufgrund der Abweichungen in den Details nicht vollständig standardkonform zu EMOF. In der Praxis sprechen aber für den Einsatz von Ecore und EMF erstens die weitestgehende Ähnlichkeit zwischen Ecore und EMOF, zweitens die daraus resultierende direkte Unterstützung von MOF-basierten Modellen durch EMF [SBPM08, S. 56 f.] und drittens die Etablierung von EMF in der Praxis durch eine Vielzahl modellgetriebener Entwicklungswerkzeuge.

2.7 Modell-zu-Text-Transformation

2.7.1 Grundlagen

Die Transformation von Modellen in lauffähige Softwaresysteme wird im MDSD üblicherweise über den intermediären Schritt der Generierung von Quelltext realisiert. Die dazu benötigten Transformationsdefinitionen können auf verschiedene Weisen spezifiziert werden. In der Praxis werden zur Beschreibung sowohl generische Programmiersprachen wie z. B. Java als auch Transformationssprachen verwendet [10.10c, S. 24 ff.].

Transformationssprachen zur Modelltransformation sind domänenspezifische Programmiersprachen zur deklarativen, prozeduralen oder gemischt deklarativ-prozeduralen Beschreibung von Modelltransformationen [SK03, S. 5]. Diese Einordnung trifft sowohl auf Transformationssprachen für M2M-Transformationsdefinitionen, als auch auf solche für

M2T-Transformationsdefinitionen zu, da in beiden Fällen Modelle die Quellartefakte sind. Zudem kann abhängig von der Modelldefinition der Quelltext im weitesten Sinne als ein Modell des Systems verstanden werden, das als Zielartefakt durch eine Transformation generiert wird [MG06b, S. 129]. Der Vorteil einer Transformationssprache liegt im Vergleich zu einer General Purpose Programming Language (GPPL) in der abstrakteren und kompakteren Repräsentation der Transformationslogik durch domänenspezifische sprachliche Ausdrucksmittel.

Templatesprachen sind im Kontext des MDSD eine Spezialform der Transformationssprachen. Sie werden im MDSD primär zur Spezifikation unidirektionaler M2T-Transformationen eingesetzt, finden aber auch vereinzelt Verwendung für M2M-Transformationsdefinitionen [CP09, CP05]. Ein generelles Charakteristikum von Templatesprachen ist, dass sie darauf abzielen, dynamisch berechnete Inhalte in statischen Text einzufügen. Ein Template ist ein textuelles Dokument, das Ausdrücke der Templatesprache enthält, welche die Anreicherung des Ausgabetexts um berechnete Werte durch eine Template-Engine steuern. Im einfachsten Fall umfassen die Ausdrücke Lesezugriffe auf Variablen, abhängig von der Komplexität der Templatesprache sind aber auch Schreibzugriffe, Kontrollstrukturen und beliebige weitere sprachliche Ausdrucksmittel verfügbar. Damit entspricht ein Template im Unterschied zu einem textgenerierenden Programm im Optimalfall weitestgehend dem gewünschten Ausgabetext [Par04]. Diese Eigenschaft macht Templatesprachen für verschiedene Anwendungsdomänen relevant, zu denen neben dem MDSD aufgrund der Dokumentenorientierung insbesondere die Webentwicklung zählt.

Im Vergleich zu der Implementierung von M2T-Transformationsdefinitionen durch textgenerierende Programme bieten templatebasierte Lösungen einige Vorteile [SVEH07, S. 145-157]. Abhängig von der GPPL sind Operationen auf Strings mit syntaktischem Ballast verbunden, wie z. B. im Fall des Operationssymbols „+“ für Stringkonkatenation, und Abfragen auf komplexen Objektstrukturen aufwendig zu implementieren [SVEH07, S. 151]. Falls die Templatesprache auf wesentliche Sprachelemente begrenzt ist, wird zudem erzwungen, dass komplexere Generatorlogik getrennt von den Templatetexten durch komplementäre Berechnungslogik repräsentiert wird und somit das Prinzip der Trennung der Belange (Separation of Concerns) eingehalten wird. Die Trennung bewirkt eine klare Repräsentation des Ausgabetexts im Generator, erhöht die Wiederverwendbarkeit aufgrund der Zerlegung von Templates in Subtemplates und verbessert auf diese Weise die Wartbarkeit und Arbeitsteilung ([Par04, S. 2] im Webkontext).

Aus Sicht des Codegenerators bestehen die Templates üblicherweise aus untypisierten Stringmustern, welche durch Templatelogik angereichert werden [CH06, S. 625]. Dies ist problematisch, da dem Entwickler Syntaxfehler im Generat erst im Anschluss an die

Transformation gemeldet werden. Des Weiteren wirkt sich wie bei jeder domänenspezifischen Programmiersprache die beschränkte Ausdrucksfähigkeit der Templatesprache nachteilig aus, falls relevante Sprachelemente fehlen, sodass die Sprache nicht zweckadäquat ist. Diesem Problem kann aber aufgrund der Vielzahl verfügbarer Templatesprachen begegnet werden. Ein weiterer Nachteil ist der mit der Sprache verbundene Einarbeitungsaufwand für den Entwickler. Dies wird relativiert, da der Aufwand primär von der Komplexität der Sprache abhängt und eine Templatesprache sinnvollerweise einen geringen Sprachumfang aufweisen sollte.

2.7.2 Xpand

Xpand ist eine Transformationssprache zur Spezifikation von M2T-Transformationsdefinitionen, die ursprünglich im Rahmen des MDSD-Generatorframeworks openArchitectureWare (oAW) [EFH+08] entwickelt wurde. Im Jahr 2009 wurden die Kernkomponenten von oAW in das Eclipse Modeling Project migriert [09.09], in dessen Unterprojekt mit dem Namen Model to Text (M2T) Xpand seitdem weiterentwickelt wird. Xpand ist eine statisch typisierte Templatesprache, mit der auf in Eingabemodellen enthaltene Werte zugegriffen werden kann, um diese kombiniert mit statischem Text in Ausgabedateien zu schreiben. Transformationsregeln werden in Xpand in Template-Dateien in Form von DEFINE-Blöcken formuliert, welche aus Kombinationen von statischem Text und Xpand-Tags bestehen. Während der Ausführung der Transformationsengine werden die Tags ausgewertet und dienen sowohl der Steuerung des Ablaufs der Generierung als auch der Berechnung von Ergebnissen, welche zusammen mit dem statischen Text ausgegeben werden.

Xpand-Tags werden syntaktisch vom restlichen statischen Text durch Klammern abgegrenzt. Abhängig von der Wahl der Klammerzeichen wird damit die Notwendigkeit für eine Kollisionsbehandlung durch einen Escape-Mechanismus geschaffen, um die Klammersymbole neben der Verwendung als Diskriminatoren in Form von Metazeichen auch als Zeichen im statischen Text darstellen zu können. Im Kontext einer Programmiersprache ist dies üblicherweise kein Problem, da statischer Text in den meisten Fällen als natürlichsprachlicher Text nicht in Programmcode transformiert wird, sondern nur verarbeitet wird. Damit unterliegt er keiner formalen Syntax, sodass durch eine adäquate Wahl des Diskriminatorzeichens z. B. als Anführungszeichen die Anzahl der Kollisionen und damit die Anwendung des Escape-Mechanismus minimiert werden kann. Dagegen wird mit Xpand in üblichen Anwendungsfällen sowohl natürlichsprachlicher als auch syntaktisch geformter Text für beliebige Programmiersprachen erzeugt, sodass im Generat entsprechend sowohl beliebige Metazeichen als auch beliebige Zeichen enthalten sein können. Die Wahl inadäquater Meta-Metazeichen als Meta-Diskriminatoren in Xpand würde damit im schlechtes-

ten Fall zur ausufernden Anwendung des Escape-Mechanismus führen, beispielsweise bei der Generierung von XML-Dokumenten. Aus diesem Grund werden Tags in Xpand durch das öffnende « und schließende » Guillemot vom restlichen Text separiert [TPB+07, S. 196][SVEH07, S. 153].

```
1  «IMPORT uml»
2  «EXTENSION extensions::Uml»
3
4  «DEFINE Root FOR Class»
5    «FILE this.getFilename()»
6      package «this.owningPackage.getQualifiedName()»;
7      public class «this.name»{
8        «EXPAND Property FOREACH this.properties»
9        «EXPAND Operation FOREACH this.operations»
10     }
11   «ENDFILE»
12 «ENDDEFINE»
```

Listing 1: Xpand Beispieltemplate UML2Java

Ein Template zur Generierung des Quelltexts einer Javaklasse aus einer UML-Klasse kann mit Xpand z. B. wie in Listing 1 dargestellt implementiert werden. Das Template besteht aus einleitenden Anweisungen, auf die ein DEFINE-Block zur eigentlichen Generierung folgt. Der Xpand-Tag *IMPORT* steuert den Import von Namensräumen verwendeter Metamodelle [EFH+08, S. 81], sodass z. B. im restlichen Template das Metamodellelement *uml::Class* mit dem unqualifizierten Namen *Class* ohne Verwendung des Paketpräfix *uml* referenziert werden kann. Durch den Xpand-Tag *EXTENSION* wird eine Xtend-Datei eingebunden, in der Funktionen für Metamodellelemente definiert werden können [EFH+08, S. 81]. Die Auslagerung von Verhaltenslogik von Metamodellelementen in Xtend-Dateien vereinfacht die Metamodelle, da die Logik als Teil des Generators und nicht als Teil des Metamodells definiert wird. Im Beispiel werden die Funktionen *getFilename()* und *getQualifiedName()* durch die Xtend-Datei *Uml.ext* im Verzeichnis *extensions* bereitgestellt. Die Kapselung von Generatorlogik durch Funktionen ermöglicht die Strukturierung und Wiederverwendung der Verhaltenslogik und schafft prägnantere Templates.

```
1  «DEFINE <Templatename>(<Parameterliste>) FOR <Metamodelelementname>»
2    <...>
3  «ENDDEFINE»
```

Listing 2: Xpand DEFINE-Block

DEFINE-Blöcke werden durch die Tags *DEFINE* und *ENDDEFINE* ein- und ausgeleitet. Sie haben einen Namen und referenzieren ein Metamodellelement als Typ (vgl. Listing 2). Optional kann eine kommaseparierte Liste von Parametern vorgegeben werden, auf deren Werte bei der Ausführung des Templates im Kontext des DEFINE-Block zugegriffen werden kann. Ein DEFINE-Block wird durch den Xpand-Tag *EXPAND* einmalig für ein Modellelement (*FOR*) oder mehrfach für mehrere Modellelemente (*FOREACH*) aufgerufen

(vgl. Listing 1). Dabei muss der Typ des jeweiligen Modellelements zu dem Metamodellelement bzw. Typ des DEFINE-Blocks passen. Jeder DEFINE-Block besitzt eine Variable mit dem Schlüsselwort-Namen *this*, die zur Ausführung des DEFINE-Blocks mit dem aufrufenden Modellelement belegt wird, sodass auf dessen Eigenschaften und Operationen zugegriffen werden kann. [EFH+08, S. 81 f.][TPB+07, S. 195][SVEH07, S. 153]

Die Zuordnung des DEFINE-Blocks zu einem Metamodellelement dient der statischen Typisierung der *this*-Variable und der Steuerung des Generators mittels Polymorphie i.V.m. spätem Binden. So können DEFINE-Blöcke mit derselben Signatur, bestehend aus dem Namen und der Parameterliste, für verschiedene Typen bzw. Metamodellelemente definiert werden. Zur Laufzeit wird dynamisch für ein Modellelement korrespondierend zu dessen Typ ein DEFINE-Block zur Ausführung ausgewählt. Die Polymorphie drückt sich darin aus, dass eine EReference sowohl eine EClass als auch deren Sub-EClasses referenzieren kann. Falls für einen Typ bzw. ein Metamodellelement kein DEFINE-Block existiert, der Typ aber Teil einer einfachvererbenden Typhierarchie bzw. einer mehrfachvererbenden Typstruktur ist, wird der DEFINE-Block des nächstliegenden Supertyps in der Hierarchie bzw. Struktur selektiert [TPB+07, S. 129]. Beispielsweise wird im Fall von Ecore eine solche Spezialisierungsbeziehung durch die EReference *eSuperTypes* ausgedrückt. Die automatisierte Selektion von DEFINE-Blöcken durch die Transformationsengine anhand von Typen erübrigt die explizite Selektion in Templates durch fallweise Überprüfung in if-instanceOf-Kaskaden [SVEH07, S. 308].

Diese Art der Polymorphie ähnelt der inklusionsbasierten Polymorphie, falls DEFINE-Blöcke als Operationen ihrer Typen bzw. Metamodellelemente verstanden werden. Inklusionsbasierte Polymorphie sieht vor, dass ein Typ im Kontext seiner Supertypen verwendet werden kann, also Operationen für diese Supertypen auch auf Subtypen angewendet werden können [CW85, S. 6].

Über die bereits genannten Schlüsselwörter hinaus bietet Xpand unter anderem folgende weitere (vgl. [TPB+07, S. 195 f.][EFH+08, S. 81-86]):

IF: Verzweigungen des Programmflusses werden durch Konditions-Ausdrücke gesteuert und können beliebig geschachtelt werden.

```
1 «IF <Kondition>» <...> [«ELSEIF <Kondition>» <...>] [«ELSE» <...>] «ENDIF»
```

FOREACH: Eine Schleife iteriert über die Elemente einer Collection. Zwischen dem öffnenden und dem schließenden Tag wird das Element an den angegebenen Variablennamen gebunden. Der optionale Iterator stellt Metainformationen über die Iteration bereit. Der optionale Separator wird bei jedem Übergang zwischen Iterationen in das Generat eingefügt.

```
1 «FOREACH <Ausdruck> AS <Varname> [ITERATOR <Iteratorname>] [SEPARATOR <Separator>]»
2   <...>
3 «ENDFOREACH»
```

FILE: Text, der innerhalb eines DEFINE-Blocks in die Tags *FILE* und *ENDFILE* eingefasst wird, wird zur Laufzeit in eine Ausgabedatei geschrieben. Im Anschluss an das *FILE*-Tag wird der Dateipfad im Dateisystem festgelegt, der relativ zu einem Wurzelverzeichnis interpretiert wird, das als Outlet bezeichnet wird. Für den Generator können mehrere solcher Outlets definiert werden, sodass von den Plattformspezifika unterschiedlicher Dateisysteme abstrahiert werden kann. Die Angabe eines Outlets ist optional, da ein generatorweites Standardoutlet vorgegeben ist.

```
1 «FILE "<Dateipfad>" [<Outletname>]» ... «ENDFILE»
```

PROTECT: Eine geschützte Region ist ein Textabschnitt im Artefakt, der bei einem Generatordurchlauf nicht modifiziert wird und üblicherweise der Erhaltung manueller Anpassungen des Artefakts dient. Im Artefakt wird der Bereich durch Markierungen gekennzeichnet, welche eine ID enthalten. Beispielsweise haben im Fall von Java die Markierungen folgende Form:

```
1 /*PROTECTED REGION ID(MODELS_CUSTOMER_GETNAME) ENABLED START*/
2 <...>
3 /*PROTECTED REGION END*/
```

Im Template wird die Region durch einen PROTECT-Block gekennzeichnet, der im Beispiel folgendermaßen strukturiert ist:

```
1 «PROTECT CSTART "/* " CEND " */" ID this.getId()»
2   <...>
3 «ENDPROTECT»
```

Bei einem Generatordurchlauf wird der Inhalt der geschützten Regionen vorab gesichert und nach der Generierung anhand der IDs wiederhergestellt. Der Wert der ID wird durch den Ausdruck hinter dem Schlüsselwort ID berechnet. Um Kollisionen mit der Syntax der Zielsprache zu vermeiden, werden die Markierungen üblicherweise in Kommentare eingefasst, deren Markierungen im Template mit CSTART und CEND definiert werden. Der Standardtext einer geschützten Region kann innerhalb des PROTECT-Blocks vorgegeben werden und wird bei der initialen Erstellung des Generats in die geschützte Region geschrieben. Falls im Template der Parameter DISABLE angegeben wird, wird die geschützte Region bei einem Generatordurchlauf mit dem Standardtext überschrieben.

```
1 «PROTECT CSTART <Ausdruck> CEND <Ausdruck> ID <Ausdruck> [DISABLE]» <...> «ENDPROTECT»
```

ERROR: Der Generatordurchlauf kann durch den ERROR-Block i.V.m. der Auslösung einer Exception abgebrochen werden.

```
1 «ERROR <Ausdruck>»
```

2.8 Modell-zu-Modell-Transformation

2.8.1 Grundlagen

Das MDSD und im Speziellen die MDA sieht neben der Transformation von Modellen zu Text ebenso die Transformation von Modellen zu Modellen vor. M2M-Transformationen erfüllen verschiedene Zwecke, zu denen die Zusammenführung einzelner Modelle, die Migration von Modellen zwischen unterschiedlichen Modellierungssprachen und Metamodellversionen sowie die Generierung von implementierungsnahen Modellen aus solchen mit einem höheren Abstraktionsgrad zählen. Der letztere Zweck ist insbesondere relevant für die MDA mit ihrer mehrstufigen Generierung von Systemen über intermediäre Modelle.

Wie für M2T-Transformationen (vgl. Kapitel 2.7) können auch bei der Formulierung bzw. Implementierung von M2M-Transformationsdefinitionen sowohl generische Programmiersprachen als auch spezielle Transformationssprachen zum Einsatz kommen. Vorteilhaft an Transformationssprachen ist die kompaktere Repräsentationsform der Abbildungsregeln durch entsprechende sprachliche Ausdrucksmittel und die Einbettung spezieller Sprachen für Abfragen auf Modellgraphen. Falls die Transformationssprache standardisiert ist, ermöglicht dies zudem die standardkonforme Implementierung der Transformationsdefinition. Darin ist ein Unterschied zur hartcodierten und somit inflexiblen Implementierung von Transformationsdefinitionen beim CASE zu sehen.

Eine Transformationssprache unterstützt ein oder im Ausnahmefall mehrere Meta-Metamodelle (M3), mit deren sprachlichen Ausdrucksmitteln die Metamodelle (M2) der Quell- und Zielmodelle (M1) zu formulieren sind (vgl. Abbildung 11). Trotz der Heterogenität der verschiedenen Modellierungssprachen bzw. Metamodelle haben diese somit ein einheitlich definiertes Meta-Metamodelle gemeinsam. Dies ermöglicht, dass Elemente verschiedener Metamodelle als Instanzen der Meta-Metamodellelemente in diversen Transformationsdefinitionen aufeinander abgebildet werden können, die sämtlich in einer gemeinsamen Transformationssprache formuliert sind. Im Rahmen der MDA wird die MOF als einheitliches Meta-Metamodell verwendet, sodass Transformationssprachen für die MDA im Minimalfall mit der MOF nur ein Meta-Metamodell referenzieren.

Für die Charakterisierung von Transformationssprachen existieren verschiedene Klassifikationsrahmen [CH06, MG06b, SK03, CH03, EMM10], von deren Kriterien folgende die markantesten sind:

Paradigma: Eine Transformationssprache folgt als Programmiersprache einem Programmierparadigma, das prozedural, objektorientiert, funktional, logisch, relational oder

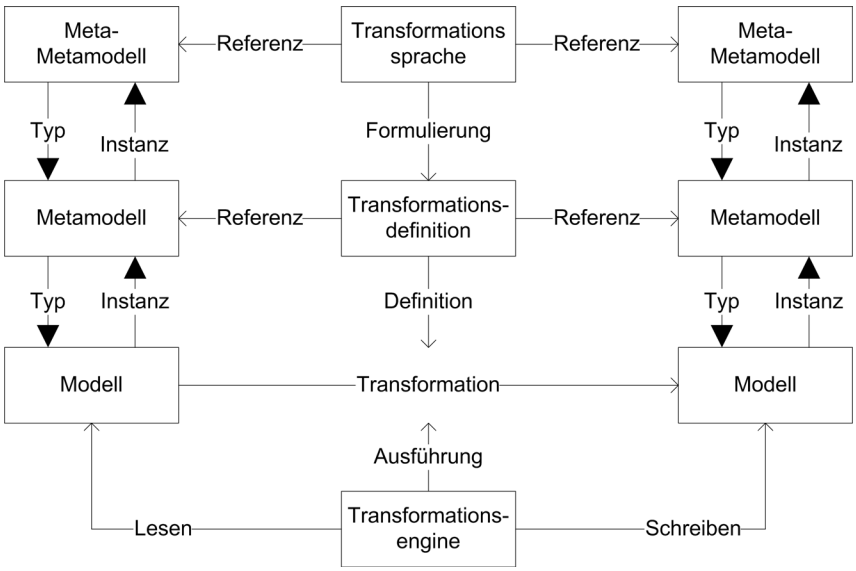

Abbildung 11: Metaebenen der Modell-zu-Modell-Transformation (vgl. [CH06, S. 623])

eine hybride Mischung aus diesen sein kann. Das Paradigma der Sprache determiniert den Charakter der Transformationsdefinitionen, also ob die Transformationsregeln deklarativ oder imperativ bzw. operational formuliert sind.

Technische Sphäre: Das Meta-Metamodell gibt die technische Sphäre bzw. den Rahmen vor, in dem Modellierungssprachen durch Metamodelle spezifiziert werden. Eine Transformationssprache bezieht sich auf diesen Rahmen, indem Elemente des Meta-Metamodells als Typen bei der Implementierung der Transformationssprache referenziert werden. Die Wahl des Meta-Metamodells schränkt somit die Menge der nutzbaren Transformationssprachen ein.

Direktionalität: Eine unidirektionale Transformationsdefinition unterstützt ausschließlich die Transformation von Modellen eines Metamodells A zu Modellen eines Metamodells B. Eine bidirektionale Transformationsdefinition ermöglicht dagegen die Transformation von Modellen in beide Richtungen. Abhängig von der Transformationssprache müssen dazu entweder entgegengesetzte unidirektionale oder einzelne bidirektionale Transformationsregeln definiert werden. Einen Überblick über bidirektionale Transformationssprachen bietet [Ste08].

Während bei den im MDSD verbreiteten templatebasierten M2T-Transformationsdefinitionen die Syntax des Generats dem Generator üblicherweise nicht bekannt ist, werden bei M2M-Transformationen die syntaktischen Vorgaben und die Typen für das Zielmodell durch das Zielmetamodell spezifiziert. Dies bietet abhängig von der Konzeption der Trans-

formationssprache den Vorteil, dass bereits vor der Transformation bzw. vor der Ausführung der Transformationsdefinition eine Überprüfung auf eine typsichere und syntaxkonforme Abbildung der Elemente des Quellmetamodells auf Elemente des Zielmetamodells möglich ist.

2.8.2 Query/View/Transformation

Query/View/Transformation (QVT) ist ein Standard der OMG, der erstmals im Jahr 2008 in einer formalen Version veröffentlicht wurde [Obj08a] und die drei Transformationssprachen *QVT Relations*, *QVT Operational Mappings* und *QVT Core* für M2M-Transformationen spezifiziert [Obj11b]. Mit QVT können MOF-basierte Metamodelle (M2) auf imperative und deklarative Weisen in Beziehung zueinander gesetzt werden, um deren MOF-basierte Modelle (M1) zu transformieren. Damit fügt sich QVT in die MDA-Vision ein, um als Werkzeug die Transformation zwischen CIM, PIM und PSM zu ermöglichen.

Die Name QVT spiegelt die drei Funktionsbereiche *Query*, *View* und *Transformation* wider. Der Bereich *Query* behandelt Abfragen auf Modellen zur Selektion einzelner Modellelemente. Der Bereich *View* befasst sich mit Sichten als Abfragen zusammenhängender Muster von Modellelementen. Der Bereich *Transformation* deckt schließlich unter Verwendung der beiden vorigen Bereiche die Transformation von Modellen ab.

Mit den drei Sprachen *QVT Core*, *QVT Relations* und *QVT Operational Mappings* deckt die Spezifikation in hybrider Form sowohl das deklarative als auch das imperative Programmierparadigma ab und verbindet diese [Obj11b, S. 9 f.]:

- QVT Relations (QVTR) ist eine deskriptive Transformationssprache, die eine Transformation als eine Menge von Relationen spezifiziert. Da diese Relationen als Beziehungen zwischen Mustern von Metamodellelementen ungerichtet sind, können Transformationen mit QVTR einerseits bidirektional als Konsistenzprüfung (check) und andererseits unidirektional als Konsistenzherstellung (enforce) ausgeführt werden. [Obj11b, S. 13-62].

- QVT Core (QVTC) ist eine deklarative Transformationssprache mit geringem Sprachumfang, die in der Gesamtarchitektur von QVT die Rolle einer gemeinsamen Ausführungsplattform für die auf ihr basierenden Sprachen QVTO und QVTR einnimmt. Transformationsdefinitionen, die in QVTR verfasst sind, können mittels der *Relations-To-Core-Transformation* [Obj11b, S. 163-200] in QVTC überführt werden. Aufgrund des im Vergleich zu QVTR geringeren Sprachumfangs sind die Transformationsregeln weniger kompakt, sodass QVTC für die Entwicklung von Transformationsdefinitionen nur eine indirekte und untergeordnete Rolle spielt. [Obj11b, S. 145-162].

- QVT Operational Mappings (QVTO) ist eine imperative Transformationssprache, mit der einerseits mittels QVTR oder QVTC verfasste Transformationsdefinitionen um imperative Anweisungen angereichert und andererseits eigenständige Transformationsdefinitionen implementiert werden können. Letztere sind unidirektional und werden als operational bezeichnet. [Obj11b, S. 63-144]

Die Anwendung der in der QVT-Spezifikation beschriebenen Transformationssprachen für die Zwecke dieser Arbeit ist durch die derzeitige Verfügbarkeit an Implementierungen restringiert. Das Eclipse M2M Project implementiert aus der QVT-Spezifikation bislang in ausgereifter Form lediglich QVTO. Für QVTR existieren mit ModelMorf[6] und medini QVT[7] anderweitige Implementierungen, deren Verbreitung aber begrenzt ist und die zudem nicht auf der in der Spezifikation vorgesehenen Transformation zu QVT Core basieren. Da bei prototypischen Tests von M2M-Transformationen komplexerer PIMs zu PSMs mit QVTR die Transformationen entweder nicht terminierten oder eine hohe Zeit- und Platzkomplexität aufwiesen, konzentriert sich die Arbeit im weiteren Verlauf auf QVTO. Zudem werden im Folgenden die Transformationsdefinitionen vom CIM über das PIM zum PSM unidirektional konzipiert, sodass die unidirektionale Funktionalität von QVTO geeignet ist.

2.8.3 QVT Operational Mappings

Im Folgenden wird basierend auf der Spezifikation [Obj11b, S. 63-144] auf einen Teilausschnitt von QVTO eingegangen, der für die Zwecke der Arbeit relevant ist. Weiterführende Erläuterungen zu Details von QVTO finden sich bedarfsweise in nachfolgenden Kapiteln, welche auf die M2M-Transformationsdefinitionen eingehen, die im Rahmen dieser Arbeit vorgestellt werden.

Eine operationale Transformationsdefinition bzw. in der Terminologie der OMG eine operationale Transformation wird in QVTO durch unidirektionale Mapping-Operationen implementiert, welche Teil der umgebenden Transformation sind. Diese kann mit einer QVTO-kompatiblen Transformationsengine ausgeführt und damit instanziiert werden. Zu diesem Zweck werden in der Signatur der Transformation die Modelltypen angegeben, die als Parameter zu übergeben sind. Die Schlüsselwörter *in*, *inout* und *out* kennzeichnen, ob der Zugriff auf die übergebenen Modelle lesend oder schreibend ist. Z. B. im Fall von Listing 3 geben die Modelltypen ein UML-Modell als Eingabe und ein ER-Modell als Ausgabe vor. Jede Transformation enthält als Einstiegspunkt eine Main-Operation, die bei der

[6] `http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`
[7] `http://projects.ikv.de/qvt/`

Instanziierung der Transformation automatisch ausgeführt wird. Falls die Transformationsdatei nur eine einzelne Transformation enthält, können die Main-Operation sowie die weiteren Operationen und Mappings alternativ außerhalb des Rumpfs der Transformation implementiert werden (vgl. Listings 3 und 4).

```
1 transformation uml2erm(in umlModel : UML, out erModel : ERM){
2   main() {...}
3 }
```

Listing 3: QVTO-Transformation in geklammerter Form

```
1 transformation uml2erm(in umlModel : UML, out erModel : ERM);
2 main() {...}
```

Listing 4: QVTO-Transformation in nicht geklammerter Form

Ein Metamodell wird für eine Transformation eingebunden, indem es als *modeltype* am Anfang einer Transformationsdatei vor der Transformation deklariert wird (vgl. Listing 5). Die im Metamodell enthaltenen Typen können anschließend durch die Transformation bzw. deren Bestandteile typsicher referenziert werden.

Transformationsregeln werden durch Mapping-Operationen definiert, die ein oder mehrere Quell-Modellelemente auf ein oder mehrere Ziel-Modellelemente abbilden. Jede Mapping-Operation ist einem Metamodellelement bzw. einer Metaklasse zugeordnet und verhält sich wie eine native Operation der Metaklasse, wie z. B. im Fall der Mapping-Operation *toSchema* für die Metaklasse *UML::Package* (vgl. Listing 5). Bei der Ausführung der Mapping-Operation wird die Metaklasse mit der Variable *self* referenziert, sodass auf die Operationen und Attribute der Metaklasse zugegriffen werden kann. Durch den Zuweisungsoperator *:=* werden Attributen des Zielmodellelements Werte zugewiesen. Das Modellelement wird auf der linken Seite einer Zuweisungsoperation nicht explizit referenziert, da es implizit als Instanz des Zieltyps referenziert wird. So wird z. B. bei der Ausführung der Mapping-Operation *toSchema* implizit eine Instanz des Metamodellelements *ERM::Schema* referenziert, deren Attribute aufgrund der Zuweisungsoperationen mit Werten belegt werden. Die Instanz wird entweder bei erstmaliger Ausführung der Mapping-Operation auf einem bestimmten Quellmodellelement neu erzeugt oder bei wiederholter Ausführung erneut referenziert. Letzteres ermöglicht die Re-Referenzierung von Instanzen, die bereits durch andere Mapping-Operationen erzeugt wurden, ohne Referenzen zwischen Mapping-Operationen übergeben oder an zentraler Stelle vorhalten zu müssen (Details dazu unter [Obj11b, S. 66 f.]).

Eine Mapping-Operation wird durch den Aufruf *map* initiiert und gibt abhängig vom Rückgabetyp eine oder mehrere Modellelemente zurück. Alternativ zu der impliziten Gene-

```
1  modeltype UML uses 'uml';
2  modeltype ERM uses 'erm';
3
4  transformation uml2erm(in umlModel : UML, out erModel : ERM);
5
6  main() {
7    umlModel.rootObjects()[UML::Package]->map toSchema();
8  }
9
10 mapping UML::Package::toSchema() : ERM::Schema{
11   name := self.name;
12   ownedElements := self.getUmlClasses().map toEntityType();
13   ownedElements += self.getUmlAssociations().map toRelationshipType();
14 }
```

Listing 5: QVTO-Transformation UML2ERM

rierung eines Modellelements als Instanz des Rückgabetyps einer Mapping-Operation kann das Element durch das Schlüsselwort *object* explizit *inline* erzeugt werden (vgl. Listing 6). Der Vorteil von Inline-Ausdrücken liegt in der knappen Darstellungsform, die Implementierung als Mapping-Operation dagegen ermöglicht die Wiederverwendung.

```
1  mapping UML::Class::toEntityType() : ERM::EntityType{
2    name := self.name;
3    attributes := self.getPrimitiveProperties().map toAttribute();
4    attributes += object ERM::Attribute{ name := 'key'; };
5  }
```

Listing 6: QVTO-Transformation UML2ERM: Inline Mapping-Operation

Helper sind Operationen, die im Unterschied zu Mapping-Operationen nicht implizit Modellelemente erzeugen, sondern lediglich Berechnungen durchführen und als Nebeneffekt Modellelemente modifizieren können. Ein Helper ohne Nebeneffekte wird als Query bezeichnet und kann sowohl auf Basisdatentypen als auch auf Modellelementen aufgerufen werden (vgl. Listing 7).

```
1  query String::firstToLower() : String {
2    return self.substring(1,1).toLower() + self.substring(2, self.length());
3  }
4
5  query UML::Class::getPrivateProperties() : OrderedSet(UML::Property) {
6    return self.properties->select(e|e.visibility = UML::VisibilityKind::private);
7  }
```

Listing 7: QVTO-Transformation UML2ERM: Queries

3 Model Driven Web Engineering

3.1 Einordnung

Mit der Verbreitung des World Wide Web (WWW) spielt sich die Ausführung von Software zunehmend im Web-Kontext ab. So decken Web-basierte Applikationen bzw. Webapplikationen als Pendants zu klassischen Desktop-Anwendungen vergleichbare Funktionalität ab. Die Entwicklung dieser Webapplikationen wird durch Innovationen bei den zugrunde liegenden Web-Technologien ermöglicht. Anfängliche Web-Präsentationen mit rein statischen HTML-Seiten wurden durch JavaScript und Java-Applets um Client-seitige Interaktivität erweitert. Skriptsprachen wie PHP und Perl fügten die serverseitige Generierung von Webseiten mit dynamischen und personalisierten Inhalten hinzu. Die Integration in bestehende Anwendungssysteme wurde schließlich durch Middleware-Techniken wie Java EE ermöglicht, bei denen Webtechniken für Präsentationsfunktionalitäten und die Anbindung von Webservices genutzt werden.

Der neue Verwendungskontext führte zum Begriff der Webapplikation, welche ein Softwaresystem basierend auf Techniken und Standards des World Wide Web Consortiums (W3C) ist, das Web-spezifische Ressourcen wie z. B. Inhalte und Dienste über den Webbrowser als Benutzerschnittstelle bereitstellt[8] [KPRR06a, S. 2]. Als Softwaresystem mit beliebig komplexer Geschäftslogik geht eine Webapplikation somit über die reine informierende Darstellungsfunktionalität vieler Webseiten hinaus (vgl. [MDHG01, S. 5 f.]), hat mit ihnen aber den Einsatz von Webstandards gemein.

Als aktuelle Entwicklung halten unter dem Begriff der Rich Internet Application (RIA) Techniken wie Asynchronous JavaScript and XML (AJAX) Einzug in die Entwicklung von Webapplikationen, welche interaktive Formen von Benutzeroberflächen durch das dynamische Nachladen von Inhalten ermöglichen und so eine Ähnlichkeit zu klassischen Desktopanwendungen schaffen [Gar05]. RIA-Webtechniken finden zudem Verwendung im Distributionsmodell der Software as a Service (SaaS), bei dem Software als Dienstleistung mittels Internet- und Webtechniken bereitgestellt wird. Mit RIA-Laufzeitumgebungen wie Adobe AIR, Microsoft Silverlight, JavaFX und Google Chrome OS können Webapplikationen im weitesten Sinne schließlich wie klassische Applikationen auf dem Desktop eingebunden werden.

Ein Vorteil von Webapplikationen ist, dass der Webbrowser als Abstraktionsschicht die Webapplikation vom lokalen System entkoppelt. Die Trennung ermöglicht eine betriebssys-

[8] Im Original: „A Web application is a software system based on technologies and standards of the World Wide Web Consortium (W3C) that provides Web specific resources such as content and services through a user interface, the Web browser.“

temunabhängige Entwicklung. Die dazu einzuhaltenden Webstandards restringieren zwar einerseits die Entwicklung, führen aber andererseits zu technisch standardisierten Benutzerschnittstellen. Zudem wird der Aufwand für die Softwareverteilung minimiert und der Zugriff auf die Webapplikation ohne initiale lokale Installation ermöglicht, da im Unterschied zu Desktopanwendungen die Auslieferung des Codes der Benutzerschnittstelle nicht in Form lokaler Installationen stattfindet, sondern durch den Webbrowser, der automatisch die aktuelle Version lädt und potentiell mittels Cachingmechanismen lokal zwischenspeichert.

Komplixitätssteigerungen ergeben sich dagegen bei der Entwicklung, falls Inkompatibilitäten verschiedener Webbrowsertypen und -versionen zu beachten sind. Der damit verbundene Aufwand lässt sich aber begrenzen, falls Einschränkungen bzgl. der adressierten Webbrowser vorgenommen werden. Einen weiteren Nachteil können Restriktionen für den Zugriff auf native Programmierschnittstellen des Betriebssystems bilden, die aus der abstrahierenden Funktion des Webbrowsers folgen. Diese werden aber potentiell durch den erweiterten Funktionsumfang des HTML5-Standards kompensiert. Ein zusätzlicher Nachteil von Webapplikationen im Vergleich zu Desktopanwendungen ist, dass die Implementierung der Benutzerschnittstelle durch Webseiten zu Ladezeiten bei Seitenwechseln führt, falls auf asynchrones Nachladen von Inhalten durch komplexere Lösungsansätze wie AJAX verzichtet wird. Eine asynchrone Kommunikation zwischen Webbrowser und Webserver wird ebenso benötigt, um bei eingeschränkter Konnektivität des Webbrowsers den Nachteil der Nichtverfügbarkeit der Webapplikation zu lösen. Mit HTML5 werden zu diesem Zweck ebenfalls standardisierte Lösungsansätze vorangetrieben.

Der zunehmende Funktionsumfang von Webapplikationen z. B. in Form von RIA-Präsentationstechniken oder der Anbindung an Middlewaresysteme erhöht deren Komplexität, die vergleichbar mit derjenigen klassischer Softwaresysteme ist. Um geeignet auf die Softwarekrise (vgl. Kapitel 2.2) bzw. Webkrise [Zel98][GM01, S. 14 f.] zu reagieren, bedarf es deshalb für die Entwicklung von Webapplikationen ingenieurmäßig-systematischer Methoden, um den Konstruktionssprozess plan- und wiederholbar zu gestalten. Ein solches methodisches Vorgehen soll das Web Engineering ermöglichen, das seit dem Jahr 1998 als wissenschaftliche Disziplin propagiert wird [MDHG01, S. 4][GM01, MD99, Gin98, S. 16] und das Software Engineering mit den Spezifika der Entwicklung Web-basierter Systeme verbindet. Dabei werden Vorgehensmodelle und Modellierungssprachen genutzt, um den künstlerischen Ad-hoc-Schaffungsprozess zu steuern und die zunehmende Komplexität der Artefakte zu kontrollieren.

Murugesan et al. definieren Web Engineering als die Etablierung und Anwendung fundierter Prinzipien aus Wissenschaft, Ingenieurwesen und Management sowie fachlicher und

systematischer Herangehensweisen für die erfolgreiche Entwicklung, Bereitstellung und Wartung hochqualitativer Web-basierter Systeme und Applikationen[9] [MDHG01, S. 6]. Damit werden bereits existierende Prinzipien und Ansätze des Software Engineering auf Spezifika Web-basierter Systeme gemünzt.

Kappel et al. sehen das Web Engineering als die Anwendung systematischer und quantifizierbarer Herangehensweisen für die kosteneffiziente Umsetzung von Anforderungsanalyse, Entwurf, Implementierung, Testen, Einsatz und Wartung hoch-qualitativer Webapplikationen sowie als die wissenschaftliche Disziplin, die sich mit diesen Herangehensweisen befasst[10] [KPRR06a, S. 3]. Dies betont die Differenzierung zwischen der wissenschaftlichen Disziplin und der praktischen Anwendung vor dem Hintergrund von Vorgehensmodellen aus dem Software Engineering.

Im Vergleich von Software Engineering und Web Engineering fallen sowohl Unterschiede bzgl. der Artefakte als auch der Entwicklungsmethoden auf:

Die Artefakte Web-basierter Systeme unterscheiden sich aus konzeptioneller Sicht von herkömmlicher Software durch den hypertextuellen Charakter, die Flexibilität bzgl. der Präsentation und die Dokumentenorientierung [KPRR06a, S. 10-12]. Technisch betrachtet fallen Unterschiede in Bezug auf verwendete Programmiersprachen und Frameworks auf. Mit der Tendenz zu interpretierten Skriptsprachen ist die Verbreitung von Webframeworks verbunden, wie z. B. JavaServer Faces (JSF)[11], Ruby on Rails (RoR)[12] oder das Zend Framework[13], die Funktionalität zum Umgang mit Webtechniken wie HTTP, HTML oder AJAX bieten.

Methodisch unterscheidet sich das Web Engineering sowohl bzgl. des Vorgehens als auch der verwendeten Modellierungssprachen. Das Vorgehen ist üblicherweise geprägt durch kurze Auslieferungszyklen, kontinuierliche Anpassungen [Gin01][KPRR06a, S. 17] und somit häufige Aktualisierungen [KMP+05, S. 5]. Abhängig vom Applikationstyp liegt zudem eine Mischung aus Softwareentwicklung und Publizieren vor [GM01, S. 16], wobei redaktionelle Tätigkeiten mittlerweile aufgrund entsprechender Softwaresysteme nicht mehr zum Kern der Entwicklungstätigkeit zählen. Durch den Ursprung des WWW als Informationsmedi-

[9] Im Original: „Web engineering is the 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."

[10] Im Original: „1) Web Engineering is the application of systematic and quantifiable approaches (concepts, methods, techniques, tools) to cost-effective requirements analysis, design, implementation, testing, operation, and maintenance of high-quality Web applications. 2) Web Engineering is also the scientific discipline concerned with the study of these approaches."

[11] `http://java.sun.com/j2ee/javaserverfaces/`

[12] `http://www.rubyonrails.org/`

[13] `http://framework.zend.com/`

um mit erweiterten künstlerisch-grafischen Gestaltungsmöglichkeiten ist der Entwicklerkreis tendenziell heterogen und interdisziplinär [MDHG01, S. 7][Gre04, S. 16][KMP+05, S. 5][KPRR06a, S. 14 f.], was durch Abstraktionsmechanismen zur Trennung von Design und Code systematisiert wird.

Für das Web Engineering existiert eine Vielzahl an Methoden und Techniken (vgl. für einen Überblick [KPRR06b, Küh01]), welche verschiedene Aufgabenbereiche wie z. B. die Anforderungsanalyse, Modellierung, Benutzerfreundlichkeit, Wartung und das Testen adressieren. Prägend für die Ansätze ist, dass der hypertextuelle Charakter von Webseiten und Webanwendungen durch Navigationsmodelle verschiedener Formen ausgedrückt wird. Mit der zunehmenden serverseitigen Generierung von Webseiten ab dem Ende der 1990er Jahre verschob sich auch bei den Modellierungssprachen der Fokus von Modellen über Hypertexte und Dokumentstrukturen zu Modellen über Softwaresysteme, welche mit dem Benutzer interagieren und Hypertexte als Ausgabe generieren. Entsprechend finden sich in den Modellen sowohl Spezifikationen des Hypertexts bzw. der Navigationsstruktur als auch der zugrunde liegenden Programmlogik und Datenstrukturen wieder.

Als ein spezielles Forschungsfeld aus der Verbindung von Web Engineering und MDSD ist das Model Driven Web Engineering (MDWE) hervorgegangen, das auf die modellgetriebene Entwicklung von Webapplikationen und Web-basierten Systemen im Allgemeinen abzielt. Für das MDWE existiert eine Vielzahl an Ansätzen mit unterschiedlich ausgeprägtem Bezug zum MDSD und zur MDA. Abhängig vom Alter und der Ausrichtung sind diese entweder Modellierungsmethoden ohne ursprünglichen Bezug zum MDSD, die nachträglich um MDSD-Fähigkeiten erweitert wurden, oder aber solche, die originär modellgetrieben konzipiert sind.

Da diese Arbeit im MDWE verortet ist, wird im Folgenden auf eine Auswahl der prominenten und primär modellgetriebenen Ansätze für das Web Engineering eingegangen. Die Charakteristika älterer Modellierungssprachen und -methoden spiegeln sich in den neueren Ansätzen aufgrund der stetigen Weiterentwicklungen und gegenseitigen Bezugnahme wider [SK06, S. 58 ff.].

3.2 Web Application Extension for UML (WAE)

Die Web Application Extension for UML (WAE) ist eine Modellierungssprache, welche die UML um Sprachelemente zur Modellierung von Webapplikationen erweitert. WAE wurde erstmals im Jahr 1999 vorgestellt [Con99a] und wie auch die UML im Rahmen der *Rational Software Corporation* entwickelt (vgl. Kapitel 2.2). Zur Definition des Metamodells von WAE wurden die zu dieser Zeit verfügbaren Erweiterungsfähigkeiten der UML verwendet [Rat97, S. 14-21], um die vordefinierte Semantik der UML-Sprachelemente für den

Webkontext umzudeuten. Das Metamodell von WAE besteht aus Stereotypen, die Metaattribute in Form von Tagged Values enthalten und ausgewählte Sprachelemente der UML spezialisieren.

Die WAE sieht Klassendiagramme zur Modellierung von Webseiten vor, welche die möglichen Navigationspfade der Benutzer in der Webapplikation beschreiben. Klassen repräsentieren in abstrahierter Form Webseiten oder im weitesten Sinne über HTTP abrufbare Inhalte, die in Form eines Navigationsgraphen bzw. einer Site Map in Verbindung zueinander stehen. Von der Implementierung der Klassen z. B. durch Servlets, CGI Module oder JavaServer Pages und den damit verbundenen Programmiersprachen wird im Modell abstrahiert. Die Logik zur serverseitigen Erzeugung bzw. Ausgabe von Webseiten sowie zu deren clientseitigen Modifikation wird durch Operationen der Klassen repräsentiert, wobei der Fokus auf der Geschäftslogik und nicht auf der Präsentationslogik liegt. [Con99b, S. 65 f.]

Zur Unterscheidung server- und clientseitiger Logik werden die Klassen durch die Stereotypen *«server page»* und *«client page»* in abstrakter Form als Webseiten gekennzeichnet. *ServerPages* repräsentieren beispielsweise Skripte oder Programme, welche auf dem Server ausgeführt werden und als Ausgabe Webseiten in Form von HTML-Code erzeugen, die ihrerseits im Modell als *ClientPages* dargestellt werden (vgl. Abbildung 12). Die Erzeuger-Beziehung zwischen einer *Server-* und einer *ClientPage* wird im Modell durch eine Assoziation mit dem Stereotyp *«build»* ausgedrückt. Navigationspfade durch das System werden durch Hyperlinks zwischen Webseiten gebildet, die im Modell jeweils durch eine Assoziation mit dem Stereotyp *«link»* zwischen zwei *ClientPages* oder zwischen einer *ClientPage* und einer *ServerPage* dargestellt wird. Im letzteren Fall wird über die *ServerPage* auch implizit die durch diese generierte *ClientPage* referenziert. Eine *«link»*-Assoziation kann eine n-wertige Kardinalität besitzen, um auszudrücken, dass eine *ServerPage* abhängig von den Aufrufparametern und serverseitigen Daten unterschiedliche *ClientPages* erzeugt. [Con99b, S. 68]

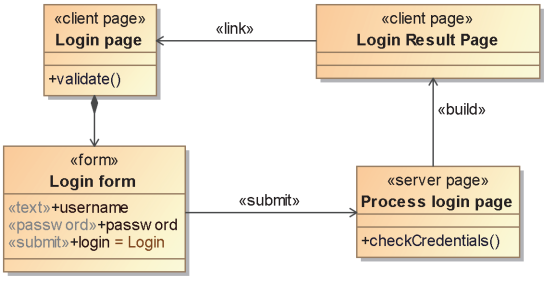

Abbildung 12: WAE-Designmodell

Die Logik einer *Server-* bzw. *ClientPage* wird durch Operationen und Attribute der entsprechenden Klasse in detaillierterer Form modelliert. Bei einer *ServerPage* bilden die Operationen z. B. Prozeduren oder Methoden zur Erzeugung von *ClientPages* oder allgemein zur Verarbeitung serverseitiger Daten ab. Im Kontext einer *ClientPage* repräsentieren Operationen dagegen z. B. JavaScript-Funktionen, also Programmlogik, die im Webbrowser ausgeführt wird. Variablen, die global im Kontext einer Server- oder Clientseite sind, werden als Attribute der entsprechenden Klassen dargestellt. [Con99b, S. 69]

Ein HTML-Formular wird im Modell durch eine Klasse mit dem Stereotyp *«form»* repräsentiert, deren Attribute die Eingabefelder des Formulars abbilden. Die Attribute werden durch Stereotypen wie *«Text»*, *«Button»* und *«CheckBox»* typisiert, sodass unterschiedliche Arten von Eingabefeldtypen von HTML-Formularen modelliert werden können. Jede *«form»*-Klasse steht in einer UML-Kompositionsbeziehung zu einer *ClientPage*, wodurch ausgedrückt wird, dass das Formular Teil einer HTML-Webseite ist. Darüber hinaus steht jede *«form»*-Klasse mit einer Assoziation in Beziehung zu einer *ServerPage*, welche die Eingabe verarbeitet. Die Assoziation wird durch den Stereotyp *«submit»* gekennzeichnet. Über die genannten Stereotypen hinaus sind in WAE weitere Stereotypen zur Modellierung von HTML-Frames vorgesehen. [Con99b, S. 69 f.]

Die Fortentwicklung WAE2 erweitert die Modellierungssprache WAE um ein Vorgehensmodell sowie um weitere Modelltypen und Sprachelemente zu einer Methode. Das Vorgehensmodell umfasst die Phasen der Anforderungsanalyse, der Analyse des Nutzungserlebnisses (User Experience), der Systemanalyse, des Entwurfs und der Implementierung. In den Phasen werden vielzählige UML-Diagramme basierend auf den Modelltypen und Stereotypen von WAE2 erstellt. In der Anforderungsanalyse werden die funktionalen und nichtfunktionalen Anforderungen an die Webapplikation primär mittels Use-Case-Diagrammen sowie durch textuelle Dokumente, Aktivitäts- und Sequenzdiagramme erfasst. Die bildliche Darstellung der Webapplikation gegenüber dem Benutzer wird abstrakt durch das User Experience (UX) Model erfasst, das aus Klassendiagrammen zur Modellierung der gerenderten Ansichten (Screens) der Benutzerschnittstelle, aus Sequenzdiagrammen zur Modellierung des Verhaltens der Benutzerschnittstelle und Aktivitäts-, Sequenz- oder Kollaborationsdiagrammen zur Darstellung von Abfolgen von Screens besteht. Im Analysemodell wird das Verhalten der Webapplikation entsprechend der ermittelten Anforderungen mittels Klassen-, Paket-, Anwendungsfall-, Aktivitäts-, Sequenz- und Kollaborationsdiagrammen modelliert, welche zur Überführung in das Designmodell um Implementierungsdetails und Komponentendiagramme zur dateiorientierten Darstellung angereichert werden. Die Elemente aus WAE werden für das Designmodell von WAE2 wiederverwendet und um zusätzliche Stereotypen z. B. für JavaServer Pages ergänzt. [Con02]

Im Vergleich zu reinen UML-Klassendiagrammen weisen Klassendiagramme mit WAE-Stereotypen bei der Modellierung von Webapplikationen eine höhere Verständlichkeit für ungeübte Entwickler auf, da die Navigationsstruktur, die serverseitige Generierung von Webseiten und die Einbindung von Formularen expliziert werden [RPT+06, S. 7].

Die Umdeutung der Semantik der Sprachelemente *Class* und *Association* für die Zwecke der Modellierung Web-basierter Systeme erscheint zunächst kontraintuitiv, da UML-Klassendiagramme üblicherweise bei der Modellierung objektorientierter Zusammenhänge Verwendung finden. Im Hinblick auf die UML-Spezifikation deckt die flexible Semantik des Metamodellelements *Class* aber auch grob die Semantik von Webseiten ab[14]. Dasselbe gilt für das Metamodellelement *Association* bzw. dessen in WAE verwendete Spezialisierung *Binary Association*[15].

Die Verwendung von Stereotypen und Tagged Values als Mechanismus zur Metamodellierung führt zu dem Problem, dass die Syntax der zugrundeliegenden UML zwar erweitert, aber nicht durch Constraints eingeschränkt wird. Infolgedessen muss dem Modellierer die Syntax komplett bekannt sein, damit er sie freiwillig bei der Modellierung einhält, um valide Modelle zu erstellen. Die Einschränkung der Syntax wäre zwar über OCL-Ausdrücke möglich, die in der WAE-Spezifikation aber nicht enthalten sind.

Der Einsatz von WAE über rein beschreibende Modelle hinaus für die generativen Zwecke des MDSD scheitert an der Ermangelung eines Codegenerators. Es existieren zwar Beschreibungen, wie die Elemente nach HTML und JSP zu transformieren sind (vgl. [Con02, Appendix A]), diese sind aber konzeptionell formuliert und können in der gegebenen Form nicht automatisiert ausgeführt werden. Zusätzlich sind die Elemente des Designmodells nicht ausreichend typisiert, um eine weitestgehend vollständige Generierung der Verhaltenslogik über rein statische Implementierungsartefakte hinaus zu ermöglichen. WAE2 ist mit dem Vorgehensmodell und der Modellierungssprache somit zwar eine ganzheitliche Methode, M2M- und M2T-Transformationsdefinitionen zur Automatisierung fehlen aber, sodass eine durchgängige Generierung weder über die verschiedenen Phasen noch im letzten Schritt vom Designmodell zum Quelltext möglich ist.

Nachteilig ist zudem, dass eine präzise Abbildung von Plattformspezifika und Präsentationsdetails in WAE-Modellen nicht möglich ist. Plattformdetails wie z. B. Servletmethoden werden lediglich mit ihrer Signatur und ohne weitergehende Typisierung erfasst, die für eine anschließende Generierung relevant wäre. Dies wird damit begründet, dass WAE auf die Modellierung von Geschäftslogik fokussiert und von technischen Aspekten sowie

[14] „A class represents a concept within the system being modeled. Classes have data structure and behavior and relationships to other elements.“ [Rat97, S. 23]

[15] „A binary association is an association among exactly two classes (including the possibility of a reflexive association from a class to itself).“ [Rat97, S. 50]

Präsentationsaspekten abstrahieren soll [Con99b, S. 64].

3.3 Web Modeling Language (WebML)

Die Web Modeling Language (WebML) ist eine Methode zur Entwicklung daten- und prozessorientierter Webapplikationen, die eine gleichnamige domänenspezifische Modellierungssprache umfasst und seit dem Jahr 2000 [CFB00] entwickelt wird. Die initiale Version fokussierte auf die plattformunabhängige Beschreibung datenorientierter Webapplikationen als System interagierender Seiten und Dateneinheiten. Im Laufe der Evolution der Sprache wurden zusätzliche sprachliche Ausdrucksmittel zur Modellierung prozess- und serviceorientierter Webapplikationen hinzugefügt (vgl. [BCFM07, S. 222]). Charakteristisch für WebML ist, dass Webapplikationen in den Modellen mit einer geringen Anzahl aggregierter und abstrakter Konzepte spezifiziert werden [BCFM07, S. 221]. In der Terminologie der MDA kann WebML deshalb der PIM-Ebene zugeordnet werden, da einerseits von Plattformdetails abstrahiert wird, andererseits aber das Lösungssystem in Form von abstrakten Komponenten einer Webapplikation beschrieben wird. WebML sieht zur Modellierung mehrere Modelltypen für die unterschiedlichen Aspekte von Webapplikationen vor [BCFM07, CFB+02, CFB00, BCFM00], von denen im Folgenden die wesentlichen vorgestellt werden:

Das Datenmodell der Webapplikation wird in WebML als *Data Model* [CFB+02, S. 61-76] oder *Structural Model* [CFB00, BCFM00] bezeichnet. Dazu ist keine eigene Modellierungssprache vorgesehen, sondern es werden die etablierten Modellierungssprachen ERM und UML verwendet.

Das *Hypertext Model* beschreibt den Hypertext der Webapplikation in Form einzelner Seiten (*Pages*), welche durch *Links* miteinander verbunden werden (vgl. Abbildung 13). Eine *Page* wird durch eine oder mehrere *Content Units* (Kurzform *Units*) genauer spezifiziert, welche in aggregierter Form die Funktionalität oder die Daten einer Seite repräsentieren. WebML sieht dazu verschiedene *Unit*-Typen vor. Die anfänglichen Versionen von WebML enthielten zudem eine Unterteilung des Hypertext Models in das *Composition Model* und das *Navigation Model* zur getrennten Modellierung von *Pages* i.V.m. *Units* und *Links* [CFB00, BCFM00].

Grundlegende *Content Unit*-Typen sind die *Data Unit* zur Darstellung eines Datenobjekts, die *Multidata Unit* zur Darstellung einer Menge von Datenobjekten, die *Index Unit* zur Listendarstellung von Datenobjekten, die *Scroller Unit* zur Navigation in einer Liste von Datenobjekten [CFB+02, S. 79-89] und in anfänglichen Versionen von WebML die *Filter Unit* für Suchmasken auf einer Menge von Datenobjekten sowie die *Direct Unit* zur Verbindung von *Units* [CFB00, S. 6-9]. Die Dateneingabe z. B. durch Formulare wird

durch *(Data) Entry Units* ermöglicht [CFB+02, S. 89 f.][BCFM00], wobei die Speicherung, Modifikation und Löschung durch entsprechende *Operation Units* expliziert wird [CFB+02, S. 137-156][BCFM00].

Die *Pages* und *Units* werden für konditionale Navigationspfade und für den Transport von Informationen durch verschiedene Arten von *Links* miteinander verknüpft [CFB+02]. Grundlegend sind *kontextuelle* und *nicht-kontextuelle Links* [CFB00, S. 10]. *Kontextuelle Links* verbinden *Units* und transportieren über Parameter Informationen, welche als Kontext bezeichnet werden. Der Kontext wird durch die Quell-*Unit* determiniert [CFB+02, S. 98][CFB00, S. 13]. So trägt z. B. ein kontextueller *Link* zwischen einer *Index Unit* und *Data Unit* die Information, welches anhand eines Schlüssels identifizierte Datenobjekt ausgewählt wurde und angezeigt werden soll. Ein *nicht-kontextueller Link* verbindet *Pages* aus reinen Navigationsgesichtspunkten ohne solche Informationen und Abhängigkeiten miteinander. Als spezielle Formen von *Links* in Verbindung mit einer *Operation Unit* drücken *OK-* und *KO-Links* aus, welche *Unit* in Abhängigkeit vom Erfolg der Operation angezeigt werden soll [BCFM00]. Über die genannten Sprachelemente hinaus ermöglichen *globale Parameter* die Modellierung seitenübergreifender Variablen im Kontext von Benutzersitzungen, auf die mittels spezieller *Units* zugegriffen werden kann [BCFM07, S. 232][CFB+02, S. 106-110].

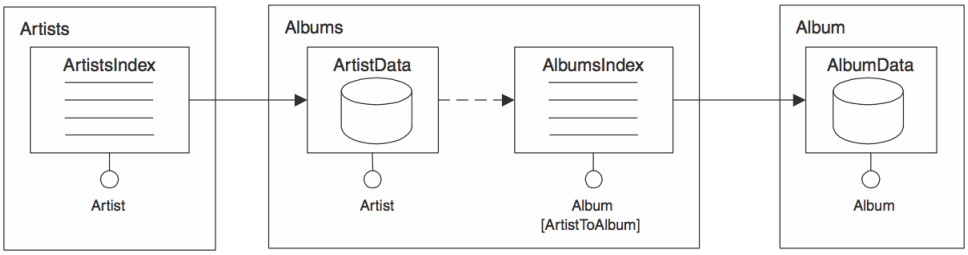

Abbildung 13: WebML Hypertext Model (Abbildung aus [CFB+02, S. 118])

Das *Presentation Model* behandelt die graphische Darstellung der Webapplikation bzw. der *Pages* in Form von *Style Sheets*. Dabei wird unterschieden zwischen *untyped style sheets*, welche das Layout der Seite unabhängig vom Inhalt definieren, und *typed style sheets* für spezifische *Pages*.

Das *Personalization Model* befasst sich mit der Modellierung von Benutzern und Gruppen, welche als Ankerpunkte für die benutzer- und gruppenspezifische Speicherung von Daten dienen.

Auf Basis der vorgestellten Sprachelemente existieren Referenzmuster von Webapplikationen, welche charakteristische Konstellationen von *Pages*, *Units* und *Links* in Webapplikationen vereinheitlichen [CFB+02, S. 117-126, 157-159]. So kann beispielsweise eine

Musikdatenbank generisch als eine kaskadierte Abfolge von Listen- und Detaildarstellungen modelliert werden (vgl. Abbildung 13). Aus einer Liste von Interpreten auf der Seite *Artists* kann ein Interpret ausgewählt werden, dessen Daten auf der Seite *Albums* inkl. einer Liste der Alben angezeigt werden. Die Selektion eines Albums führt zu einem Aufruf der Seite *Album* mit Detailangaben zum Album. Die durch Kreise gekennzeichneten Datenobjekte *Artist* und *Album* sind im Data Model enthalten und werden im Hypertext Model referenziert.

Über die Modellierungssprache hinaus sieht WebML als Methode ein Vorgehensmodell vor [CFB+02, S. 191-326][CFB00, S. 4], das analog zu den Modelltypen die Phasen der Anforderungserfassung und -analyse, des Datenentwurfs, des Hypertextentwurfs, des Präsentationsdesigns, des Benutzer- und Gruppenentwurfs und des Entwurfs spezifischer Anpassungen umfasst. Dabei werden entsprechende Diagrammtypen der UML wie z. B. Use-Case-Diagramme unterstützend einbezogen.

Die Menge der Unit-Typen kann projektspezifisch durch generische Operationen erweitert werden, deren Bedeutung vom Entwickler festzulegen ist [CFB+02, S. 163 f.]. Darüber hinaus decken Weiterentwicklungen von WebML die Unterstützung von Geschäftsprozessen und Webservices [BB10, BCFM06] (vgl. Kapitel 6) sowie desktopähnlichen Webapplikationen (Rich Internet Applications, RIAs) [PLCS07, Tof07] mit einer getrennten Betrachtung von client- und serverseitiger Logik [BCFT06] ab.

Das Metamodell von WebML existiert in verschiedenen Repräsentationsformen, die sich mit den verschiedenen Sprachversionen und der Weiterentwicklung des MDSD herausgebildet haben. Die technische Sphäre von WebML ist XML, da die Modelle als XML-Dateien serialisiert werden [BCFM07, S. 235-238], welche konform zu DTDs der jeweiligen Sprachversion von WebML sind [CFB+02]. Die Transformation basierte zudem in älteren Versionen auf XSLT als Transformationssprache [BCFM07, S. 235-238], was die Zuordnung von WebML zur technischen Sphäre von XML unterstreicht.

Als Alternative zu der XML-basierten Version von WebML existieren Varianten für UML 1 [CFB+02, S. 129-135] und UML 2 in Form von UML-Profilen [MFV07, MFV06]. Das Hypertext Model wird dabei als Klassen- oder Strukturdiagramm formuliert, in dem *Pages* und *Units* als Klassen oder StructuredClassifier und Komponenten dargestellt werden. *Links* werden als Assoziationen [CFB+02, S. 129] oder andere UML-Beziehungen zwischen Klassen [MFV06, S. 7] dargestellt. Die UML-Elemente werden durch die in den UML-Profilen enthaltenen Stereotypen gekennzeichnet (vgl. Abbildung 14). Auch bei der Darstellung von Webapplikationen durch UML-Modelle zielt WebML nicht auf die detaillierte Modellierung prozeduraler Programmlogik ab, schließt diese aber nicht aus [CFB+02, S. 171]. Zwischen den domänenspezifischen Sprachelementen von WebML und deren UML-

basierten Pendants existiert eine nicht-formale Transformationsdefinition in Form einer konzeptionellen Abbildungsvorschrift [CFB+02, S. 129-136].

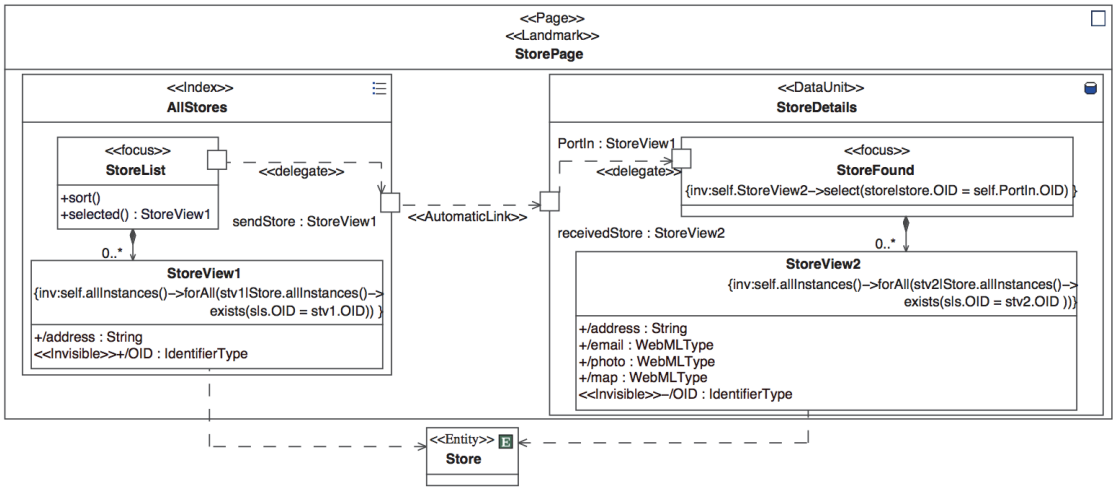

Abbildung 14: UML-basiertes WebML-Modell (Abbildung aus [MFV07, S. 33])

Als weitere Sprachdefinition von WebML existiert ein MOF- bzw. Ecore-basiertes Metamodell [SWK+07, SWK06]. Das ursprüngliche DTD-basierte und das MOF-basierte Metamodell sind durch generierte Transformationsdefinitionen miteinander verbunden [BFT08], sodass WebML-Modelle zwischen den beiden technischen Sphären ausgetauscht werden können.

Die WebML-Methode wird durch das kommerzielle Entwicklungswerkzeug *WebRatio Site Development Studio* (WebRatio)[16] implementiert, welche die Modellierung der Data und Hypertext Models sowie daraus die Generierung von Webapplikationen unterstützt. WebRatio ist eingebettet in die Eclipse IDE und erweitert diese um eine Implementierung der ursprünglichen DSL-XML-basierten Version von WebML [ABBB07]. Als Zielplattformen werden Java-Applikationsserver verschiedener Hersteller adressiert, wobei der generierte Quelltext unabhängig von WebRatio lauffähig ist und weiterentwickelt werden kann [Web11, S. 6]. Die Transformationsdefinition ist in WebRatio in Form von Templates bzw. Generatorregeln in der Programmiersprache Groovy formuliert[17]. Intermediäre PSMs zwischen den plattformunabhängigen WebML-Modellen und dem Quelltext sind nicht vorgesehen, sondern der Code wird direkt aus dem PIM generiert (vgl. [FT10]).

Mit WebRatio existiert für WebML eine ausgereifte kommerzielle Implementierung, die über den prototypischen Charakter anderer MDWE-Ansätze hinausgeht und in einer Vielzahl an Referenzprojekten die Praktikabilität einer abstrakten Modellierung von

[16] `http://www.webratio.com`
[17] `http://wiki.webratio.com/index.php/Getting_started_with_Groovy`

Webapplikationen mittels aggregierender Sprachelemente nachgewiesen hat. Mit dem Codegenerator von WebRatio verfügt der Ansatz über MDSD-Fähigkeiten, eine vollständige Unterstützung der MDA ist aber nicht gegeben. So finden in der Praxis die UML- und MOF-basierten Metamodelle keine Anwendung, sodass die Transformationsdefinition für die Codegenerierung nicht in einer standardisierten Transformationssprache aus der technischen Sphäre von MOF verfasst ist. Darüber hinaus steht eine Beschreibung von Webapplikationen durch PSMs nicht im Fokus von WebML.

3.4 UML-based Web Engineering (UWE)

UML-based Web Engineering (UWE) ist eine Methode für das MDWE, die seit dem Jahr 1999 entwickelt wird [BKM99, KM99] und eine gleichnamige Modellierungssprache umfasst. UWE war nach der Etablierung der UML neben WAE einer der ersten MDWE-Ansätze, der Webapplikationen ausschließlich mit Mitteln der UML modelliert. Das Metamodell von UWE ist zu diesem Zweck als UML-Profil mit Stereotypen, Tagged Values und Constraints spezifiziert und kann in UML-Werkzeuge eingebunden werden.

Das Vorgehensmodell von UWE umfasst eine *Analysephase* zur Erstellung der Anforderungsspezifikation und eine *Entwurfsphase* zur Erstellung von Entwurfsmodellen für die Daten, die Navigationsstruktur und die Präsentation [KKZB08, S. 163-176]. Die Methode sieht vor, dass über die Phasen hinweg die zu entwickelnde Webapplikation für eine Trennung der Belange (Separation of Concerns) mittels der verschiedenen Modelltypen aus verschiedenen Sichten repräsentiert wird [KKZB08, S. 158]. Die Phasen sowie die in diesen erstellten Modelltypen variieren im Detail mit den Versionen von UWE und den konkreten Projektanforderungen, befassen sich aber im Wesentlichen anhand eines *Requirements Model* mit den Anforderungen an die Webapplikation und anhand eines *Content Model*, *Navigation Model* und *Presentation Model* mit der Struktur und dem Verhalten des (Daten-)Inhalts, mit der Navigationsstruktur und mit der Präsentation [SK06, S. 41][KK03][Koc01, S. 145-194].

Im *Requirements Model* werden die funktionalen Anforderungen erfasst. Es gibt eine grobe Beschreibung der Funktionalität durch Use-Case-Diagramme und eine detailliertere Darstellung nicht-trivialer Use-Cases z. B. durch verfeinernde Aktivitätsdiagramme [KKZB08, S. 163-166]. Nicht-funktionale Anforderungen können als Teil der *Anforderungsspezifikation* ergänzend durch textuelle Dokumente erfasst werden, welche z. B. den Benutzerkreis und Einsatzszenarien der Webapplikation abdecken [Koc01, S. 213].

Das *Content Model* beschreibt als Entwurfsmodell die domänenrelevanten Daten bzw. Inhalte der Webapplikation. Es adressiert zwar die Webapplikation als System im Lösungsraum, fokussiert aber auf fachliche Zusammenhänge des Problemraums und abstrahiert von

technischen Details. Die Inhalte bzw. die durch die Webapplikation abgebildeten Domänenkonzepte werden in einem UML-Klassendiagramm als Klassen dargestellt, die Beziehungen zwischen den Inhalten als Assoziationen. [KKZB08, S. 166-168]

Das *Navigational Model* beschreibt die Navigationsstruktur bzw. die Struktur des Hypertexts der Webapplikation in Form eines Klassendiagramms (vgl. Abbildung 15). Die Navigationsstruktur ist ein Graph von Navigationsknoten und Links, der den hypertextuellen Charakter der Webapplikation widerspiegelt und mögliche Navigationspfade des Benutzers im System definiert. Der Graph wird in einem Klassendiagramm durch (Navigations-)Klassen und Assoziationen dargestellt.

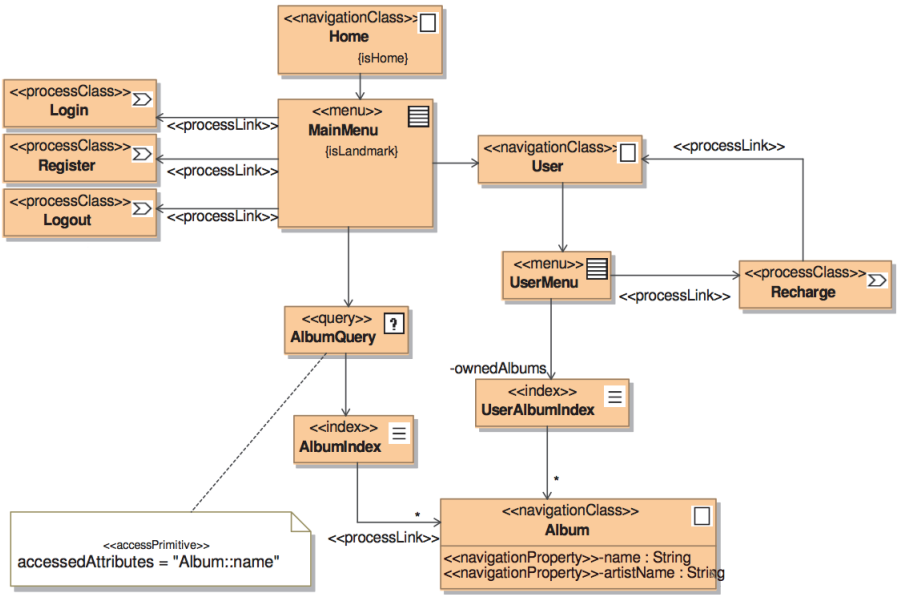

Abbildung 15: Navigation Model in UWE (Ausschnitt aus [KK08, S. 27])

Initial werden Navigationsklassen datenorientiert aus gleichnamigen Klassen des *Content Model* abgeleitet, indem für relevante Domänenklassen, auf die der Nutzer mittels der Navigation zugreifen soll, entsprechende Navigationsklassen als Sicht auf das *Content Model* angelegt werden (vgl. [Koc01, S. 156-159]). Die Domänenklassen finden sich damit in Form der Navigationsklassen indirekt als Teil des Navigationsgraphen wieder. Anschließend werden die Navigationsklassen als datenorientierte Knoten im Navigationsgraph um Zugriffsknoten in Form von Menüs und Zugriffsprimitiven (*access primitives*) ergänzt (vgl. [Koc01, S. 159-164]). Letztere sind z. B. *Index* für Listen über Instanzen von Navigationsklassen, *Guided Tour* für seitenweise sequentielle Datenausgaben, *Query* für die Selektion von Datensätzen z. B. durch Suchformulare und *Process Class* für die Anbindung von Geschäftsprozessen. [KKZB08, S. 168-171]

Das *Presentation Model* beschreibt basierend auf dem *Navigation Model* die Benutzer-

schnittstelle der Webapplikation (vgl. Abbildung 16). Dabei wird von graphischen Details abstrahiert, indem plattform- und implementierungsunabhängig lediglich die Struktur der Benutzerschnittstelle mit ihren Elementen und Verbindungen zu den Navigationsknoten dargestellt wird [KKZB08, S. 172-173][Koc01, S. 173-189].

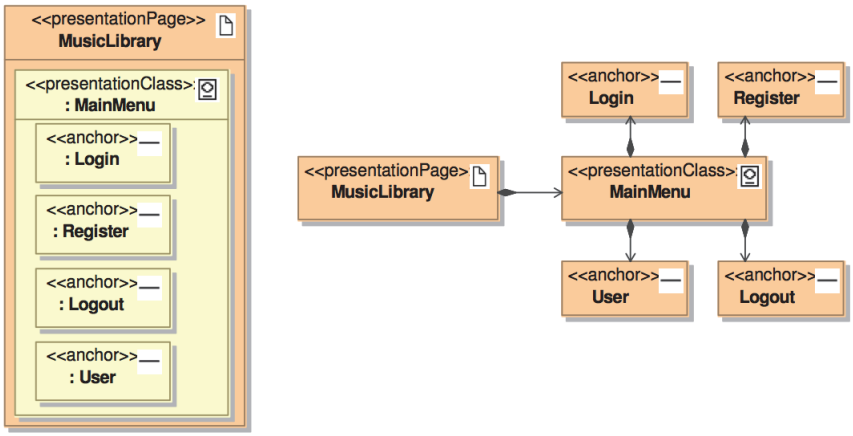

Abbildung 16: Beispiel für zwei synonyme Presentation Models in UWE (Abbildung aus [KK08, S. 33])

Webseiten (*pages*) der Webapplikation werden in einem Klassendiagramm modelliert und bestehen aus Präsentationsklassen, welche durch schachtelbare Präsentationsgruppen (*presentation group*) organisiert werden können. Jeder Präsentationsklasse wird gewöhnlich aus dem *Navigation Model* jeweils ein gleichnamiger Knoten bzw. eine Navigationsklasse, ein Zugriffsprimitiv, etc. zugeordnet. Eine Präsentationsklasse wird feingranular durch UI-Elemente wie z. B. *text*, *form*, *anchor* etc. detailliert, die in einer Kompositionsbeziehung zu der Präsentationsklasse stehen. Dabei werden Attribute von Navigationsklassen durch korrespondierende Attribute der Präsentationsklasse abgebildet. Das Verhalten der GUI-Elemente wird darüber hinaus mithilfe von Zustandsdiagrammen [BKM99] modelliert, im Speziellen für die Spezifikation von Mustern bei der Präsentationslogik von RIAs [KPZM09].

Das Metamodell von UWE ist als leichtgewichtige UML-Erweiterung in Form eines UML-Profils [Obj10b, S. 177 ff.] mit Stereotypen, Tagged Values und Constraints spezifiziert[18] [KK08]. Die Metamodellelemente von UWE sind Stereotypen, welche Metamodellelemente der UML erweitern und in UML-Paketen entsprechend der Modelltypen organisiert sind. Das UML-Paket *Requirements* enthält Stereotypen für die UML-Metaklassen *UseCase* und *ActivityNode* [EK06, S. 4]. Die Pakete *Navigation* und *Presentation* enthalten Stereotypen der UML-Metaklasse *Class* für die Klassendiagramme der beschriebenen Ent-

[18] `http://uwe.pst.ifi.lmu.de/publicationsMetamodelAndProfile.html`

wurfsmodelltypen. Die Pakete *Content* und *Process* enthalten keine Stereotypen, da das *Content Model* und *Process Model* mit den nativen Sprachelementen von UML formuliert werden. Die statische Semantik von UWE wird darüber hinaus mit mit OCL spezifiziert.

Das Vorgehensmodell von UWE wird durch Transformationsdefinitionen unterstützt, welche Transformationen über die verschiedenen Modelltypen hinweg ermöglichen. Die Transformation von UWE-Modellen zu Quelltext wurde mit den Innovationen bei den Transformationstechniken weiterentwickelt [KKK07b, S. 2]. Anfänglich war der Transformationsprozess manuell durchzuführen und an den Rational Unified Process angelehnt [Koc01, S. 211]. Der manuelle Prozess wurde anschließend durch hartcodierte Transformationsdefinitionen ersetzt, die mittlerweile in etablierten Transformationssprachen formuliert werden. Da UWE mit der UML in der technischen Sphäre von MOF entwickelt wird, finden für die Transformation verschiedene MOF-kompatible Technologien wie QVT, ATL und OCL Verwendung [KKZB08, S. 180-185][Koc06, S. 6].

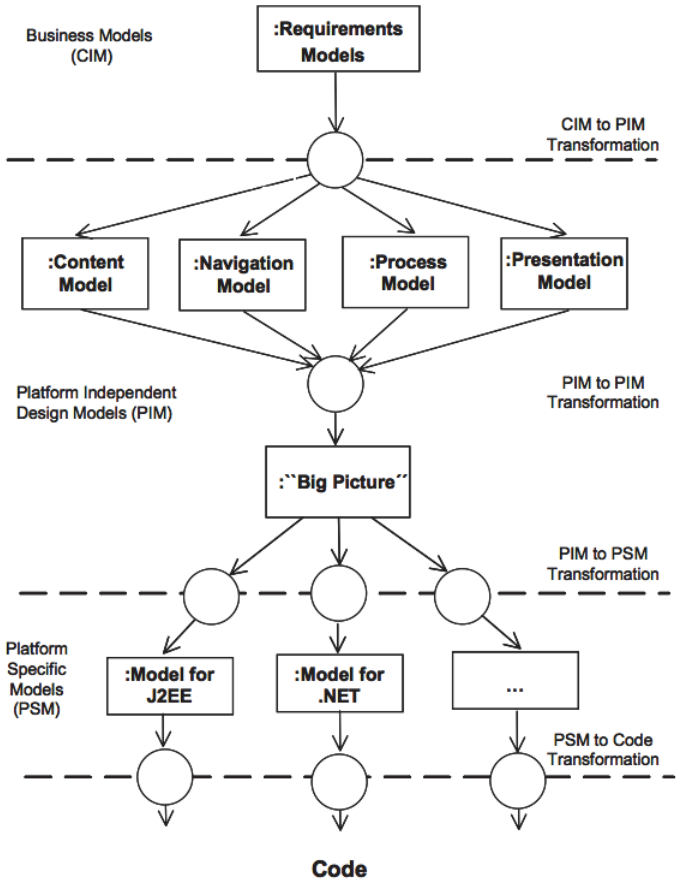

Abbildung 17: Generatorprozess von UWE (Abbildung aus [KZE06, S. 282])

Im Ordnungsrahmen der MDA wird das *Requirements Model* als CIM eingeordnet.

Die Modelle *Content*, *Navigation*, *Presentation* und *Process Model* werden als PIMs unter der Bezeichnung *Functional Models* aus diesem generiert [KZE06] (vgl. Abbildung 17). Die PIMs werden anschließend in ein gemeinsames plattformunabhängiges Totalmodell (*Big Picture*) zusammengeführt, das zur Validierung dient. Konzeptionell ist es zudem als Ausgangsmodell für die Generierung von PSMs vorgesehen. Die Transformation läuft mittels des Generators automatisch ab, setzt aber teilweise Anreicherungen der generierten Modelle durch den Entwickler voraus, z. B. die Kennzeichnung von Content-Klassen bzgl. ihrer Relevanz für das *Navigation Model*. [KKZB08, S. 181][KKK07b, Koc06]

Weiterentwicklungen von UWE integrieren zudem ein Architekturmodell [KKZB08, S. 181,183] mittels der *Web Software Architecture* (WebSA). Diese ist ein MDA-orientierter Ansatz, der verschiedene MDWE-Ansätze wie UWE und OOH (vgl. Kapitel 3.5) komplementär um eine UML-basierte Beschreibung der Systemarchitektur ergänzt [MGS07, MG06a].

UWE wird als Methode und Modellierungssprache durch mehrere Werkzeuge implementiert. ArgoUWE[19] ist eine nicht mehr weiterentwickelte Erweiterung des CASE-Werkzeugs ArgoUML[20], das eine Modellierungsumgebung für UML 1.4- sowie UWE-Diagramme bereitstellt und über eine hartcodierte M2M-Transformationsdefinition verfügt. MagicUWE[21] ist ein Plugin für das UML 2-Werkzeug MagicDraw[22] und erweitert dieses um die Modelltypen von UWE sowie um hartcodierte M2M-Transformationsdefinitionen [BK09]. UWE4JSF[23] ist ein Eclipse-Plugin, welches die Transformation von in Eclipse erstellten UWE-Modellen in Webapplikationen für JavaServer Faces ermöglicht [KK09, Kro08]. UWE2JSF basiert auf EMF und enthält Transformationsdefinitionen, die in ATL für M2M-Transformationen und JET für die Codegenerierung formuliert sind.

Das charakteristischste Merkmal von UWE ist die Standardisierung durch eine strikte Ausrichtung an der UML und an Standards der OMG. Dies hat den Vorteil einer werkzeugunabhängigen Implementierung der UWE-Konzepte und vermeidet proprietäre Insellösungen wie im CASE. Eine weitere Stärke von UWE ist die methodische Integration strukturierter Anforderungsmodelle in den Generatorprozess. Die Fokussierung auf die UML wird aber zur Schwäche, wenn es um eine präzise Abbildung der Zielplattformen geht, da der UML Sprachelemente für die detaillierte Modellierung von z. B. Konfigurationsdateien, relationalen Schemata, Webservice-Definitionen oder GUI-Schablonen fehlen. Dieses Problem zeichnet sich auch darin ab, dass UWE bislang primär Codegeneratoren

[19] `http://uwe.pst.ifi.lmu.de/toolargoUWE.html`
[20] `http://argouml.tigris.org/`
[21] `http://uwe.pst.ifi.lmu.de/toolMagicUWE.html`
[22] `http://www.magicdraw.com/`
[23] `http://uwe.pst.ifi.lmu.de/toolUWE4JSF.html`

für objektorientierte Zielplattformen wie Java EE bietet.

3.5 Object-oriented Hypermedia Method (OO-H)

Die Object-oriented Hypermedia Method (OO-H) wird seit dem Jahr 2000 entwickelt [GCP00] und ist eine partiell UML-basierte Entwicklungsmethode für Webapplikationen, die auf der *OO-Method* basiert.

Die *OO-Method* ist eine Entwicklungsmethode und -umgebung für objektorientierte Softwaresysteme, mit der ganz allgemein ein objektorientiertes Softwaresysteme durch die Verbindung aus einem graphischen konzeptionellen Modell (*Conceptual Model*) und einer textuellen formalen Spezifikationen (Ausführungsmodell, *Execution Model*) beschrieben wird [PGIP01, PIP+97]. Die Methode ist mehrstufig modellgetrieben, da aus dem konzeptionellen Modell das Ausführungsmodell, und aus diesem prototypischer Code für verschiedene Programmiersprachen (C++, Delphi, Java, VB) generiert wird (vgl. Abbildung 18). Das konzeptionelle Modell wird mit den Ausdrucksmitteln der UML formuliert und besteht aus einem Klassendiagramm von Domänenklassen und Benutzerrollen (*Object Model*), einem Zustandsübergangs- und Objektinteraktionsdiagramm (*Dynamic Model*) und einem Modell zur Beschreibung der Semantik von Zustandsübergängen (*Functional Model*). Das Ausführungsmodell beschreibt mit Mitteln der textuellen Spezifikationssprache OASIS [PGIP01, S. 513] auf formale, abstrakte und objektorientierte Weise die Zugriffskontrolle, die Objektwelt des Softwaresystems und die Dienste auf Basis der Objekte. Mit der Einteilung in Modelltypen auf verschiedenen Abstraktionsebenen und der schrittweisen Generierung weist die OO-H Method Charakteristika der MDA auf, wurde aber bereits vor dieser entwickelt.

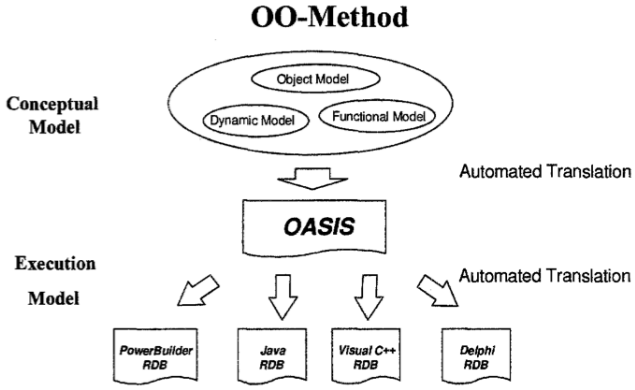

Abbildung 18: Phasen der OO-Method (Abbildung aus [PIP+97, S. 148])

Um über allgemeine Softwaresysteme hinaus Webapplikationen mit ihren Spezifika konzeptionell abbilden zu können, erweitert die Object-oriented Hypermedia Method (OO-H)

das konzeptionelle Modell der OO-Method im Wesentlichen um die plattformunabhängigen Diagrammtypen *Navigation Access Diagram* (NAD) [GCP00] für die Navigationssicht und *Abstract Presentation Diagram* (APD) [GCP01] für die Präsentationssicht (vgl. Abbildung 19). Zusätzlich umfasst OO-H Transformationsdefinitionen für die Generierung prototypischer Webapplikationen.

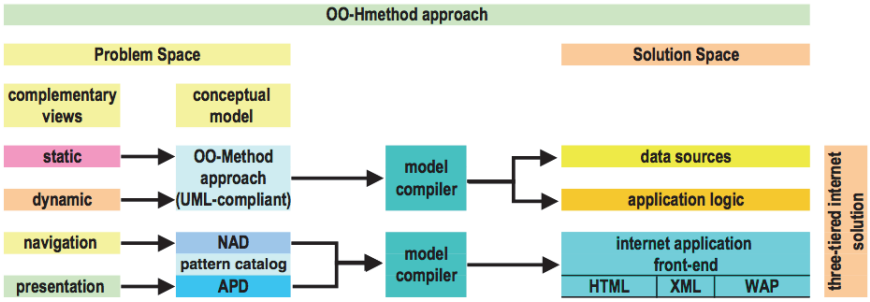

Abbildung 19: Phasen der OO-H Method (Abbildung aus [GCP01, S. 3])

Die Navigationsstruktur der Webapplikation wird durch ein oder mehrere NADs beschrieben, wobei für jede Benutzerrolle aus dem *Object Model* der OO-Method die möglichen Navigationspfade im System mit jeweils mindestens einem NAD dargestellt werden. Ein NAD hat die Form eines UML-Klassendiagramms, das die Navigationsstruktur datenorientiert als Graph von *Navigationsklassen* (Navigation Class, NC), *Navigationszielen* (Navigation Target, NT), *Navigationslinks* (Navigation Link, NL) und *Collections* beschreibt (vgl. Abbildung 20). [GC02, S. 148-151][GCP01, S. 5-7])

Navigationsklassen sind angereicherte Domänenklassen und repräsentieren in Form von UML-Klassen Daten, auf die Benutzer durch die Webseiten zugreifen können. Sie werden durch Navigationsziele in Form von UML-Paketen zusammengefasst, die als Knoten im Navigationsgraphen Use-Cases abbilden und auf den Navigationsklassen operieren. Ein Navigationsziel repräsentiert somit nicht direkt eine Webseite, sondern einen Use-Case bzw. eine Benutzeranforderung, der durch eine Menge beliebig vieler Webseiten erfüllt wird.

Navigationspfade zwischen Navigationsklassen bzw. deren darstellenden Webseiten werden durch gerichtete Navigationslinks bzw. im Graphen durch Kanten dargestellt. Es werden fünf Typen von Links unterschieden. Ein *I(nternal)-Link* definiert einen Navigationspfad innerhalb eines Navigationsziels, ein *T(raversal)-Link* den Navigationspfad zwischen zwei Navigationszielen, ein *R(equirement)-Link* kennzeichnet den Startpunkt der Navigation in einem Navigationsziel, ein *(E)X(it)-Link* einen Endpunkt und ein *S(ervice)-Link* drückt aus, dass der Benutzer mit einer Navigationsklasse als Service interagieren kann [GC02, S. 149]. Jeder Navigationslink ist mit einem *Navigation Pattern* attributiert, das

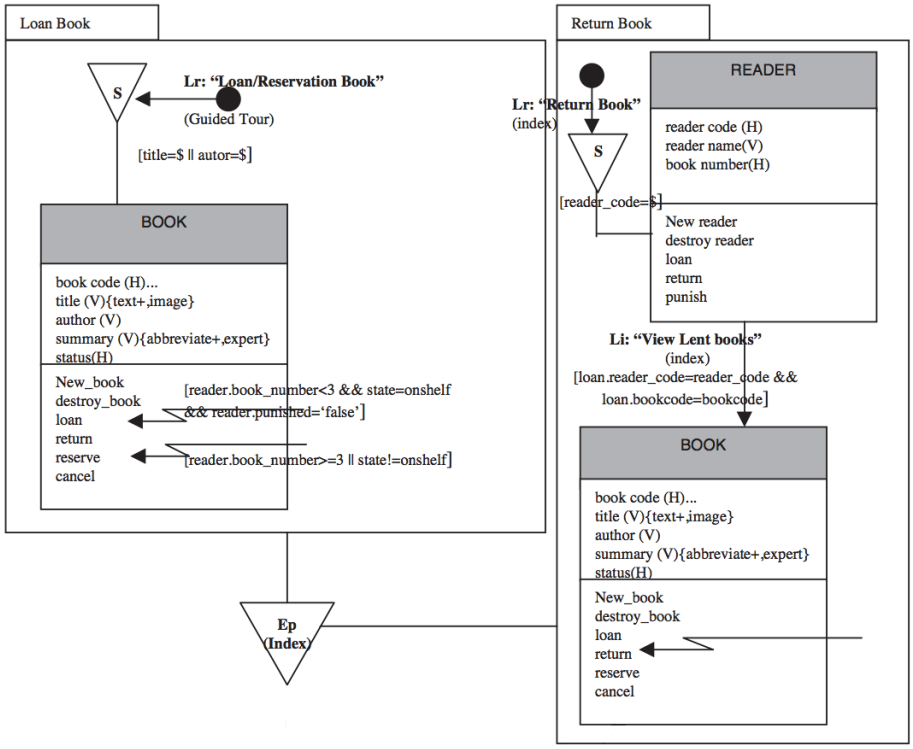

Abbildung 20: NAD für ein Bibliothekssystem (Ausschnitt aus [GCP00, S. 88])

mit den Werten *Index*, *Guided Tour*, *Indexed Guided Tour* oder *Showall* ausdrückt, ob die Navigationsklassen am Ziel des Navigationslinks auf der resultierenden Webseite als Liste von Verweisen auf Datensätze, als seitenweise Darstellung von Datensätzen, als Mischung daraus oder als Liste der Datensätze dargestellt werden.

Collections repräsentieren in Form von Dreiecken Navigationshilfen für den Benutzer wie z. B. menüartige Listen zusammengehöriger Links oder Listen von Links, welche der Benutzer bisher aufgerufen hat.

So können beispielsweise für die Webapplikation einer Bücherei (vgl. Abbildung 20) die Ausleih- und Rückgabevorgänge aus Sicht der Benutzerrolle *Bibliothekar* durch die zwei Navigationsziele *Loan Book* und *Return Book* dargestellt werden. Im Fall des Navigationsziels *Return Book* ist der Einstiegspunkt in das Navigationsziel durch den R-Link *Return Book* gekennzeichnet, welcher auf eine S(elector)-Collection verweist, die den Datensatz *Reader* des Benutzers selektiert. Der ausgehende I-Link *View Lent Books* erzeugt einen Index bzw. eine Liste von Verweisen auf Datensätze der Navigationsklasse *Book*. Der S-Link auf die Operation bzw. den Service *return* (dargestellt durch einen gezackten Pfeil) kennzeichnet den Aufruf des Services im Kontext des Navigationsziels.

Das APD beschreibt konzeptionell und plattformunabhängig die GUI der Webapplikation durch Templates, die in Form von XML-Dokumenten strukturiert werden. Für das

APD sind verschiedene Templatetypen vordefiniert [GC02, S. 164 f.], mit denen verschiedene Arten von Webseiten bzw. Ansichten z. B. zur Datenausgabe oder Dateneingabe per Formular einheitlich beschrieben werden. Beispiele für Templatetypen sind *Tstruct* zur Spezifikation der Informationen auf einer Webseite, *Tstyle* zur Konfiguration der graphischen Formatierung, *Tform* zur Beschreibung formularbasierter Benutzereingaben, *Tfunction* für clientseitige Präsentationslogik (z. B. durch JavaScript) und *Twindow* für Kompositionen aus Templates auf einer Webseite (vgl. [GCP01, S. 7]). Einige der Templatetypen können darüber hinaus bzgl. des graphischen Layouts in einem *Composite Layout Diagram* (CLD) konfiguriert werden [GC02, S. 165].

Bei der Erstellung des NAD und des APD kann auf einen Katalog vordefinierter Entwurfsmuster zurückgegriffen werden, die mittels Transformationsregeln in das NAD bzw. APD injiziert werden [GC02, S. 165]. Ein Entwurfsmuster für das APD ist beispielsweise das *Location Pattern*, das zur Orientierung des Benutzers im Hypertext der Webapplikation unterstützende Kennzeichnungen z. B. in Form von Kopfzeilen auf den Webseiten vorschlägt. [GCP01, S. 4 ff.]

Die Entwicklung startet mit der Erstellung von Use-Case-Diagrammen und einem Business Class Diagram [GC02, S. 151 ff.], das ursprünglich als Object Model ein Teil der OO-Method ist [GCP00, S. 81]. Aus diesen werden für die verschiedenen Benutzerrollen vom Modellierer NADs erstellt, indem grobgranular aus den Use-Cases Navigationsziele abgeleitet werden, die ausführen, welche Daten und Funktionen über Navigationsknoten von dem Benutzer erreicht werden können [GC02, S. 155-164][GCP00, S. 3]. Aus den NADs werden anschließend automatisiert prototypische APDs abgeleitet, die als Vorlage für manuelle Verfeinerungen dienen [GC02, S. 164][GCP01, S. 9]. Die verfeinerten XML-basierten APDs dienen abschließend als Ausgangspunkt für die Transformation in Quelltext mittels eines *Model Compilers* (vgl. Abbildung 19).

Weiterentwicklungen von OO-H befassen sich unter anderem mit der Repräsentation von Geschäftsprozessen mit Aktivitätsdiagrammen und NADs [KKCM03] sowie der Generierung von RIAs [MGPD08, PDMG08]. Mit WebSA wird OO-H zudem um eine Beschreibung der Systemarchitektur ergänzt [MGS07, MG06a].

OO-H wird durch das CASE-Werkzeug *VisualWADE*[24] unterstützt [Góm04], das NADs mittels der UML 2 formuliert. Es kann aus den NADs XML-Dateien für das APD generieren und bietet einen WYSIWYG-Editor für deren Modifikation. Die XML-Dateien werden abschließend mit einem Model Compiler in PHP-basierten Code überführt [Góm04]. Bei alternativen Versionen von OO-H kommen aktuelle Techniken der MDA wie z. B. MOF für Metamodelle und QVT für Transformationsdefinitionen zum Einsatz. So existiert z. B. für

[24] `http://www.visualwade.com/`

Präsentationsmodelle von RIAs ein entsprechendes plattformspezifisches Metamodell für das Google Web Toolkit (GWT) [MGPD08], das im Unterschied zu dem XML-basierten APD auf MOF basiert.

OO-H ist damit mehreren technischen Sphären zuzuordnen. Das konzeptionelle NAD verwendet UML-Techniken, während plattformspezifische Modelle entweder mit Mitteln der XML oder des MOF ausgedrückt werden. Bezüglich der Transformationsdefinitionen liegt ebenso eine Mischung aus hartcodierten und standardbasierten Implementierungen vor.

Ähnlich zu UWE sind auch bei OO-H Datenklassen ein Teil des Navigationsgraphen. Charakteristisch für OO-H ist aber die Verwendung von Navigationslinks im NAD als Informationsträger für die Art der Darstellung von Daten am Ende des Links, z. B. als Index. Dies führt zu einer verdichteten Darstellung, welche die Lesbarkeit erschweren kann. Der Katalog von Entwurfsmustern ermöglicht die Wiederverwendung vordefinierter Lösungsansätze, die ähnlich zu der aspekteorientierten Programmierung eingewoben werden.

Als Nachteil für das MDSD sind die nur partielle Standardisierung und die Aufteilung auf unterschiedliche technische Sphären zu sehen, welche die Spezifität und Komplexität der Implementierung erhöht. Hinzu kommt im Fall des APD, dass trotz der Plattformnähe die detaillierte Darstellung von Plattformspezifika ausbleibt oder im Fall MOF-basierter Weiterentwicklungen nur ausschnittweise erfolgt.

3.6 Object Oriented Web Solution (OOWS)

Die Object Oriented Web Solution (OOWS) ist eine Softwareentwicklungsmethode für Webapplikationen, die seit dem Jahr 2001 [PAF01a, PAF01b] entwickelt wird und wie auch OO-H (vgl. Kapitel 3.5) eine Erweiterung der OO-Method ist [FVR+03]. OOWS umfasst basierend auf der UML Modelltypen zur konzeptionellen Repräsentation der Navigations- und Präsentationsaspekte von Webapplikationen sowie generative Komponenten.

Die Methode ist in die Phasen der *konzeptionellen Modellierung* (Conceptual Modelling) und der *Lösungsentwicklung* (Solution Development) unterteilt [FVR+03]. Dabei beinhaltet die Modellierungsphase sequentiell die Erhebung der Anforderungen z. B. mittels Aktivitäts- [VFP05] oder Use-Case-Diagrammen, die Erstellung des konzeptionellen Modells im Sinne der OO-Method und die Modellierung der Navigation sowie Präsentation mit OOWS-eigenen Mitteln.

Das *Navigationsmodell* enthält für jeden Benutzertyp der Webapplikation einen gerichteten Navigationsgraphen, der personalisiert die möglichen Navigationspfade beschreibt. Jeder Knoten des Graphen repräsentiert einen Bereich der Webapplikation, der als *Navigationskontext* bezeichnet wird und in Form eines UML-Pakets modelliert wird, das mit dem

Stereotyp *«context»* gekennzeichnet ist (vgl. Abbildung 21). Die Erreichbarkeit eines Navigationskontexts von anderen Knoten des Navigationsgraphen wird sowohl über die Kanten bzw. *Navigationslinks* als auch durch das Attribute *reachability* festgelegt. Der Wert *S* kennzeichnet, dass ein Knoten ausschließlich über die Navigationspfade des Graphen erreichbar ist, der Wert *E* dagegen kennzeichnet eine generelle Erreichbarkeit von allen anderen Knoten, z. B. über ein Navigationsmenü. Der Wechsel des Benutzertyps beim Übergang von Navigationskontexten z. B. im Fall einer Authentifizierung wird durch einen UML-Actor gekennzeichnet, der an einem Navigationslink annotiert wird. [FVR+03, FPAP03]

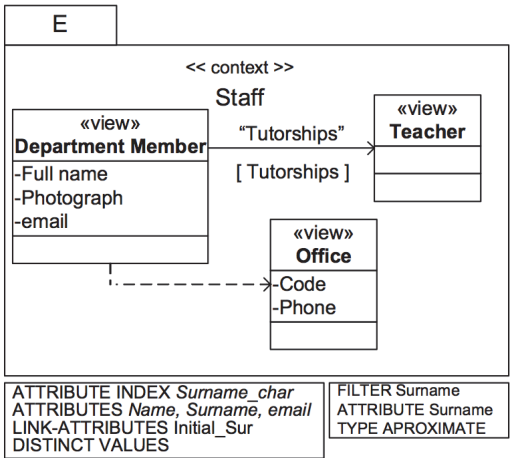

Abbildung 21: OOWS Navigationskontext (Abbildung aus [FVR+03, S. 3])

Navigationskontexte enthalten als *Navigationsklassen* bezeichnete UML-Klassen, die mit dem Stereotyp *«view»* ausgezeichnet werden und den Navigationskontext genauer spezifizieren. Navigationsklassen werden aus den Datenklassen des konzeptionellen Modells abgeleitet und repräsentieren, auf welche Daten in welcher Form in einem Navigationskontext zugegriffen werden kann. Für einen Navigationskontext wird dazu spezifiziert, in welcher Form ein Index bzw. eine Liste von Datensätzen ausgegeben oder eine Suche ermöglicht werden soll. Entsprechend den Datenklassen stehen die Navigationsklassen eines Navigationskontexts durch verschiedene UML-Beziehungstypen in Beziehung zueinander, die ausdrücken, ob die Information einer assoziierten Navigationsklasse in einem eigenständigen Navigationsknoten (*Context Relationship*, durchgezogener Pfeil) oder auf der Ansicht des verweisenden Knotens (*Context Dependency relationship*, gestrichelter Pfeil) dargestellt werden soll. Die Navigationsklasse (*Manager Class*) eines Navigationskontexts mit ausschließlich ausgehenden Beziehungstypen zu referenzierten Klassen (*Complementary Classes*) wird initial bei einem Aufruf des Kontexts aufgerufen. [FVR+03, FPAP03]

Das *Präsentationsmodell* verfeinert die *Navigationskontexte* des Navigationsmodells mit Annotationen, die vordefinierte *Präsentationsmuster* repräsentieren. Diese Muster sind na-

mentlich *Information Paging* für eine seitenweise Ausgabe mehrerer Datensätze, *Ordering* für die Sortierung mehrerer Datensätze und *Presentation of Instances* für die Spezifikation des Layouts. [FVR+03, FPAP03]

Für OOWS existieren verschiedene Generatoren, die korrespondierend zu den technischen Fortschritten im MDSD und in der MDA weiterentwickelt wurden. Für die Übersetzung von Anforderungsmodellen (CIM) zu Navigationsmodellen (PIM) existiert eine Transformationdefinition basierend auf dem Graphersetzungssystem AGG[25] [VFP05]. Eine andere Erweiterung von OOWS ermöglicht die Transformation von BPMN-Modellen (CIM) zu OOWS-Navigationsmodellen (PIM), aus denen anschließend Web-Benutzerschnittstellen zur Interaktion mit den Geschäftsprozessen generiert werden [TP06] (vgl. Kapitel 6.1). Dabei finden als Transformationstechniken QVTO für die M2M-Transformationsdefinition und MOFScript für die M2T-Transformationsdefinition Verwendung.

Die Generierung von Quelltext wird durch das kommerzielle Werkzeug *OlivaNOVA*[26] ermöglicht, das allgemein die OO-Method implementiert und im Speziellen um Editoren und Generatorkomponenten (Model Compiler) für die Modelltypen von OOWS erweitert wird [VVFP07]. Die Verbindung der OO-Method mit OOWS spiegelt sich darin wider, dass als Plattformen für die Geschäftslogik Java EE und .NET unterstützt werden, für die Webschnittstelle dagegen PHP. Das Metamodell von OOWS ist mit Ecore spezifiziert und greift Konzepte der UML auf [VVFP07, S. 7]. OOWS ist damit der technischen Sphäre von MOF und der UML zuzuordnen. Die Generatorkomponente von OOWS ist eingebettet in Eclipse und vermeidet eine M2M-Transformationsdefinition vom PIM zum PSM, indem vom PIM direkt Code gegen ein abstrahierendes Laufzeitframework generiert wird [VVFP07, S. 9 f.]. Als Transformationssprache für den Codegenerator fungiert Xpand [VVFP07, S. 11].

Die Standardorientierung von OOWS ist vorteilhaft für Anpassungen des Generators durch den Entwickler. Gleichwohl schränkt die Eigenschaft von OOWS, aus den konzeptionellen Modellen den Quelltext prototypischer Implementierungen ohne intermediäre PSMs zu generieren, die Flexibilität des Ansatzes ein. Hinzu kommt die Verwendung eines Laufzeitframeworks, das eine dauerhafte Koppelung des Generators an das Generat bedingt, die unerwünscht sein kann.

3.7 Weitere MDWE-Ansätze

MIDAS ist ein MDA-basierter Ansatz, welcher UML-basierte Modellierungssprachen für die plattformunabhängige und -spezifische Modellierung von Web-Informationssystemen im Sinne von Webapplikationen umfasst [CCVM06, MMÁ+06]. Die PIMs befassen sich

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

[26] `http://www.integranova.com/` und `http://www.programmiermaschine.de/`

mit den Daten, dem Hypertext und der Präsentation von Webapplikationen und werden durch Transformationsdefinitionen des MIDAS-CASE-Werkzeugs automatisiert in PSMs in Form von Klassendiagrammen und XML-Schemadefinitionen ohne Bezug zu einer konkreten Webplattform übersetzt [CMV03].

Hera ist eine modellgetriebene Methode, die ebenfalls zur Modellierung von Web-Informationssystemen dient. Es werden Aspekte von Drei-Schichtenarchitekturen durch verschiedene Diagrammtypen abgedeckt, zu denen solche für die Datenhaltung, die Navigation und Logik, die Präsentation, und orthogonal solche für plattformspezifische Anpassungen und für Benutzer zählen [FHB06, VFHB03]. Als technische Sphäre kommt das Resource Description Framework (RDF) bzw. XML zum Einsatz. Zur Modellierung und Codegenerierung existiert mit dem *Hera Presentation Generator* ein CASE-Tool, das Transformationsdefinitionen basierend auf XSLT und Java umfasst. Die Ausgabe ist entweder passiver Hypertext, oder eine Java-basierte Webapplikation [FHB06, S.197].

WebDSL ist eine domänenspezifische Modellierungssprache zur Modellierung von Webapplikationen [GHV10, Vis07]. Modelle werden mittels eines ausgereiften Eclipse-Plugins in textueller und plattformunabhängiger Form verfasst, und mit Hilfe des Transformationswerkzeugs Stratego/XT aus der technischen Sphäre des Syntax Definition Formalism (SDF) über interne intermediäre Modelle in Java-basierte Implementierungen transformiert [HKGV10].

Die *Fornax-Plattform*[27] befasst sich mit der Bereitstellung von Komponenten und Werkzeugen für das MDSD. Zu diesen zählen *Cartridges*, die UML-Profile für plattformspezifische UML-Modelle und M2T-Transformationsdefinitionen für Java-Frameworks mit Schwerpunkt auf objektrelationalen Mappern wie z. B. Hibernate enthalten.

3.8 Vergleichende Evaluation

Im Folgenden wird eine vergleichende und klassifizierende Zusammenfassung der beschriebenen MDWE-Ansätze gegeben. Dabei liegt der Fokus auf technischen Charakteristika mit Bezug zu der modellgetriebenen Softwareentwicklung, und weniger auf sprachlichen Ausdrucksmittel für die konzeptionelle Modellierung von Webapplikationen. Die Analyse mündet in eine Bewertung, zu welchem Grad die Ansätze konform zu den Merkmalen der MDA sind. Der in dieser Arbeit vorgestellte Ansatz wird anschließend vor dem Ergebnis der Bewertung motiviert und in das Klassifikationsschema eingeordnet.

Die MDWE-Ansätze werden danach klassifiziert, für welche *Plattformen* Webapplikationen in welcher *technischen Sphäre* spezifiziert werden, und mittels welcher *PSMs* sowie *M2T*- und *M2M*-Techniken die Generierung durchgeführt wird (vgl. Tabelle 1). Bzgl. der

[27] `http://fornax.itemis.de/`

Plattformen werden solche genannt, die durch einen Codegenerator konkret unterstützt werden. Dies schließt aber nicht aus, dass potentiell Transformationsdefinitionen für andere Plattformen spezifiziert werden könnten. Die Plattformbezeichnungen reflektieren sowohl die unterstützten Programmiersprachen als auch die üblicherweise in deren Rahmen verwendeten Frameworks. Die *technische Sphäre* bezeichnet den Beschreibungsrahmen oder das Metametamodell und determiniert die Auswahl an Werkzeugen für den MDWE-Ansatz, wie z. B. Transformationssprachen. Als wichtigste Sphären sind XML, MOF und UML als Spezialfall eines Metamodells in der technischen Sphäre von MOF zu nennen. Mit der Kategorie *PSM* wird erfasst, zu welchem Grad der jeweilige Ansatz Plattformspezifika im Modell abbildet. Die Unterschiede sind graduell, da sie von einer Repräsentation sämtlicher Details der Implementierung, über abstrahierende Ausschnitte für als relevant befundene Teile der Implementierung bis hin zu einem vollständigen Verzicht reichen. Die Kategorien *M2T* und *M2M* fassen zusammen, ob und in welcher Form bzw. welcher Transformationssprache die Codegeneratoren bzw. M2M-Transformationsdefinitionen formuliert sind.

Techniken, die keine zentrale Rolle bei der praktischen Anwendung des jeweiligen Ansatzes bzw. bei der Implementierung des MDWE-Werkzeuges spielen oder mittlerweile substituiert wurden, sind in der tabellarischen Darstellung des Klassifiationsschemas durch runde Klammern gekennzeichnet. Dies betrifft z. B. Transformationsdefinitionen, welche im Umfeld eines MDWE-Ansatzes als Proof-of-Concept entwickelt wurden, aber nicht in das jeweilige Kernwerkzeug integriert wurden.

Die MDWE-Ansätze weisen deutliche Unterschiede bzgl. der Kategorieausprägungen auf. Sämtliche Ansätze bieten Modellierungssprachen für PIMs, die zudem wegen der durchgängigen Darstellung von Navigations- und Präsentationsinhalten Ähnlichkeiten aufweisen. Als technische Sphäre für die PIMs wird tendenziell die UML als Ausgangspunkt für die Metamodelle der Modellierungssprachen verwendet. Entsprechend der technischen Sphäre der UML wirkt sich dies standardisierend auf die M2M-Transformationsdefinitionen aus, bei denen QVT als Transformationssprache dominierend ist. Die eingesetzten Techniken für die Codegeneratoren der MDWE-Ansätze sind heterogen und nicht standardisiert, da für den OMG-Standard MOFM2T für M2T-Transformationsdefinitionen [Obj08b] erst seit Mitte des Jahres 2010 mit Acceleo 3 eine Implementierung vorliegt. Bzgl. der Plattformen dominiert Java EE mit diversen Framework-Kombinationen. Als alternative Plattformen werden solche mit objektorientierten Programmiersprachen wie z. B. .NET bei OOWS genannt, generell aber nicht mehrere unterschiedliche Plattformen adressiert. In der Webentwicklung verbreitete Skriptsprachen wie PHP und Ruby sind unterrepräsentiert. Die Verwendung von PSMs ist heterogen, da PSMs entweder fehlen (WebML, OOWS) oder in Form abstrahierter UML-Modelle (WAE, UWE, OO-H) vorliegen, die Plattform- und Fra-

Name, Tool	Zeitraum	Plattformen	technische Sphäre	PSM	M2T	M2M
WAE	1999-2002	JSP [Con02, S. 315-376]	UML	abstrakt, UML	konzeptionell [Con02, S. 315-376]	nein
WebML, WebRatio	seit 2000	Java EE [Web11, S. 6]	XML/DTD [BCFM07, S. 235-238], (UML [MFV07]), (MOF [SWK+07])	nein	Groovy, (XSLT [BCFM07, S. 235-238])	nein
UWE, MagicUWE, (ArgoUWE)	seit 1999	Java EE [KK09]	UML [KK08]	abstrakt, UML	hartcodiert, semiautomatisch	QVT, ATL, OCL [KKZB08, S. 180-185] [Koc06, S. 6]
OO-H, VisualWADE	seit 2000	PHP [Góm04], GWT [MGPD08]	UML, XML [Góm04], (MOF [MGPD08])	abstrakt	hartcodiert	QVT [MGPD08], hartcodiert
OOWS, OlivaNOVA	seit 2001	Java EE, .NET, PHP für GUI [VVFP07]	UML, MOF [VVFP07, S. 7]	nein	Xpand [VVFP07, S. 11], (MOF-Script [TP06])	QVT [TP06], (AGG [VFP05])
WASL	seit 2008	Java EE, PHP, Python	MOF	detailliert, DSLs	Xpand	QVTO

Tabelle 1: MDSD-Charakteristika der MDWE-Ansätze

meworkdetails wie z. B. Konfigurationsdateien oder prozeduralen Quelltext nicht detailliert repräsentieren.

Die MDWE-Ansätze können als MDSD-konform klassifiziert werden, falls das MDSD als Softwareentwicklungsmethode zur Generierung lauffähiger Software aus formalen Modellen verstanden wird (vgl. Kapitel 2). Lediglich im Fall von WAE ist die M2T-Transformationsdefinition von konzeptioneller Natur, aber prinzipiell technisch realisierbar. Im Vergleich dazu ist die Überprüfung der MDWE-Ansätze auf Konformität zu den Prinzipien der MDA mit restriktiveren Kriterien verbunden. Charakteristisch für die MDA ist die multi-perspektivische Spezifikation und Generierung von Systemen mittels standardisierter Techniken aus der technischen Sphäre von MOF bzw. der UML (vgl. Kapitel 2.5). Insbesondere die Systemspezifikation durch PSMs sowie die Verwendung von Techniken aus der technischen Sphäre von MOF bzw. UML werden wie beschrieben durch die MDWE-Ansätze unterschiedlich stark erfüllt, sodass WebML tendenziell weniger und UWE, OO-H und OOWS tendenziell stärker MDA-konform sind.

Der Verzicht der beschriebenen MDWE-Ansätzen auf PSMs mit detailliertem Bezug zu Plattformkonzepten induziert jedoch eine Abstraktionslücke zwischen den PIMs und den Implementierungen. Diese Abstraktionslücke zwischen der Modellwelt des PIM und der Implementierungswelt des Quelltexts erzeugt Probleme sowohl bei der Modellierung als auch bei der Entwicklung von Transformationsdefinitionen. Erstens ist ohne detaillierte PSMs die Repräsentation von relevanten Plattformkonzepten wie z. B. prozeduralem Quelltext, GUI-Templates, relationalen Schemata und XML- oder YAML-Konfigurationsdateien im Modell nicht möglich und demnach innerhalb die Modellwelt nicht gestaltbar. Zweitens wird für den Modellierer der Lernaufwand erhöht, da die Verbindung zwischen den Modellinhalten als Ursache und der Implementierung als Wirkung der Transformation nicht in Form eines strukturähnlichen Modells, sondern durch die Transformationsdefinition als vergleichsweise komplexe Abbildung expliziert wird. Neben der fehlenden Intuitivität für den Modellierer wird dabei drittens auch die Komplexität für den Entwickler der M2T-Transformationsdefinition erhöht, da die im MDSD-Kontext verwendeten Templatesprachen üblicherweise agnostisch bzgl. der Syntax der Zielprogrammiersprachen sind, sodass während der Definition der textuellen Templates Syntaxfehler nicht gekennzeichnet werden. Dieses Problem liegt bei M2M-Transformationsdefinitionen nicht vor, sodass eine Verlagerung der Komplexität in vorgelagerte Modelltransformationen hilfreich ist. Viertens wird durch anreichernde M2T-Transformationsdefinitionen die Problemspezifität des Werkzeugs erhöht und damit ein universeller Einsatz des Generators für unterschiedliche Applikationstypen verhindert.

Für eine Reduktion der Abstraktionslücke bieten sich im Rahmen des MDSD zwei

Möglichkeiten. Erstens kann das Abstraktionsniveau der Zielplattform durch abstrahierende Laufzeitkomponenten des Generators angehoben werden, was aber Interdependenzen zwischen dem Generat und dem Generator erzeugt, die nicht immer gewünscht sind. Die zweite Variante ist, das Abstraktionsniveau des PSMs abzusenken, was aber mit dem Ziel klarer Modellaussagen durch kompakte Modelle konfligiert. Die Verwendung des PIM zur Abstraktion und des PSM zur Explikation löst diesen Zielkonflikt, sodass ein strukturähnliches PSM den Vorteil einer geringen Abstraktionslücke mit der Unabhängigkeit von Generatorkomponenten zur Laufzeit des Softwaresystems bietet. Zur Erstellung solch detaillierter PSMs muss die Modellierungssprache die Repräsentation der Plattformkonzepte ermöglichen. Dies betrifft sowohl die Elemente des Metamodells der Modellierungssprache, die als Typen die Plattformkonzepte repräsentieren, als auch die Beziehungen zwischen den Metamodellelementen. Letztere müssen konform zu der Syntax der Zielplattform sein, um die gewünschte Strukturähnlichkeit der PSMs zu ermöglichen.

Ein Ergebnis der Standardisierungsbemühungen im MDWE ist die vorherrschende Verwendung der UML in den verschiedenen MDWE-Ansätzen. Deren Dominanz entspringt dem Wunsch nach vereinheitlichten Methoden für die Softwareindustrie bzw. einer Eindämmung des unübersichtlichen Zoos von DSLs. Dem Nachteil einer ungenügenden Interoperabilität von DSLs wurde in Form von MOF mittlerweile aber ein standardisierter Rahmen entgegengestellt, sodass bei der Entscheidung zwischen UML und DSL als Modellierungssprache für die PSMs beide Alternativen zu erwägen sind.

Für die UML spricht im Rahmen des MDWE zunächst die Verfügbarkeit ausgereifter Entwicklungswerkzeuge mit UML-Unterstützung. Vorteilhaft ist zudem, dass die UML aufgrund ihrer Verbreitung für die Modellierer als bekannt vorausgesetzt werden kann und Erfahrungen zur Anwendung in Softwareentwicklungsprojekten existieren, sodass der Einarbeitungsaufwand in das UML-Profil einer Web-Modellierungssprache tendenziell geringer ausfällt, als bei einer eigenständigen DSL.

Im Unterschied dazu weist die UML für die Modellierung von Webapplikationen mittels Navigations- und Präsentationsmodellen aber auch einige Nachteile auf. Erstens treten Abbildungsdefekte bei der Nutzung der UML für das Web Engineering auf, da die Metamodellelemente der UML wie beschrieben semantisch nicht vollständig adäquat für die Modellierung von relevanten Plattformkonzepten wie z. B. prozeduralem Quelltext, GUI-Templates, relationalen Schemata und XML- oder YAML-Konfigurationsdateien sind. Die Minimierung der infolgedessen entstehenden Abstraktionslücke ist mittels Erweiterungen der UML um entsprechende Metamodellelemente nur bedingt möglich, da mit UML-Profilen (vgl. [Obj10c, S. 669-697]) zusätzliche Sprachelemente konservativ lediglich als Erweiterungen bestehender Metamodellelemente hinzugefügt werden, zu denen aus Gründen

der Klarheit ein semantischer Bezug bestehen sollte. Die Abstraktionslücke konterkariert zweitens den reduzierten Lernaufwand für die UML, da aus ihr ein entsprechend erhöhter Lernaufwand für das Wissen der Modellierer über die M2T-Transformationsdefinitionen resultiert. Drittens weisen UML-basierte Ansätze das Problem auf, dass UML-Modelle abhängig vom Anwendungsszenario im Vergleich zu DSL-basierten Modellen weniger kompakt sind, da sie Semantik durch eine Kombination generischer sprachlicher Ausdrucksmittel der UML formulieren (vgl. [MFV06, S. 8]). Viertens ist ein Problem, dass die Mächtigkeit der UML zwar durch OCL-Constraints eingeschränkt werden kann, sich dies aber in nachträglichen Modellvalidierungen niederschlägt. Wünschenswert wäre dagegen bereits während der Modellierung eine proaktive Erzwingung valider Modelle durch eine dynamisch-kontextuelle Einschränkung der Palette der Sprachelemente, um die Modellierung zu vereinfachen. So sollte z. B. bei der WAE beim Hinzufügen einer *«submit»*-Assoziation zum Modell bereits die Palette der Sprachelemente für die plausiblen Modellelemente an den Assoziationsenden auf *Client*- und *ServerPages* eingeschränkt werden. Weitere Ausführungen zu Stärken und Schwächen der UML im Vergleich zu WebML als exemplarischer DSL finden sich zudem in [MFV07, S. 14-20].

Im Unterschied zur UML weisen DSLs eine größere Flexibilität bei der Wahl der Metamodellelemente und Beziehungen zwischen diesen auf, sodass adäquate Modellierungssprachen für strukturähnliche Modelle erstellt werden können.

Nachteilig ist bei DSLs der initiale Aufwand zur Entwicklung der Modellierungssprache, mit dem ein erhöhter Lernaufwand für den Modellierer einhergeht. Dies wird aber durch einen geringeren Aufwand für die Erstellung und Kontrolle der M2T-Transformationsdefinition kompensiert, da ein geringeres Abstraktionsniveau die Transformationsdefinition vereinfacht und das Modell intuitiv verständlich macht. Ein weiterer Nachteil von DSLs ist schließlich die mangelnde Standardisierung des Metamodells, die sich aber durch eine Standardisierung des Metametamodells z. B. in Form von MOF nicht auf die Werkzeugunterstützung in Form von Modellierungsumgebungen, Transformationsengines und Serialisierungsformaten auswirkt.

Als alternativer MDWE-Ansatz wird im Folgenden das WASL-Generatorframework vorgestellt, welches die Generierung lauffähiger datenbasierter Webapplikationen für verschiedene Webplattformen unterstützt, auf MDA-Standards basiert und die Abstraktionslücke zwischen dem PSM und der Implementierung durch plattformspezifische DSLs minimiert (vgl. Tabelle 1).

4 WASL-Generatorframework

4.1 Einführung

Das WASL-Generatorframework ist ein modellgetriebenes Softwareentwicklungswerkzeug zur automatisierten Generierung datenbasierter Webapplikationen. Der Fokus liegt dabei auf Web-basierten betrieblichen Anwendungssystemen (vgl. [SH04, S. 326 ff]), deren Kernfunktionalität die Haltung und Verarbeitung betrieblicher Datensätze umfasst. Charakteristisch für das Generatorframework sind die Anwendung von Prinzipien sowie Techniken der MDA, die unterstützten Plattformen und die Art der PSMs. Das Generatorframework umfasst mit der *Web Application Specification Language*-Familie (WASL) mehrere Modellierungssprachen zur Erstellung von CIMs, PIMs und PSMs. Mit diesen ermöglicht es die Modellierung von Webapplikationen für die Webplattformen Java EE, Python und PHP und deckt somit drei praxisrelevante Plattformen ab, die bzgl. ihrer Programmiersprachen unterschiedlichen Paradigmen folgen. Um die Abstraktionslücke zwischen den PSMs und den Implementierungen zu minimieren, unterstützt das WASL-Generatorframework strukturähnliche PSMs durch plattformspezifische DSLs, die individuell für jede Plattform mit ihren Programmiersprachen und Frameworks definiert sind. Die Codegeneratoren zur Transformation der PSMs in Implementierungen für die Plattformen sind aufgrund der Modularität des Generatorframeworks jeweils selbstständig einsetzbar.

Die Prinzipien der MDA werden verfolgt, indem Webapplikationen aus verschiedenen Perspektiven (vgl. [Obj03, S. 2-5]) durch die Modelltypen CIM, PIM und PSM spezifiziert werden, und aus diesen automatisiert lauffähige Implementierungen erzeugt werden können. Zu diesem Zweck sind die Metamodelle der WASL-Modellierungssprachen durch Transformationsdefinitionen verknüpft, sodass eine Systemspezifikation mit einem hohen Abstraktionsgrad schrittweise in Implementierungen für verschiedene Plattformen überführt werden kann. Technisch finden für die Spezifikation der Metamodelle und Transformationsdefinitionen zudem MDA-konforme und standardisierte Techniken Verwendung.

Die Architektur des WASL-Generatorframeworks ist in Form lose gekoppelter Metamodelle und Transformationsdefinitionen modular strukturiert, um eine Zerlegung des Gesamtsystems für projektspezifische Anforderungen zu ermöglichen. Die Einzelkomponenten können zur Konfiguration des Frameworks durch die Adaptionsmechanismen der Spezialisierung und Aggregation (vgl. [BDK04]) kombiniert bzw. erweitert werden. Dies ermöglicht es, einerseits Metamodelle und Transformationsdefinitionen des Frameworks zu erweitern und andererseits bereitgestellte sowie eigenentwickelte Metamodelle und Transformationsdefinitionen flexibel zusammenzustellen. Das WASL-Generatorframework trägt somit dem Sachverhalt Rechnung, dass die Geschäftslogik betrieblicher Anwendungssoftware abhängig

von der benötigten Funktionalität viele unterschiedliche domänen-, technik- und lösungsspezifische Formen annehmen kann, die ex-ante nicht festzulegen sind.

Der Entwicklung des Generatorframeworks liegt die Fragestellung zugrunde, wie ein Generatorframework gestaltet sein muss, um akzeptabel für die diversen Entwickler-Communities zu sein. Den bisherigen MDWE-Ansätzen ist gemein, dass sie den Charakter von Forschungsprojekten aufweisen und in den verschiedenen Entwickler-Communities abgesehen von Referenzprojekten nur vereinzelt eingesetzt werden. In der Praxis finden sich breite Anwendungen des MDSD vielmehr in Form generativer Komponenten von Web-Entwicklungsframeworks. Beispiele dafür sind das Scaffolding von Ruby on Rails und Klassengeneratoren diverser objekt-relationaler Mapper (ORM), die zwar plattformspezifisch einzelne Phasen im Softwareentwicklungsprozess unterstützen, aber keine plattform- und frameworkübergreifenden Merkmale aufweisen. Die Charakteristika des WASL-Generatorframeworks sind das Ergebnis einer Fokussierung auf diese Entwickler-Communities. So werden etablierte Web-Plattformen individuell durch PSM-Modellierungssprachen adressiert, die unabhängig voneinander für die jeweilige Plattform für Modellierungs- und Generierungszwecke verwendet werden können. Durch die Minimierung der Abstraktionslücke mittels strukturähnlicher PSMs sollen diese zudem generisch einsetzbar für unterschiedliche Applikationstypen sein und der Lernaufwand für nicht MDA-affine Webentwickler reduziert werden. Als verbindendes Element zwischen den verschiedenen Plattformen dient nachgelagert im Sinne der MDA einerseits die PIM-Modellierungssprache des Generatorframeworks und andererseits die einheitliche Technologiebasis in Form von MOF.

Im Unterschied zu dem Top-Down-Vorgehen der geschilderten MDWE-Ansätze verfolgt das WASL-Generatorframework einen Bottom-Up-Ansatz, nach dem die Erstellung von PIMs nicht obligatorisch ist, sondern optional zur automatisierten Erzeugung der PSMs durchgeführt wird. Die zugrunde liegende Intention ist eine generelle Etablierung von PSM-Modellierungssprachen als einheitliche und gemeinsame Abstraktionsschicht für verschiedene Arten von PIMs mit deren jeweiligen Anwendungstypen und -szenarien. Als konkrete Beispiele werden im Folgenden die verschiedenartigen PIM-Modellierungssprachen *WASL Generic* (vgl. Kapitel 4.6.2) und *TaskUI Generic* (vgl. Kapitel 6.4) vorgestellt, die durch Transformationsdefinitionen mit denselben PSM-Modellierungssprachen verbunden sind. Ausgeweitet auf andere MDWE-Ansätze bietet diese Herangehensweise die Möglichkeit, diesen über eine einheitliche PSM-Schicht ein gemeinsames Fundament zu bieten und die Integrationsabsicht einiger MDWE-Ansätze [WSSK07, VKC+07] zu unterstützen. Da die meisten MDWE-Ansätze mit ihren UML-basierten PIMs in der technischen Sphäre von MOF verortet sind, können zu diesem Zweck die Transformationstechniken von MOF für die PIM-zu-PSM-Transformationen genutzt werden. Ähnlich zu WebSA [MGS07, MG06a]

kann das WASL-Generatorframework somit als komplementäre Erweiterung der MDWE-Ansätze verwendet werden. Im Unterschied zu WebSA liegt der Fokus aber nicht auf der Modellierung der Architektur im PIM, sondern auf Modellierungssprachen und Codegeneratoren für PSMs.

4.2 Anforderungen

Der Architektur und Implementierung des Generatorframeworks liegen konzeptionelle und technische Anforderungen zugrunde, mit denen die Relevanz und Anwendbarkeit des Frameworks für die Praxis sichergestellt werden soll. Diese werden im Folgenden beschrieben und sind nach ihrem Bezug zu den Schichten M1 bis M3 in Anforderungen unterteilt, die sich auf die Techniken des Generatorframeworks (M3), auf die Spezifikation von Modellierungssprachen und Transformationsdefinitionen (M2) sowie auf die Modellierung (M1) beziehen:

Adressierung relevanter Zielplattformen (M2): Die praktische Relevanz von Softwareentwicklungswerkzeugen wird restringiert durch die Relevanz der Softwareplattformen, für die mit den Werkzeugen Software entwickelt werden kann. Aus diesem Grund sind durch die Modellierungssprachen des Generatorframeworks Plattformen zu adressieren, die etabliert für die Ausführung von Webapplikationen sind. Dies schließt neben Java EE die verbreiteten Webplattformen bzw. deren Programmiersprachen Python und PHP ein.

Transformierbarkeit sämtlicher Metamodellelemente (M2): Sämtliche der Metamodellelemente der PIM-Sprachen sind über Transformationsdefinitionen auf Metamodellelemente der PSM-Sprachen der Webplattformen abzubilden. Mit dieser Restriktion soll der Entwurf beliebiger konzeptioneller CIM- und PIM-Modellierungssprachen ohne Problemlösungsbezug verhindert werden, um den Hauptzweck des Generatorframeworks, nämlich die Generierung von Implementierungen, sicherzustellen. Methodisch wird die Repräsentation von PIM-Konzepten durch PSM-Konzepte mittels eines Vorgehens nach dem Bottom-Up-Prinzip gewährleistet (vgl. Kapitel 4.4).

Strukturähnliche PSMs (M2, M1): Die Ausdrucksmittel der Zielplattformen sowie deren typischen Anwendungs- und Entwurfsmuster sind in den PSM-Metamodellen möglichst strukturähnlich zu repräsentieren. Durch die Strukturgleichheit oder zumindest -ähnlichkeit zwischen PSM und Quelltext und der daraus folgenden geringen Abstraktionslücke sollen organisatorische und technische Vorteile erzielt werden.

Organisatorisch soll die detaillierte Repräsentation identitätsstiftender Merkmale und Zusammenhänge der Plattformen im Modell Akzeptanz bei Modellierern und Ent-

wicklern generieren. Indem Plattformkonzepte bei der Erstellung und Nutzung von PSMs aufgegriffen werden, wird erstens der Einarbeitungsaufwand für Modellierer und Entwickler zur Nutzung des Generators reduziert und zweitens der Bezug bzw. bisweilen die Loyalität von Entwicklern zu den von ihnen präferierten Plattformen berücksichtigt.

Technische Vorteile ergeben sich, da PSM-Metamodellelemente als Zieltypen für typsichere PIM-zu-PSM-Transformationsdefinitionen verwendet werden können. M2M-Transformationsdefinitionen zeichnen sich im Unterschied zu M2T-Transformationsdefinitionen durch eine Typisierung der Elemente des Generats aus (vgl. Kapitel 2.8). Insofern ist es sinnvoll, den Abstraktionssprung vom PIM über das PSM zum Quelltext generell auf M2M-Transformationsdefinitionen zu verlagern. Indem das PSM strukturgleich zum Quelltext ist, ist die Komplexität des PSM-zu-Quelltext-Schritts gering, die Komplexität des PIM-zu-PSM-Schritts dagegen hoch.

Modularität (M2): Das Generatorframework soll aus Modulen bestehen, die getrennt voneinander funktionieren und für verschiedene Applikations- und Plattformtypen individuell zusammengestellt werden können. Jeder Programmiersprache und jedem Framework der Ausführungsumgebung soll ein Modul zugeordnet werden. Die aus den Zusammenstellungen resultierenden Generatoren sollen autonom funktionieren, sodass z. B. ein aus dem Generatorframework komponierter Generator für PHP ohne die Generatormodule für Java EE lauffähig ist, aber auch zusammen mit diesen funktionsfähig wäre. Dazu sollen die Module untereinander lose gekoppelt sein.

Erweiterbarkeit (M3): Die Metamodellierungstechnik soll nachträgliche Erweiterungen der Metamodelle und Transformationsdefinitionen unterstützen, um projektspezifische Anpassungen zu ermöglichen.

Agnostisches Generat (M2): Dem Generat soll die Herkunft aus dem Generator nicht anzumerken sein. Dies bezieht sich sowohl auf den Inhalt der generierten Dateien als auch auf deren Verortung im Dateisystem. Letztere soll entsprechend zu der individuellen Verzeichnisstruktur des Entwicklungsprojekts in beliebiger Form im Modell festgelegt werden können. Die freie Verortung des Generats im Dateisystem bedingt, dass der Generator sowohl komplette Entwicklungsprojekte generieren können soll, als auch seine Inhalte in beliebige Entwicklungsprojekte bzw. deren Verzeichnisstruktur injizieren können muss. Mit dieser Restriktion werden sowohl Erweiterungen der Laufzeitumgebung um generatorspezifische Programmierschnittstellen als auch vom Generator vordefinierte Dateisystemstrukturen ausgeschlossen.

Technologien (M3): Für die Implementierung des Generatorframeworks sind standardisierte oder zumindest industriell etablierte Technologien mit ausgereiften Implementierungen zu wählen, deren dauerhafte Wartung sichergestellt ist. Aus praktischen Gründen soll zudem die Auswahl auf Implementierungen beschränkt werden, die als Erweiterungen von Eclipse i.V.m. EMF entwickelt und installiert werden können.

4.3 Architektur

Das WASL-Generatorframework ist als Schichtenarchitektur strukturiert, deren Schichten mit den Modelltypen CIM, PIM und PSM bzw. deren verschiedenen Abstraktionsstufen korrespondieren (vgl. Abbildung 22). Den Modelltypen sind Modellierungssprachen aus der WASL-Sprachfamilie zugeordnet, welche durch Transformationsdefinitionen miteinander verbunden sind.

Für CIMs auf der obersten Schicht fungiert die Sprache *WASL Data*, deren Sprachelemente die Datenmodellierung von Domänenkonzepten abdecken (vgl. Kapitel 4.6.1). PIMs auf der darunter eingeordneten Abstraktionsschicht werden mit der Sprache *WASL Generic* formuliert, die Sprachelemente für die plattformunabhängige Spezifikation von Webapplikation hinsichtlich der Datenhaltung, Navigation, Geschäftslogik und Präsentation umfasst (vgl. Kapitel 4.6.2). Die plattformspezifischen Implementierungen werden als PSMs mit den Modellierungssprachen *WASL JavaEE*, *WASL PHP* und *WASL Python* modelliert (vgl. Kapitel 4.6.3).

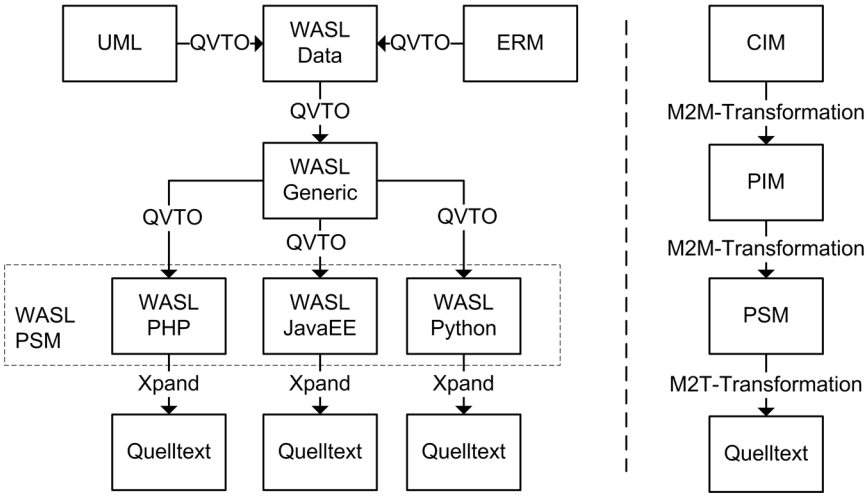

Abbildung 22: Architektur des WASL-Generatorframeworks

Die Modellierungssprachen für diese Modelltypen werden durch Metamodelle mit Ecore als Metasprache spezifiziert. Für *WASL Data* und *WASL Generic* existiert jeweils ein gleichnamiges Metamodell. Die Modellierungssprachen *WASL JavaEE*, *WASL PHP* und

WASL Python sind dagegen jeweils das Resultat der Zusammenstellung einer Vielzahl von Metamodellen, die allgemein das Dateisystem und im Speziellen die Programmiersprachen und Frameworks der Plattformen adressieren. Die Unterteilung in diese drei plattformspezifischen Modellierungssprachen ist lediglich als eine Option zu verstehen, da die plattformspezifischen Metamodelle entsprechend der projektspezifischen Anforderungen kombiniert werden können. Die Kombinierbarkeit der Metamodelle für die Programmiersprachen und Frameworks wird durch ein gemeinsames Metamodell für das Dateisystem gewährleistet, das Teil jeder Metamodellkombination ist und die anderen plattformspezifischen Metamodelle verbindet. Die typischen Modellierungssprachen bzw. Metamodellkombinationen *WASL JavaEE*, *WASL PHP* und *WASL Python* sowie die weiteren möglichen Kombinationen der plattformspezifischen Metamodelle werden im Folgenden unter der Bezeichnung *WASL PSM* zusammengefasst.

Das WASL-Generatorframework verfügt über Transformationsdefinitionen, die unidirektional Transformationen vom CIM über das PIM und das PSM zum Quelltext ermöglichen. *WASL Data*-Modelle können entweder manuell angelegt werden, oder aus UML-Klassendiagrammen sowie ER-Modellen erzeugt werden, um anschließend automatisiert in prototypische *WASL Generic*-Modelle transformiert zu werden. Mit der Transformation ist eine Anreicherung um zusätzliche Semantik verbunden, die in den Transformationsdefinitionen spezifiziert ist. So werden bei der Transformation eines *WASL Data*-Modells zu einem *WASL Generic*-Modell automatisch Details zur Repräsentation von CRUD-Operationen (Create, Read, Update, Delete) aus dem Datenmodell abgeleitet und dem *WASL Generic*-Modell hinzugefügt. *WASL Generic*-Modelle werden nach einer optionalen Verfeinerung dieser Details ebenfalls automatisiert in das PSM für die jeweilige Zielplattform transformiert und aus diesem der Quelltext generiert. Die M2M-Transformationsdefinitionen (vgl. Kapitel 4.7) sind in der Transformationssprache QVTO (vgl. Kapitel 2.8.3) formuliert, die M2T-Transformationsdefinitionen (vgl. Kapitel 4.8) in der Templatesprache Xpand (vgl. Kapitel 2.7.2).

Die Codegeneratoren sind selbstständig für die jeweils adressierte Webplattform und unabhängig von den restlichen Komponenten des Generatorframeworks nutzbar. Dies ist relevant bei Softwareentwicklungsprojekten, die lediglich einen einzelnen Codegenerator als eigenständiges Werkzeug für eine bestimmte Webplattform benötigen und die keinen Bedarf für eine plattformunabhängige Modellierung aufweisen. Aufgrund der Entkoppelung der Codegeneratoren von bestimmten plattformunabhängigen Modellierungssprachen können diese somit potentiell für andere MDWE-Modellierungssprachen wiederverwendet werden.

Da die Metamodelle mit Ecore als Metasprache spezifiziert sind, können die Meta-

modellelemente zudem mittels des Spezialisierungsmechanismus von Ecore verfeinert und somit die Metamodelle erweitert werden. Die Möglichkeit zur Spezialisierung erstreckt sich sowohl auf die Metamodellelemente für spezielle Konzepte wie z. B. Java-Methoden, als auch auf grundlegende Konzepte wie z. B. Dateitypen. Insofern kann das Generatorframework auf verschiedenen Granularitätsstufen sowohl im Detail um bestimmte Methoden und Klassen, als auch im Ganzen um Frameworks oder Programmiersprachen erweitert werden. Im Vergleich zu den festen Diagrammtypen der UML weist WASL PSM mit der Abbildung des Dateisystems zwar einen geringeren Abstraktionsgrad auf, bietet aber eine größere Flexibilität hinsichtlich der Erweiterbarkeit.

4.4 Forschungsmethode

Das WASL-Generatorframework ist das Resultat konstruktionsorientierter Forschungen zum generellen Nachweis der prinzipiellen Realisierbarkeit von Konzepten durch die Entwicklung und den Test von Prototypen (vgl. [LF06, S. 35 f.]). In diesem Kontext ist es ein Artefakt im Sinne der gestaltungsorientierten Wissenschaft bzw. Design Science. Letztere zeichnet sich durch eine Abfolge der Phasen der Konstruktion (Build) von Artefakten und deren Evaluation (Evaluate) aus [MS95, HMPR04], wobei Artefakte *Konstrukte*, *Modelle*, *Methoden* oder *Instanziierungen* sein können [MS95, S. 253]. Im Rahmen dieser Artefakt-Kategorien sind die Modellierungssprachen und Transformationsdefinitionen des Generatorframeworks als Konstrukte zu klassifizieren, das Generatorframework in Gänze als ausführbare Implementierung zur Erfüllung konkreter Aufgaben der Softwareentwicklung dagegen als Instanziierung.

Die Entwicklung des Generatorframeworks verlief über mehrere Projektabschnitte hinweg, in denen initial plattformspezifische Referenzimplementierungen der Prototypen entwickelt wurden und anschließend mit sukzessive zunehmendem Abstraktionsgrad Metamodelle und Transformationsdefinitionen für die Schichten des PSM, des PIM und final des CIM abgeleitet wurden. Im Detail befassten sich die Projektabschnitte 1-3 mit der Entwicklung von Prototypen für die verschiedenen Zielplattformen, die Projektabschnitte 4-6 mit der Ableitung von Metamodellen und Transformationsdefinitionen für die PSM-Schicht, Abschnitt 7 analog mit der PIM-Schicht und Abschnitt 8 mit der CIM-Schicht (vgl. Abbildung 23). Ab Projektabschnitt 4 bestand jeder Projektabschnitt aus einer iterativen Abfolge von Konstruktions- und Evaluationsphasen, die primär der Erstellung von Metamodellen sowie sekundär der Implementierung von Transformationsdefinitionen dienten. Die Motivation für das Bottom-Up-Vorgehen war die Absicht, dass der Generator während der Entwicklung stets die Fähigkeit haben sollte, sämtliche der mit WASL-Sprachen fomulierbaren Modelle in Quelltext überführen zu können.

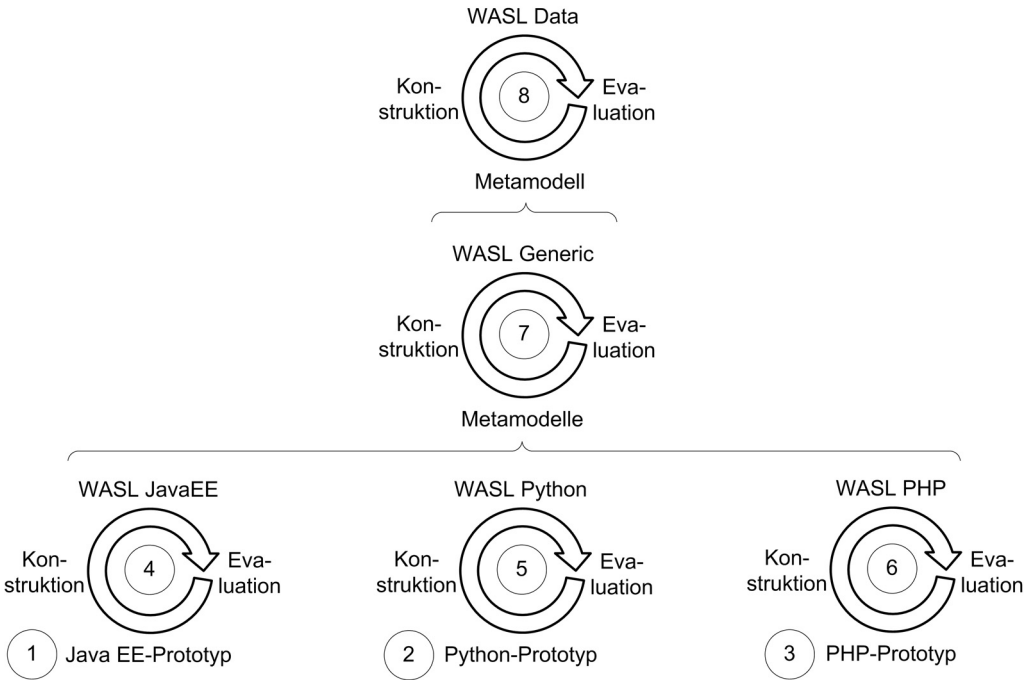

Abbildung 23: Forschungsmethode zur Entwicklung des WASL-Generatorframeworks

In den Projektabschnitten 1-3 wurden Prototypen in Form datenorientierter Webapplikationen mit Funktionalität zur Verwaltung von Kunden- und Veranstaltungsdaten für die Plattformen Java EE, Python und PHP konstruiert. Die Applikationen unterstützen grundlegende Datenoperationen zum Anlegen (Create), Anzeigen (Read), Modifizieren (Update) und Löschen (Delete) von Datensätzen (CRUD). Die prototypischen Webapplikationen dienten als Referenzimplementierungen für die verschiedenen Zielplattformen und vereinfachten die nachfolgende Entwicklung der Codegeneratoren (vgl. [SVEH07, S. 219 f.]). So bietet jeder Prototyp Anhaltspunkte für ein strukturiertes Vorgehen zur Entwicklung der entsprechenden Metamodelle und M2T-Transformationsdefinition und lässt als Referenzpunkt Vergleiche zum Generat bzw. Rückschlüsse über den Reifegrad des Codegenerators zu. Als Alternative zu der Eigenentwicklung von Prototypen können auch bereits existierende Prototypen mit einer passenden Funktionalität, Frameworkauswahl und verfügbarem Quelltext dienen, was aber aufgrund der Konstellation der zu unterstützenden Plattformen für das Forschungsvorhaben nicht praktikabel war.

Die Projektabschnitte 4-6 befassten sich jeweils in einer iterativen Abfolge von Konstruktions- und Evaluationsphasen mit der Entwicklung der Metamodelle für die Modellierungssprachen *WASL JavaEE*, *WASL Python* und *WASL PHP* sowie der entsprechenden Codegeneratoren. Die Konstruktion der Metamodelle verlief initial für die Programmiersprachen und anschließend für Entwurfsmuster und Frameworks. Bei der Erstellung von

Metamodellen aus Frameworks wurde mit solchen für Datenhaltungszwecke begonnen, da diese aufgrund der Schichtenarchitekturen der Prototypen am wenigsten Interdependenzen zu den restlichen Frameworks aufwiesen. Die Konstruktion von Metamodellen für Konfigurationsdateien verlief im Fall von solchen für XML-Dokumente semiautomatisiert mittels eines Metamodellgenerators von EMF, der aus XML-Schemadefinitionen serialisierte Metamodelle generiert [SBPM08, S. 197-259][04.04b]. Parallel zur der Metamodellerstellung verlief die Konstruktion von M2T-Transformationsdefinitionen, die aufgrund der geringen Abstraktionslücke mittels des eigenentwickelten Templategenerators *Metagen* (vgl. Kapitel 5) ebenfalls partiell automatisiert durchgeführt wurde. Im Anschluss erfolgte jeweils eine analytische Evaluation der Modifikationen (vgl. [HMPR04, S. 85-87]), indem als Evaluationsmethode ein Vergleich zwischen dem Generat und der Referenzimplementierung durchgeführt wurde und eine Modifikation als zulässig deklariert wurde, falls die Unterschiede zwischen dem Generat und der Referenzimplementierung im Vergleich zu der vorherigen Iterationen reduziert wurden. Für die Generierung wurden vorab manuell PSM-Referenzmodelle erstellt, die als Eingabemodelle für die Transformation dienten. Jeder der drei Projektabschnitte wurde nach mehreren Iteration finalisiert, sobald der Quelltext vollständig generiert werden konnte und die Unterschiede zwischen dem Generat und dem Prototyp einen akzeptablen Stand erreicht hatten.

In Projektabschnitt 7 wurden das Metamodell der Modellierungssprache *WASL Generic* sowie die M2M-Transformationsdefinitionen zu den Metamodellen der Modellierungssprachen *WASL JavaEE*, *WASL Python* und *WASL PHP* konstruiert und evaluiert. Die Konstruktion verlief induktiv durch Generalisieren der Gemeinsamkeiten der drei PSM-Sprachen unter der Prämisse, dass das Metamodell von *WASL Generic* die plattformunabhängige Modellierung der Daten, Navigation, Logik und Präsentation von Webapplikationen abdecken soll. Die Evaluation erfolgte analog zu der Evaluation in den Projektabschnitten 4-7 durch Vergleiche der PSM-Referenzmodelle mit generierten PSM-Modellen, die mittels der M2M-Transformationsdefinitionen aus einem vorab manuell erstellten PIM-Referenzmodell erzeugt wurden. Die Finalisierung des Projektabschnitts setzte eine Übereinstimmung der generierten PSM-Modelle mit dem jeweils korrespondierenden PSM-Referenzmodell voraus.

In Projektabschnitt 8 wurden das Metamodell der Modellierungssprache *WASL Data* sowie eine M2M-Transformationsdefinition von *WASL Data* zu *WASL Generic* konstruiert. Die Evaluation erfolgte analog zu den vorigen Projektabschnitten. Ergänzend wurden M2M-Transformationsdefinitionen von *UML* sowie *ERM* zu *WASL Data* erstellt, mit denen UML-Klassendiagramme bzw. ER-Modelle in die Generierung eingehen können.

4.5 Designprinzipien

Softwaresysteme sind im Vergleich zu materiellen Konstrukten aus den Ingenieurwissenschaften keinen physischen Restriktionen unterworfen, sondern weisen im Rahmen der von den Hardware- und Softwareplattformen zur Verfügung gestellten Fähigkeiten beliebige Freiheitsgrade auf. Aus dem Mangel an physischen Restriktionen folgt, dass die Qualität eines Softwareartefakts zwar an seiner Anforderungserfüllung bemessen werden kann, nicht aber an der Einhaltung materieller Kriterien. Die fehlenden Einschränkungen durch die physische Realwelt führen zu der Notwendigkeit von Designprinzipien, um die Freiheitsgrade bei der Softwareentwicklung zur Erlangung sinnvoller Lösungen zu restringieren. Ein Designprinzip ist dabei definiert als ein normatives Prinzip für die Gestaltung eines Artefakts, das als deklarative Aussage normativ die Designfreiheit einschränkt[28] [GP11, S. 35]. Der normative Charakter spiegelt sich darin wider, dass der Designer durch Verhaltensmaßregeln in der Vorgehensweise eingeschränkt und angeleitet wird, die nicht Naturgesetzen entspringen, sondern selbst auferlegt sind.

Dem WASL-Generatorframework liegen diverse Designprinzipien zugrunde, die sich auf die Metamodellierung im Allgemeinen, die Metamodellierung für die Modellierungssprachen *WASL Generic* und *WASL PSM* im Speziellen, und die Gestaltung der Transformationsdefinitionen beziehen.

4.5.1 Metamodellierung im Allgemeinen

In Bezug auf die Erstellung der Metamodelle ist ein zentrales Designprinzip, dass die Metamodelle ausschließlich durch die Syntax und Semantik des jeweils abzubildenden Objektsystems (vgl. [Bro03, S.10 ff.]) determiniert werden. So ist die Struktur eines Metamodells nicht im Hinblick auf bestimmte Transformationsdefinitionen zu optimieren, da diese nicht Teil des Objektsystems sind. Dies betrifft insbesondere den Verzicht auf zusätzliche Referenzen im Metamodell, mit denen die Navigation auf dem Objektgraphen des Modells mit Mitteln der Transformationssprache vereinfacht würde. Falls Navigationsshortcuts bzw. abkürzende Referenzen auf der Ebene des Metamodells erlaubt würden, wären Redundanzen zu den ursprünglichen Navigationspfaden die Folge, und es würden zudem Interdependenzen zwischen dem Metamodell und der Transformationssprache manifestiert. Des Weiteren ist darauf zu verzichten, bei der Benennung von Metamodellelementen und -attributen solche Namen auszuschließen, die in der gewählten Transformationssprache als Schlüsselwörter vorgesehen sind und deren Verwendung zu Namenskollisionen führen kann. Ein Beispiel dafür liegt bei der Verwendung des Attributnamens *class* vor, der mit dem gleich-

[28] Im Original: „DESIGN PRINCIPLE A normative-principle on the design of an artifact. As such, it is a declarative statement that normatively restricts design freedom.“

namigen geschützten Schlüsselwort von QVTO kollidiert [Obj11b, S. 136]. Generell wird bei der Konzeption eines Metamodells davon ausgegangen, dass Transformationssprachen für diese Fälle Escape-Mechanismen aufweisen.

4.5.2 Metamodellierung für *WASL Generic*

Das plattformunabhängige Metamodell der Modellierungssprache *WASL Generic* ist ein Ergebnis des Designprinzips der Separation of Concerns (SoC). Dieses sieht bei der Auseinandersetzung mit einem Sachverhalt eine getrennte Betrachtung der Teilaspekte des Sachverhalts vor, um die Fokussierung auf Teilprobleme zu ermöglichen [Dij82]. Dabei ist eine Minimierung der Interdependenzen zwischen den Teilproblemen hilfreich, um Auswirkungen von Teiländerungen auf das Gesamtsystem kontrollieren zu können.

Im Kontext des Web Engineering wird bei der Modellierung von Webapplikationen generell eine getrennte Modellierung der Teilaspekte *Inhalt*, *Hypertext* und *Präsentation* vorgenommen [SK06]. Der Inhalt umfasst Daten und Logik der Applikation, der Hypertext die Strukturierung des Inhalts durch einen Navigationsgraphen und die Präsentation die Benutzerschnittstelle zur Darstellung der Inhalte.

Dem Anspruch des Prinzips der SoC nach funktional möglichst wenig überlappenden und interdependenten Teilbereichen kann im Kontext der Softwarearchitektur durch die Verwendung einer Schichtenarchitektur Rechnung getragen werden [SBD+10, S. 210 ff.], welche die Anwendung in Schichten mit jeweils spezifischer Funktionalität aufteilt. Das Metamodell von *WASL Generic* setzt die SoC somit in Form einer Schichtenarchitektur um, indem eine Einteilung der Funktionalität von Webapplikationen in die Schichten der *Datenhaltung*, der *Geschäftslogik*, der *Navigation* und der *Präsentation* vorgenommen wird und folgt auf diese Weise grundsätzlich der Einteilung des Web Engineering.

4.5.3 Metamodellierung für *WASL PSM*

Die Metamodelle der plattformspezifischen WASL-Modellierungssprachen sind geprägt vom Designprinzip der vollständigen Explikation deklarativer Beschreibungsmechanismen der Plattformen. Zu diesen zählen generell deskriptive Plattformkonstrukte wie z. B. Konfigurationsdateien oder Descriptoren aus dem Umfeld von Java EE, aber auch statische Aspekte imperativer Programmiersprachen wie z. B. Funktions- und Klassendefinitionen.

Dagegen wird auf die Nachbildung imperativer Programmlogik verzichtet, wie sie sich in Rümpfen von Methoden, Funktionen, Prozeduren oder generell Skripten wiederfindet. Die Imitation von Quelltext in Modellen weitet deren Umfang signifikant aus und führt zu einer entsprechenden Zahl an Intramodellreferenzen. Der damit verbundene Zuwachs an Modellkomplexität ist einerseits nicht kompatibel zu dem mit der Modellierung verbundenen

Zweck der Komplexitätsreduktion durch Abstraktion und induziert zudem entsprechende Komplexität in den M2M- und M2T-Transformationsdefinitionen.

Die Anwendung dieses Prinzips bedeutet z. B. im objektorientierten Kontext, dass Klassendefinitionen mit ihren Klassen- und Objektvariablen sowie Methodensignaturen modelliert werden. Auch Accessor-Methoden für den Zugriff auf gekapselte Objektvariablen werden dem Modell explizit hinzugefügt. Die Explikation hat den Vorteil, dass Accessoren bzw. generell Methoden im Modell referenziert werden können. Dagegen wird die in den Methodenrümpfen enthaltene Programmlogik durch aggregierende Methodentypen dargestellt. Falls für die Generierung der Methodenrümpfe zusätzliche Informationen benötigt werden, sind diese den Methodentypen in Form von Eigenschaften bzw. Meta-Parametern hinzuzufügen. Die Kombination aus Methodentyp und zusätzlichen parametrisierenden Eigenschaften ist damit eine aggregierte Darstellung von Programmlogik, welche durch korrespondierende M2T-Transformationsdefinitionen vollständig generiert werden kann.

Im Kontext deskriptiver Programmier- und Auszeichnungssprachen kann neben der Möglichkeit einer vollständigen Explikation zusätzlich die Modellierung durch aggregierende Sprachelemente sinnvoll sein. So kann z. B. als Alternative zur vollständigen Ausmodellierung eines XHTML-Dokumentenbaums eine abstraktere Darstellung von XHTML-Seitenelementen bzw. Teilbäumen im XHTML-Dokument durch aggregierende Modellelemente, z. B. für Navigationsmenüs oder Kopfzeilen, sinnvoll sein. Der Vorteil dieses Vorgehens liegt in einer stärkeren Ausdruckskraft des Modells, der Nachteil in den redundanten Darstellungsmöglichkeiten für dieselbe Semantik, da ein Konzept des Objektsystems sowohl durch ein einzelnes aggregierendes Modellelement als auch durch ein Konstrukt aus mehreren feingranularen Modellelementen abgebildet werden kann.

Da bei imperativen Ausdrucksmitteln einer Plattform auf die Modellierung von Programmlogik verzichtet wird, dagegen deklarative Ausdrucksmittel vollständig abgebildet werden und zudem optional durch aggregierte Formen erweitert werden, sind Letztere bei der Wahl der Zielplattform zu präferieren. Beispielsweise bieten Objekt-relationale Mapping-Frameworks (ORM) üblicherweise eine deskriptive Möglichkeit zur Ansteuerung relationaler Datenbanken durch objektorientierte Klassen und sind somit imperativen Mitteln zur Interaktion mit Datenbanken vorzuziehen.

Als Ausnahme ist bei textuellen Referenzen eine klare Abweichung von den deskriptiven Ausdrucksmitteln der Zielplattform vorzunehmen. Textuelle Referenzen als Ausdrucksmittel der Plattform sind im Modell nach Möglichkeit durch navigierbare Intramodellreferenzen abzubilden. Dies trifft beispielsweise im Fall von ORM-Frameworks auf Mapping-Descriptoren zu, die Datenbanktabellen objektorientierte Klassen zuordnen. Die Verweise auf die Klassen und Datenbanktabellen erfolgen dabei als textuelle Referenzen auf die kanoni-

schen Klassen- und Tabellennamen. Falls Descriptoren dieser Art in Modellen abgebildet werden, sind die textuellen Referenzen durch Intramodellreferenzen auf diejenigen Modellelemente abzubilden, welche die Klassen bzw. Tabellen repräsentieren.

4.5.4 Gestaltung der Transformationsdefinitionen

M2M-Transformationsdefinitionen, die wie im Fall der Transformationsdefinition von *WASL Generic* zu *WASL PSM* Modelle von Schichtenarchitekturen transformieren, referenzieren in ihren Transformationsregeln Metamodellelemente für Elemente aus den Schichten dieser Architekturen. Analog zur Strukturierung von Metamodellen sollen ebenso die Transformationsregeln dem Prinzip der SoC entsprechen, indem die Metamodellelement-Referenzen auf die tiefstmöglichen Schichten begrenzt werden. So soll z. B. eine QVTO-Transformation, die aus Entities Klassen generiert, keine Referenzen zu Metamodellelementen höherer Schichten wie z. B. der Präsentationsschicht aufweisen. Eine Verletzung dieses Prinzips hätte zur Folge, dass eine Minimierung der Interdependenzen zwischen den Teilproblemen der Gesamttransformationsdefinition verhindert wird, und somit die Komplexität der Teilprobleme erhöht wird.

Im Hinblick auf die M2T-Transformationsdefinitionen gilt das Designprinzip, dass bei der Generierung von Programmlogik in Form von Quelltext lediglich minimale implizite Prämissen über den Kontext der Programmlogik gewählt werden. Der Kontext von Programmlogik wird bei objektorientiertem Quelltext z. B. durch die Zuordnung einer Klasse zu einem Paket oder Importe von Paketen und Klassen im Rahmen einer Klassendefinition determiniert. Im Fall deklarativer Programmierung wird der Kontext beispielsweise bei XML-Dokumenten durch XML-Namespaces [06.06] festgelegt. Prämissen über den Kontext führen zu Abhängigkeiten, indem der Quelltext nur bei der Erfüllung der Annahmen über den Kontext valide ist. Z. B. bedeutet dies bei Methodenrümpfen objektorientierter Klassen, dass bei der Referenzierung von Klassen aus anderen Paketen entweder eine Referenzierung über einen vollqualifizierten Klassennamen oder aber über den Klassennamen i.V.m. einem Import im Kontext der Klasse realisiert werden kann. Die Nennung des vollqualifizierten Klassennamens setzt keinen Kontext voraus, und ist deshalb zu präferieren. Bzgl. der Programmierung von Codegeneratoren weist diese Herangehensweise den Vorteil auf, dass die Komplexität der M2T-Transformationsdefinitionen reduziert wird, da weder Importe in Abhängigkeit vom restlichen Quelltext generiert werden müssen, noch Import-Anweisungen im Modell zu explizieren sind.

4.6 Metamodelle

Im Folgenden werden die Metamodelle der domänenspezifischen Modellierungssprachen des WASL-Generatorframeworks beginnend mit *WASL Data* erläutert. Diese definieren die abstrakten Syntaxen der Modellierungssprachen in Form von Metamodellelementen und deren Beziehungen untereinander. Aus den Metamodellen werden mit EMF automatisiert Editoren zur Bearbeitung der Modelle erzeugt, welche die Modelle entsprechend zu den Containment-Beziehungen zwischen den Modellelementen als Containment-Hierarchien darstellen (vgl. Kapitel 2.6.2, [SBPM08, S. 83]). Aufgrund der automatisierten Bereitstellung von Modelleditoren wird auf die Spezifikation konkreter Syntaxen für textuelle oder graphische Notationsformen verzichtet.

4.6.1 Metamodell von *WASL Data*

WASL Data ist eine semiformale und domänenspezifische Modellierungssprache, die im WASL-Generatorframework zur Erstellung von CIMs dient. Die sprachlichen Ausdrucksmittel ermöglichen die Modellierung der Datenstrukturen von Webapplikationen, welche als Ausgangspunkt für nachfolgende Anreicherungen durch Transformationsdefinitionen bis hin zu PSMs und Quelltext dienen. *WASL Data* abstrahiert als CIM-Sprache von sämtlichen berechnungs- und plattformspezifischen Details, ist aber im Hinblick auf den Verwendungszweck für anschließende Transformationen zu PIMs optimiert.

Für die Zwecke der Datenmodellierung existiert eine Vielzahl an Modellierungssprachen, die seit den 1950er Jahren [YK58, Bub07][HKL95, S. 40 ff.] Forschungsgegenstand sind. Von diesen sind insbesondere im Kontext der fachkonzeptionellen und berechnungsunabhängigen Modellierung Entity-Relationship-Modelle (ERMs) [Che76] in verschiedenen Abwandlungen wie z. B. [SS77], [TYF86] als De-facto-Standard etabliert. Eine Erweiterung der zunächst strukturellen Datenmodellierung durch ERM ist die mit dem objektorientierten Ansatz verbundene objektorientierte Modellierung, welche die Erfassung von Verhaltensaspekten hinzufügt [Sta05, S. 19]. Im Kontext der objektorientierten Modellierung von Softwaresystemen sind zu diesem Zweck Klassendiagramme der UML [Obj07b] als Standard verbreitet.

Die Modellierungssprache *WASL Data* weist im Vergleich zu ERM und UML einige charakteristische Unterschiede auf, welche aus Optimierungen vor dem Hintergrund des MDSD resultieren und ihren Einsatz im Rahmen des WASL-Generatorframeworks motivieren.

Ein Defizit von ERM für die Nutzung zum MDSD ist die mangelnde einheitliche Typisierung von Attributen. So sieht das Metamodell von ERM abhängig von der Sprachdefinition die Kennzeichnung von Attributen durch Domänen [Sch01, S. 74] bzw. Wertmengen

[Che76], durch Domänen i.V.m. Typen ohne Bezug zu Operationen [Gia10, S. 311] oder aber ohne jegliche Angabe von Wertmengen oder Typen vor. Entsprechend wird abhängig vom ERM-Dialekt durch die Begrenzung auf Wertebereiche nur ein Teilaspekt von Datentypen abgedeckt, bei denen darüber hinaus Syntax und Semantik von Operationen auf den Wertebereichen eine Rolle spielen [DW96, S. 2 ff.]. Zudem werden Domänen auf der Modellebene ausgeprägt und sind nicht als Metamodellelemente auf der Sprachebene spezifiziert, sodass eine typsichere Referenzierung durch Transformationsdefinitionen nicht möglich ist.

Mit dem *Enhanced Entity-Relationship (EER) Model* existiert eine etablierte Erweiterung von ERM um einen Generalisierungs- bzw. Spezialisierungsmechanismus [SS77], der als solcher für die Zwecke des Generatorframeworks benötigt wird, aber die mehrfache Generalisierung und damit verbunden die Mehrfachvererbung von Attributen vorsieht [EN10, S. 245 ff.]. Dies führt in Abhängigkeit von den Zielplattformen des Generatorframeworks zu Problemen, da einige der objektorientierten Programmiersprachen dieser Plattformen Mehrfachvererbung nicht unterstützen. Zur Vermeidung von Abbildungsdefekten der Modelle ist aus diesem Grund eine Beschränkung der CIM-Sprache auf Einfachvererbung sinnvoll.

Die Problematik der unerwünschten Mehrfachvererbung trifft ebenso auf die UML zu, welche die Zuordnung multipler Superklassen zu einer Klasse ermöglicht [Obj10c, S. 95 f.]. Generalisierungen können zudem als Mengen in Form von *GeneralizationSets* zusammengefasst werden [Obj10c, S. 76-84][Fow03, S. 57 f.], für die ebenfalls überwiegend keine Entsprechungen in objektorientierten Programmiersprachen existieren.

Hinsichtlich der Abbildung von Basisdatentypen weist die UML analog zu ERM das Problem auf, dass Basisdatentypen nicht durch das Metamodell, sondern als Bestandteil des Modells formuliert und von typisierten Modellelementen referenziert werden [Obj10b, S. 101]. Dies trifft ebenso auf die Spezifikation des UML-Metamodells durch MOF zu, bei dem Basisdatentypen Teil des Metamodells [Obj10b, S. 171] sind, und nicht wie bei Ecore durch das Metametamodell bereitgestellt werden [SBPM08, S. 143 f.]. Die Definition von Basisdatentypen auf der Modellebene bietet zwar Flexibilität, um verschiedene Programmiersprachen mit ihren unterschiedlichen primitiven Datentypen als Bestandteile der Typsysteme adressieren zu können. Allerdings führt sie analog zu ERM zu dem Problem, dass eine typsichere Referenzierung von Basisdatentypen durch Transformationsdefinitionen verhindert wird und stattdessen Namensvergleiche zur Identifizierung von Typen durchgeführt werden müssen.

WASL Data weist durch ein klassenähnliches Konzept intendiert Ähnlichkeiten zu der UML auf, da mit der Sprache Softwareentwickler adressiert werden, welche aufgrund der

Verbreitung des objektorientierten Programmierparadigmas als UML-affin angenommen werden. Als wesentliche Merkmale unterstützt das einfach gehaltene Metamodell (vgl. Abbildung 24) erstens die Modellierung von Entities, welche durch Pakete strukturiert und durch typisierte Attribute beschrieben werden, zweitens bietet es Einfachvererbung durch einfache Generalisierung und drittens gibt es sprachlich fixierte Basisdatentypen vor.

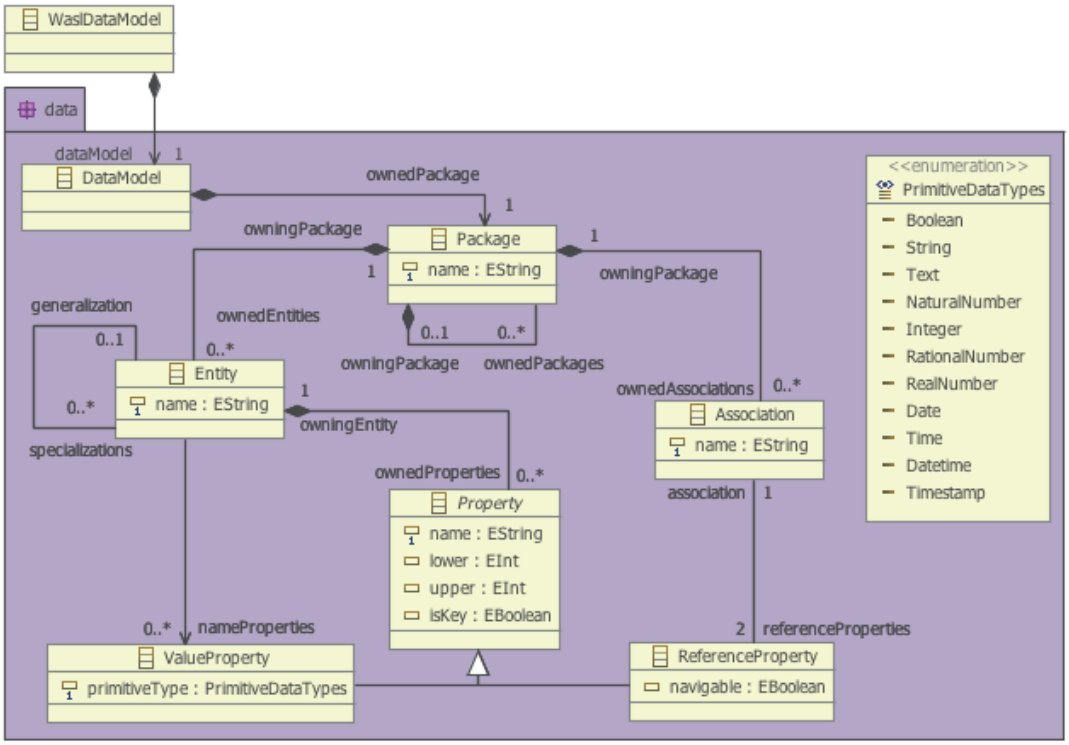

Abbildung 24: Metamodell von *WASL Data*

In EMF wird ein Modell in Form einer Containment-Hierarchie strukturiert, die mit XMI als eine XML-typische Elementhierarchie serialisiert werden kann [SBPM08, S. 57]. Als Wurzelelement dieser Hierarchie dient in *WASL Data* das Metamodellelement *WaslDataModel*, dessen Instanz im Modell als Kindelement ein Modellelement vom Typ *DataModel* enthält. Die Trennung zwischen *WaslDataModel* und *DataModel* dient dazu, das Metamodell von *WASL Data* ähnlich dem Metamodell von *WASL Generic* (Kapitel 4.6.2) zu strukturieren. Die in einem *DataModel* enthaltenen Modellelemente werden in einer Hierarchie von *Packages* strukturiert, deren Wurzelknoten durch das EAttribute *ownedPackage* in dem *DataModel* enthalten ist. Jedes Paket kann zur Bildung der Hierarchie durch die gegenseitigen EReferences *owningPackage* und *ownedPackages* mit den Kardinalitäten *0:1* bzw. *0:n* weitere Pakete enthalten, sowie *Entities* und *Associations* aufnehmen.

Entities bilden identifizierbare Konzepte aus der betrieblichen Domäne ab. Dies schließt

z. B. *Entities* für die Konzepte *Kunde*, *Artikel* oder *Bestellung* ein, dagegen das technische Konzept eines *EntityManagers* oder einer *Konfigurationsdatei* aus. Jedes *Entity* kann durch *Properties* mit typisierten Attributen verfeinert werden, welche in *ValueProperties* und *ReferenceProperties* unterteilt sind. Jedes *Property* ist ein- oder mehrelementig und besitzt dazu mit den EAttributes *lower* und *upper* eine Kardinalität in Min-Max-Notation, wobei eine beliebig mehrwertige obere Schranke durch den Wert *-1* kodiert wird.

ValueProperties dienen der Modellierung von Attributen mit primitiven Basisdatentypen, welche im Metamodell in der EEnum *PrimitiveDataTypes* als EEnumLiterale zur Auswahl vorgegeben sind. Die Definition der Basisdatentypen im Metamodell stellt sicher, dass sie in Transformationsdefinitionen typsicher referenziert werden können. Exemplarisch sind neben den Typen *String* und *Integer* weitere Typen mit Zeitbezug vorgesehen. Von einer Repräsentation derselben Zahlenmenge durch numerische Datentypen mit unterschiedlichen Wertebereichen wird abgesehen, da im CIM von berechnungsspezifischen Details abstrahiert wird.

Im Unterschied zu *ValueProperties* bilden *ReferenceProperties* Beziehungsattribute zu anderen *Entities* ab. Wie auch bei der UML [Obj10c, S. 38 f., 125 f.] können zwei *Properties* durch eine *Association* in Beziehung zueinander gesetzt werden, um bidirektionale Referenzen und damit eine bidirektionale Navigation im Modell zu ermöglichen. Im Unterschied zu einem *ValueProperty* ist der Typ des *ReferenceProperty* nicht ein Basisdatentyp, sondern derjenige Typ bzw. dasjenige *Entity*, das durch die *Association* referenziert wird. Durch das EAttribute *navigable* kann eine Beziehung als unidirektional definiert werden, was abhängig von der Zielplattform z. B. in einseitige Referenzen mündet. Da eine *Association* zwei verschiedene *Entities* referenzieren kann, die in unterschiedlichen *Packages* enthalten sind, ist aus den *Packages* der *Entities* nicht zu bestimmen, an welcher Stelle in der Containment-Hierarchie des Modells die *Association* zu verorten ist. Deshalb wird jede *Association* mit der EReference *owningPackage* explizit einem *Package* zugeordnet, wobei eine Restriktion auf bestimmte *Packages* wie z. B. solche der beiden *Entities* nicht existiert.

Entities können über die gegenseitigen EReferences *generalization* und *specializations* mit den Kardinalitäten *0:1* bzw. *0:n* in Generalisierungs- bzw. Spezialisierungsbeziehungen zueinander stehen und auf diese Weise in einem Modell mehrere Typhierarchien bilden. Semantisch ist mit einer Typhierarchie die Vererbung von Attributen an spezialisierende Entitites verbunden.

Des Weiteren können *ValueProperties* eines *Entity* in der mehrwertigen EReference *nameProperties* zusammengefasst werden, die der Benennung von *Entity*-Instanzen dient.

4.6.2 Metamodell von *WASL Generic*

WASL Generic ist eine semiformale und domänenspezifische Modellierungssprache zur plattformunabhängigen Modellierung von Webapplikationen im WASL-Generatorframework. Sie ermöglicht entsprechend dem Prinzip der Separation of Concerns eine getrennte Modellierung von Datenstrukturen, Navigationsstruktur, Geschäftslogik und Präsentationsaspekten. Die Trennung orientiert sich an der für Client-Server-Systeme und betrieblichen Anwendungssysteme üblichen Strukturierung als Schichtenarchitektur [Fow02, S. 17-24][SBD+10, S. 211-215], mit dem Ziel, die Schichten möglichst unabhängig voneinander zu konzipieren. Korrespondierend zu den Schichten ist das Metamodell in die EPackages *data*, *navigation*, *logic* und *presentation* unterteilt (vgl. Abbildung 25). Das Metamodellelement *WASLGenericModel* dient als Wurzelelement der Containment-Hierarchie, dessen Instanz im Modell die Untermodelle vom Typ *DataModel*, *NavigationModel* und *ViewModel* enthält.

Das EPackage *data* entspricht isomorph dem EPackage *data* aus dem Metamodell von *WASL Data*, sodass *WASL Generic* als eine Erweiterung von *WASL Data* interpretiert werden kann und die Transformationsregeln zwischen den beiden *data*-EPackages trivial sind.

Die Metamodellelemente im EPackage *navigation* dienen der Modellierung der Navigationsstruktur der Webapplikation in Form eines gerichteten Graphen mit Webseiten als Knoten (*Node*) und Hyperlinks als Kanten (*Link*). Ein *Node* ist dabei definiert als „ein Teil einer Web-Applikation, der eine bestimmte Funktionalität abdeckt, durch eine URL eindeutig adressiert ist und durch die Bedienelemente des Webbrowsers wie etwa Navigationsknöpfe, Lesezeichen und Webbrowser-Historie erreicht werden kann“ [Wol10]. Durch einen *Node* wird entweder im einfachsten Fall eine einzelne Webseite abgebildet, oder alternativ eine Mehrzahl unterschiedlicher Darstellungen bzw. Ansichten. Für beide Fälle gilt die Bedingung, dass die Seiten bzw. Darstellungen durch dieselbe URL zu adressieren sind. *Nodes* sind jeweils in einer Knotengruppe (*NodeGroup*) enthalten. Knotengruppen sind ihrerseits in einer Hierarchie von Knotengruppen strukturiert, die sich an der Unterteilung der Webapplikation in Funktionsbereiche orientiert. Ein *Link* zwischen zwei *Nodes* drückt einen Navigationspfad aus, der in der plattformspezifischen Implementierung als Verweis auf eine URL z. B. in Form eines Hyperlinks oder eines *action*-Attributs eines Formulars implementiert wird.

Das globale Navigationsmenü einer Webapplikation und die aus diesem resultierenden globalen Navigationspfade werden getrennt vom Navigationsgraphen mit dem speziellen Metamodellelement *MenuEntry* für Menüeinträge und *MenuGroup* für Menüordner modelliert. Bei einem Verzicht auf eine separate Modellierung müssten im Navigationsgraphen al-

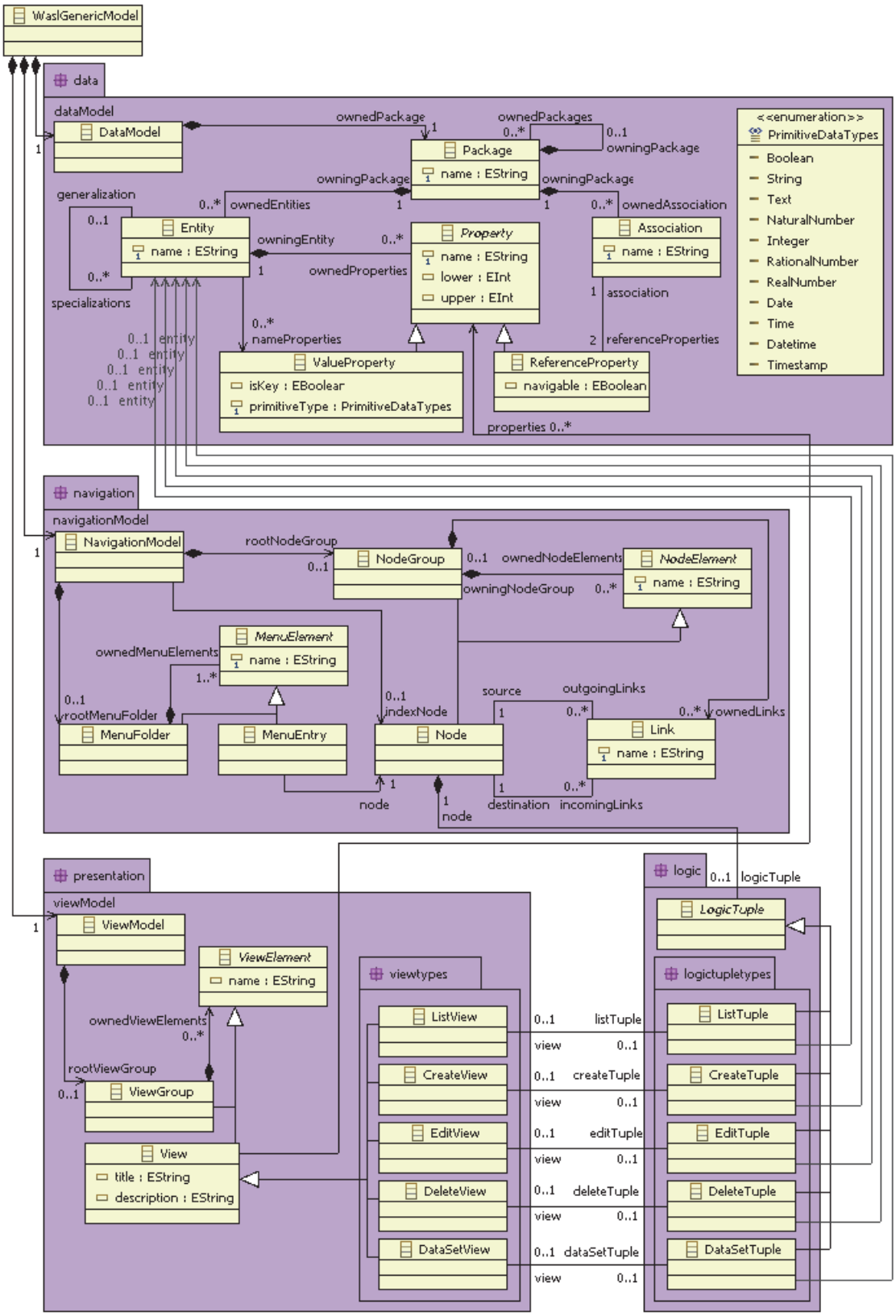

Abbildung 25: Metamodell von *WASL Generic*

ternativ für jeden Menüeintrag *Links* zwischen sämtlichen *Nodes* und dem im Menüeintrag referenzierten *Node* eingefügt werden, da der referenzierte *Node* von sämtlichen *Nodes* aus erreichbar ist. Aus einer Gesamtanzahl von *n Nodes* im Navigationsgraphen, von denen *m* durch einen Menüeintrag erreichbar sind, würden somit *n*m* zusätzliche *Links* im Navigationsgraphen resultieren, sodass der Grundsatz der Klarheit des Modells verletzt wäre (vgl. [BRS95]). Durch die getrennte Modellierung wird zudem dem semantischen Unterschied zwischen Hyperlinks im Navigationsmenü und Hyperlinks in Seiteninhalten Rechnung getragen, sodass die Transformationsdefinitionen klarer strukturiert werden können.

Ähnlich zu WebML wird auch in *WASL Generic* Geschäftslogik in Form aggregierter Logikeinheiten modelliert und auf eine verfeinerte Spezifikation von Programmlogik verzichtet. Im Unterschied zu WebML wird aber eine Trennung des Geschäftslogikmodells vom Navigationsmodell vorgenommen, indem ein Node an sich keine Semantik bzgl. der Geschäftslogik trägt, sondern jeweils durch ein separates *LogicTuple* angereichert wird. Ein *LogicTuple* drückt aus, welcher *Node* welche Geschäftslogik, optional basierend auf welchem *Entity*, abdeckt. Ähnlich zu den standardisierten CRUD-Datenoperationen der verschiedenen MDWE-Ansätze sind in *WASL Generic* Spezialisierung von *LogicTuple* in Form von *ListTuple* für Auflistungen, *CreateTuple*, *EditTuple* und *DeleteTuple* für Datenmodifikationen, und *DataSetTuple* für die Ausgabe eines Datensatzes vordefiniert.

Die graphische Darstellung der Daten und Geschäftslogik wird schließlich im Präsentationsmodell durch aggregierende *Views* repräsentiert. Analog zu der Spezialisierung von *LogicTuple* existieren mit *ListView*, *CreateView*, *EditView*, *DeleteView* und *DataSetView* vordefinierte *View*-Subtypen. Die Trennung von *Views* und *LogicTuples* ermöglicht eine getrennte Modellierung von Präsentation und Geschäftslogik, sodass dieselbe Geschäftslogik durch unterschiedliche *Views* auf unterschiedliche Weisen dargestellt werden kann. Dies ist erstens vorteilhaft, falls unterschiedliche Gerätetypen adressiert werden sollen, ermöglicht zweitens die Modellierung von Geschäftslogik ohne Präsentation und vereinfacht drittens eine arbeitsteilige Projektplanung, indem Geschäftslogik personell und zeitlich getrennt von der Präsentation modelliert werden kann.

Ein weiterer Vorteil der Trennung von Geschäftslogik und Präsentation ist vor dem Hintergrund der Verbreitung von AJAX bei Webapplikationen zu sehen. AJAX ermöglicht als Programmierkonzept auf der Basis von Javascript, XML und JSON prinzipiell eine clientseitige Modifikation von HTML-Seiteninhalten i.V.m. asynchronen Datenübertragungen zwischen dem Webbrowser und dem Webserver ohne die Notwendigkeit einer erneuten Generierung der Webseite durch den Webserver [Gar05, Zuc07]. Abhängig von der technischen Realisierung führt AJAX aber zu dem Problem, dass die Navigations-Schaltflächen des Webbrowsers nicht mit den Interaktionen mit der Webseite korrespondieren, da

keine klassische Navigation zwischen URLs stattfindet. In *WASL Generic* wird diese Problematik kontrolliert, indem AJAX-Funktionalität ausschließlich *Views* zugeordnet wird. Damit wird der Einsatz von AJAX auf reine Präsentationslogik begrenzt, da Navigationslogik und Wechsel zwischen URLs sich ausschließlich im Navigationsgraphen widerspiegeln, der klar vom Präsentationsmodell separiert ist. Indem in der plattformspezifischen Implementierung die Verwendung von AJAX auf reine Präsentationslogik eingeschränkt wird, wird verhindert, dass ein Systementwickler mit AJAX über einfache Präsentationslogik hinaus komplexere Navigationsstrukturen implementiert und ein unerwartetes Verhalten der Navigations-Schaltflächen des Webbrowsers erzeugt.

Die Menge der vordefinierten Typen von *LogicTuples* und *Views* kann im Rahmen einer Sprachkonfiguration mit dem Spezialisierungsmechanismus von Ecore beliebig ausgeweitet werden, um zusätzliche Semantik abzubilden, ohne *WASL Generic* im Kern und damit die auf dem Metamodell aufbauenden Transformationsdefinitionen grundlegend restrukturieren zu müssen. Ein Beispiel für eine Erweiterung ist ein *LogicTuple* zum Aufruf von Webservices oder BPMN-Prozessen.

4.6.3 Metamodelle von *WASL PSM*

Die plattformspezifischen Modellierungssprachen *WASL JavaEE*, *WASL PHP* und *WASL Python* des WASL-Generatorframeworks werden als Kombinationen einer Vielzahl von Metamodellen für das Dateisystem, Programmiersprachen und Frameworks spezifiziert, die generell als *WASL PSM* bezeichnet werden. Die Metamodelle sind in einer Schichtenarchitektur strukturiert, bei der Metamodelle höherer Schichten Metamodelle aus tieferen Schichten referenzieren (vgl. Abbildung 26). Referenzen zwischen Metamodellen resultieren entweder aus Spezialisierungsbeziehungen zwischen EClasses verschiedener Metamodelle, oder aus der Referenzierung von EClasses bzw. EEnums als Typen in EReferences bzw. EAttributes von EClasses anderer Metamodelle.

Die Schichtenarchitektur ist entsprechend den Metamodelltypen in die *Dateisystemschicht*, *Sprachenschicht* und *Frameworkschicht* unterteilt. Der Dateisystemschicht ist das Metamodell *Filesystem* zur Repräsentation des Dateisystems zugeordnet, das primär von den Metamodellen *Java6*, *PHP5* und *Python2* aus der Sprachenschicht referenziert wird. Ebenso ist der Sprachenschicht das Metamodell *XHTML1-Strict_XSD_Adapted* für die Auszeichnungssprache XHTML zugeordnet, das aus der XML-Schemadefinition von XHTML mittels einer Meta-Transformationsdefinition von EMF [SBPM08, S. 197-259][04.04b] automatisiert abgeleitet ist. Da das Metamodell von XHTML Dokumente, nicht aber Dateien spezifiziert, wird durch das Metamodell *XHTML1File* eine Verbindung zwischen dem XHTML-Metamodell und dem *Filesystem*-Metamodell geschaffen. Die restlichen Metamodelle

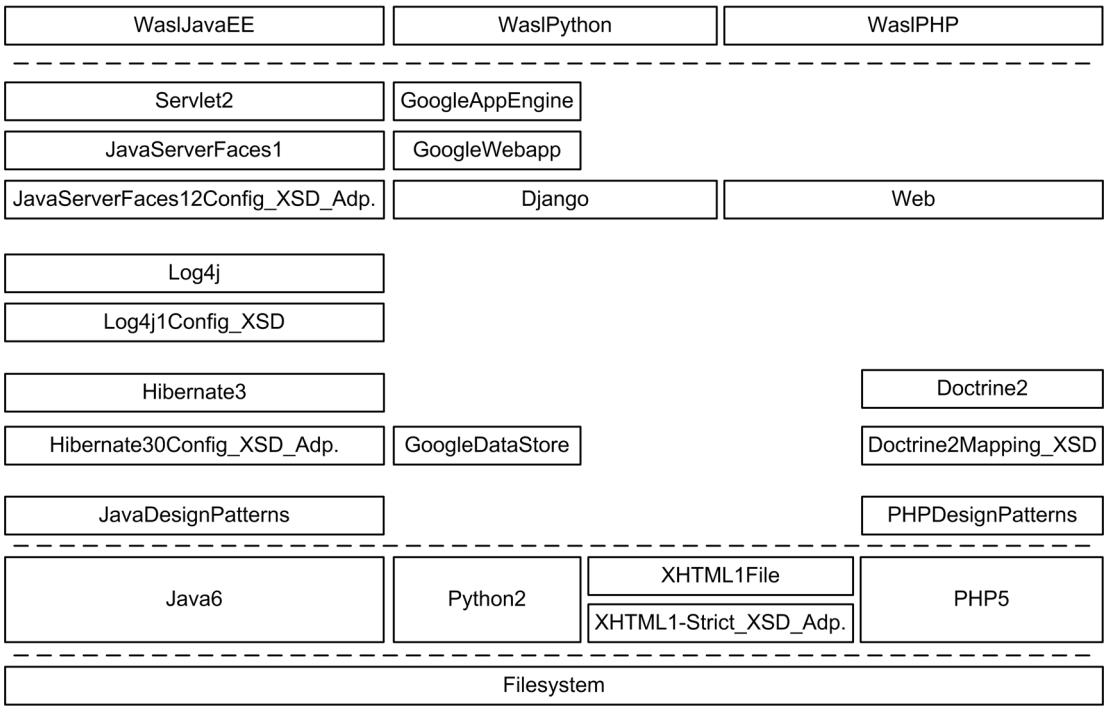

Abbildung 26: Architektur von *WASL PSM*

sind den drei Webplattformen zugeordnet und in den meisten Fällen jeweils durch Referenzen auf das Metamodell der zugehörigen Programmiersprache für die Nutzung im Kontext dieser Plattform restringiert.

Der Ausschnitt der Frameworkschicht für *WASL JavaEE* umfasst für die Frameworks *Log4j*, *Hibernate 3* und *JavaServer Faces (JSF)* die Metamodelle *Log4j1Config_XSD*, *Log4j*, *Servlet2*, *Hibernate30Config_XSD_Adapted*, *Hibernate3*, *JavaServerFaces1* und *JavaServerFaces12Config_XSD_Adapted*. Diese nutzen partiell Entwurfsmuster wie z. B. das Singleton, die im Metamodell *JavaDesignPatterns* typisiert sind. Das Metamodell *WaslJavaEE* spezifiziert abschließend das Wurzelelement der Containment-Hierarchie von *WASL JavaEE*-Modellen.

Der Ausschnitt der Frameworkschicht für *WASL Python* ist geprägt von der Google App Engine[29] als Beispiel für eine Python-basierte Webplattform. Die Metamodelle *GoogleDatastore*, *GoogleWebapp* und *GoogleAppengine* adressieren Google-eigene Frameworks und werden ergänzt um das Webframework *Django* für die Erstellung von Webseiten-Templates. Analog zu dem Metamodell *WaslJavaEE* enthält das Metamodell *WaslPython* ein Metamodellelement zur Typisierung des Wurzelelements der Containment-Hierarchie von *WASL Python*-Modellen.

[29] `http://appengine.google.com`

Der Ausschnitt der Frameworkschicht für *WASL PHP* adressiert exemplarisch als einziges Framework den objektrelationalen Mapper Doctrine 2[30] mit den Metamodellen *Doctrine2Mapping_XSD* und *Doctrine2*. Die restlichen Metamodelle *PHPDesignPatterns* und *Web* typisieren ohne Bezug zu Frameworks beispielhaft projektspezifische Methoden und Codefragmente mit prozeduralem Charakter, wie sie in dem vergleichsweise heterogenen Ökosystem von PHP verbreitet sind. Mit dem Metamodell *WaslPHP* existiert ebenfalls eine Typisierung für das Wurzelelement der Containment-Hierarchie von *WASL PHP*-Modellen.

```
1 <FrameworkName><FrameworkVersion>([Mapping|Config|...])?((_XSD)|(_XSD_Adapted))?(_<
          MetamodellVersion>).ecore
```

Listing 8: Benennungsschema der Metamodelldateien

Die Benennung der Metamodelldateien folgt einem einheitlichen Schema, das eine Konkatenierung des Frameworknamens mit der Frameworkversion, optional einer Kategorisierung und der Metamodellversion vorsieht (vgl. Listing 8). Der Name des Frameworks wird mit Binnenmajuskeln (CamelCase) und ausschließlich mit dem lateinischen Alphabet in Groß- und Kleinschreibung formuliert. Für die Frameworkversion wird die Hauptversionsnummer des Frameworks angegeben, damit einerseits alte Versionen von Frameworks parallel zu neuen Versionen im Generatorframework und in Modellen verwendet werden können, die Metamodelldateien aber andererseits nicht bei kleineren Versionssprüngen der Frameworks umbenannt werden müssen. Im Fall von Metamodellen für Konfigurationsdateien oder OR-Mapper-Konfigurationen werden die Metamodelldateien durch Zusätze wie z. B. *Config* oder *Mapping* als Metamodelldateien kategorisiert. Zudem werden Metamodelldateien, die aus einer Konvertierung von XML Schema zu EMF resultieren, per *XSD* gekennzeichnet, die um ein *Adapted* erweitert wird, falls nach der Erzeugung Anpassungen vorgenommen wurden. Jede Metamodelldatei wird abschließend versioniert. Auf eine Angabe der Metamodellversion wird in dieser Arbeit verzichtet, da sämtliche Metamodelle die Versionsnummer 1.0 aufweisen.

4.6.3.1 Metamodell *Filesystem*

Das Metamodell für das Dateisystem besteht aus der abstrakten EClass *DirectoryElement*, welche durch die Sub-EClasses *Directory* und *File* spezialisiert wird (vgl. Abbildung 27). Die Hierarchie des Dateisystems wird durch ein Kompositum-Entwurfsmuster (vgl. [GHJV94, S. 163 ff.]) abgebildet, indem ein *Directory* über die Containment-EReference

[30] `http://www.doctrine-project.org/`

directoryElements beliebig viele *Directories* und polymorph *Files* beliebigen Typs enthalten kann. Jedes *DirectoryElement* hat zudem einen Namen, der im Fall eines *File* mit dem EAttribute *extensionName* um eine Dateiendung ergänzt wird. Das Metamodellelement *File* ist ein Ankerpunkt für Erweiterungen durch andere Metamodelle, die spezielle Formen von Dateien spezifizieren. Die Containment-Hierarchie im Modell wird in Form der Verzeichnisstruktur aufgespannt und integriert die verschiedenen Dateitypen mit ihren speziellen Inhalten wie z. B. Javaklassen.

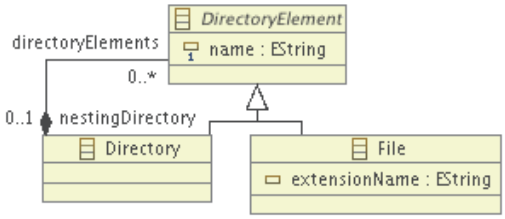

Abbildung 27: Metamodell *Filesystem*

Die Abbildung des Dateisystems im Modell ist ein Kompromiss zwischen einem geringen Abstraktionsgrad und einer erhöhten Flexibilität. Letztere äußert sich bei der Modellierung darin, dass beliebige Kombinationen von Sprachen und Frameworks adressiert werden können sowie projektspezifische Artefakte mit Bezug zum Dateisystem wie z. B. ANT-Skripte ebenfalls modelliert und ohne weitergehende Parametrisierung des Generators erzeugt werden können. Vorteilhaft bzgl. der Metamodelle ist zudem, dass diese über das generische *Filesystem*-Metamodell als gemeinsame Basis lose gekoppelt werden.

4.6.3.2 Metamodelle *XHTML1File* und *XHTML1-Strict_XSD_Adapted*

Das Metamodell *XHTML1File* verbindet das Metamodell *XHTML1-Strict_XSD_Adapted* mit dem Metamodell *Filesystem* und ist somit ein Beispiel für eine Spezialisierung für dessen Metamodellelement *File* (vgl. Abbildung 28). Indem das Metamodellelement *XHTMLFile* eine Containment-EReference auf das Wurzelelement *DocumentRoot* des XHTML-Metamodells besitzt, kann ein *XHTMLFile* in der Containment-Hierarchie des Modells ein XHTML-Dokument enthalten.

Das Metamodell *XHTML1-Strict_XSD_Adapted* ist automatisch aus der XML-Schemadefinition für XHTML [W3C02] erzeugt und repräsentiert deren Konzepte mit den sprachlichen Ausdrucksmitteln von Ecore (vgl. Abbildung 29). Es umfasst Metamodellelemente als Pendants für die verschiedenen Elementtypen von XHTML, wie z. B. die EClass *HtmlType* für den XHTML-Elementtyp *html*, die EClass *BodyType* für *body*, die EClass *HeadType* für *head* und die EClass *TitleType* für *title* als ein Unterelement von *head* (vgl. Listing

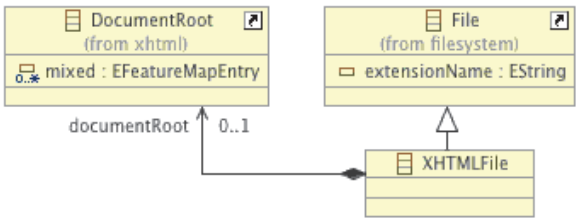

Abbildung 28: Metamodell *XHTML1File*

9). Mit diesen können Modelle strukturähnlich zu XHTML-Dokumenten erstellt werden. Das Metamodellelement *DocumentRoot* ist nicht originär als Elementtyp in der XML-Schemadefinition enthalten, sondern wird dem Metamodell bei der Erzeugung aus der XML-Schemadefinition automatisch hinzugefügt, falls in der Schemadefinition mindestens ein globales Element [W3C04a, vgl. Kapitel 2.2.2] enthalten ist [SBPM08, S. 200].

Das übrige Metamodell wird aufgrund des Gesamtumfangs von 86 EClasses nicht weiter ausgeführt, da die Syntax von XHTML als bekannt vorausgesetzt wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" ...>
  <xs:element name="html">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="head"/>
        <xs:element ref="body"/>
      </xs:sequence>
      ...
      <xs:attribute name="id" type="xs:ID"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="head">...</xs:element>

  <xs:element name="body">...</xs:element>
  ...
</xs:schema>
```

Listing 9: Auszug aus der XML-Schemadefinition *xhtml1-strict.xsd* [W3C02]

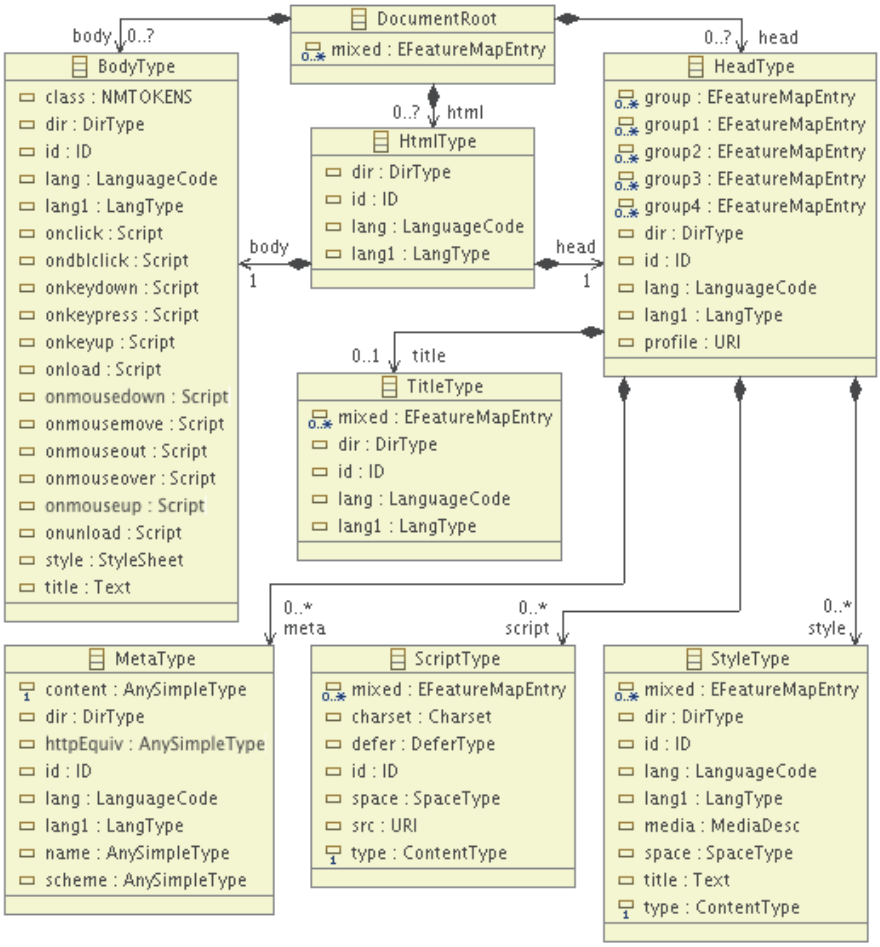

Abbildung 29: Ausschnitt aus dem Metamodell *XHTML1-Strict_XSD_Adapted*

4.6.3.3 Metamodelle für Java EE

Metamodell *Java6*

Das Metamodell *Java6* für die Programmiersprache Java bildet deren wichtigsten Ausdrucksmittel zur objektorientierten Programmierung ab (vgl. Abbildung 30). Das zentrale Metamodellelement ist die abstrakte EClass *Classifier*, welche durch die EClasses *Class* und *Interface* spezialisiert wird. Analog zu *class*-Dateien sind diese in Instanzen der EClass *ClassifierFile* enthalten, die über das Metamodellelement *File* in das Modell des Dateisystems eingebunden wird. Das Package einer Javaklasse bzw. eines -interfaces wird mit dem EString *nestingPackage* erfasst. Auf die strukturierte Modellierung einer Pakethierarchie wird also verzichtet, da diese redundant zu der Dateisystem-Hierarchie ist.

Parallel zu diesen Metamodellelementen werden Frameworkklassen und -interfaces durch die EClasses *FrameworkClass* und *FrameworkInterface* repräsentiert. Die separate Typisierung ist vorteilhaft, um Javaklassen und -interfaces von Frameworks zwar in das Modell zu integrieren, diese aber nicht zu generieren.

Die EClasses *FrameworkClassifier* und *Classifier* werden durch die EClass *AbstractClassifier* generalisiert, deren Funktionalität durch die abstrakten EClasses *Variable* und *Method* i.V.m. der EClass *Parameter* spezifiziert wird. Die drei EClasses spezialisieren die EClass *MultiplicityElement*, deren EAttribute *isMultiple* ausdrückt, ob die Variable, der Parameter oder der Rückgabetyp der Methode ein- oder mehrwertig ist. Die drei EClasses werden jeweils in zweifacher Ausführung als Version mit primitivem Datentyp durch die EClasses *PrimitiveVariable*, *PrimitiveParameter* und *PrimitiveMethod* und als Version mit Referenzdatentyp durch die EClasses *ReferenceVariable*, *ReferenceParameter* und *ReferenceMethod* spezialisiert. Variablen, Parameter und Methoden mit einem primitivem Datentypen werden von solchen mit einem Referenzdatentypen unterschieden, um Ersteren über das jeweilige EAttribute *type* einen Typen aus der sprachlich vordefinierten EEnum *PrimitiveType* zuzuweisen, Letzteren dagegen über die jeweilige EReference *type* eine Instanz der EClass *AbstractClassifier*. Die statische Typisierung von Variablen, Parametern und Methoden in Java drückt sich in der Kardinalität des jeweiligen EAttribute bzw. der jeweiligen EReference mit dem Wert *1* aus, d. h. jede Instanz eines dieser drei Metamodellelemente muss typisiert werden. Die EClass *ReferenceMethod* wird durch die EClass *Constructor* spezialisiert, um Konstruktoren als spezielle Methoden im Modell durch einen eigenen Modellelementtypen zu kennzeichnen.

Methoden und *Classifier* können als Spezialisierungen von *AnnotableElement* mit Annotationen als Spezialisierungen der abstrakten EClasses *AbstractAnnotation* und *FrameworkAnnotation* versehen werden. Das Metamodell selbst enthält somit keine instanziierba-

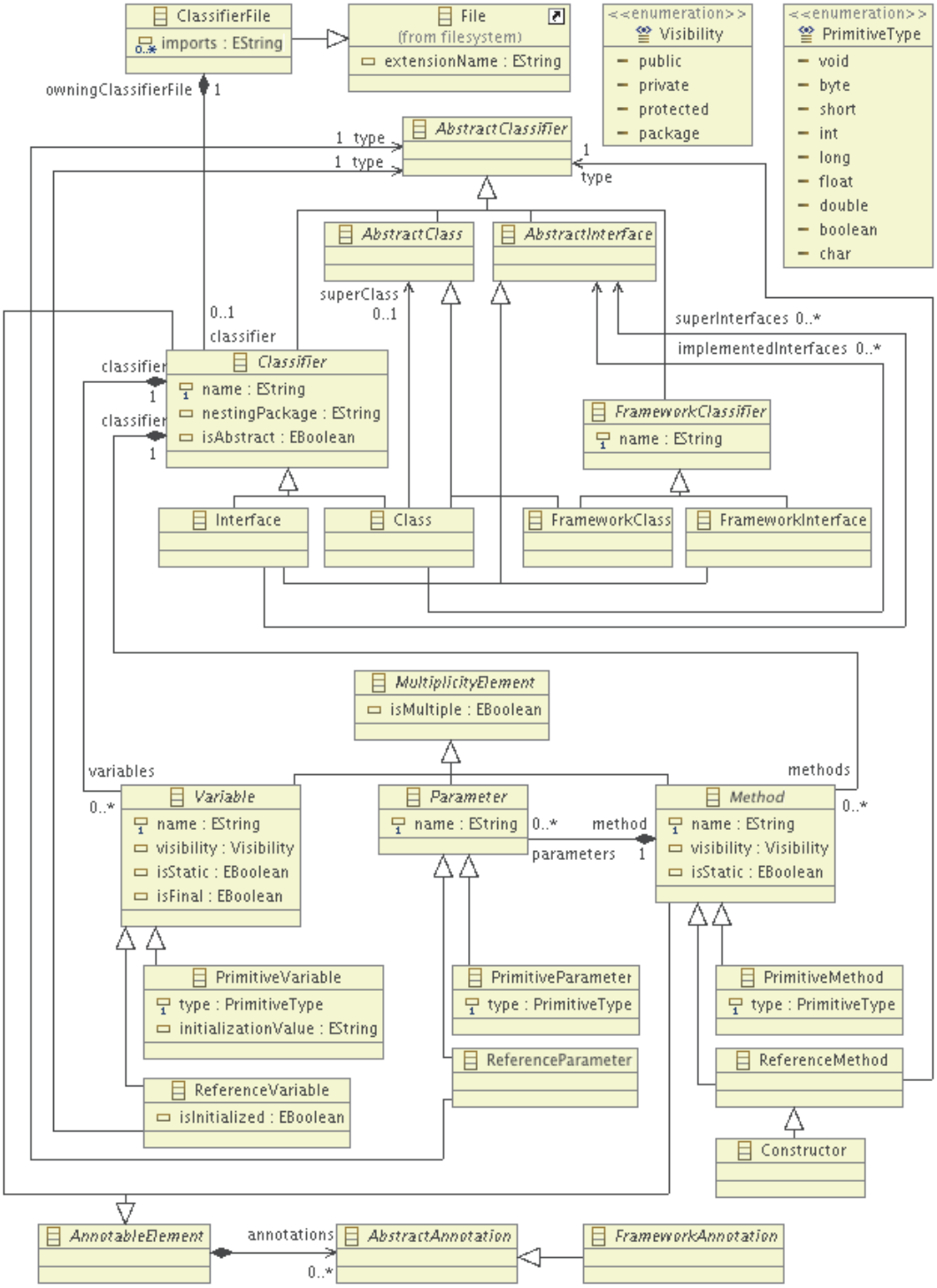

Abbildung 30: Metamodell *Java6*

ren Annotationstypen, kann aber per Spezialisierung durch EClasses anderer Metamodelle z. B. für Frameworks um diese ergänzt werden.

Metamodell *JavaDesignPatterns*

Das Metamodell *JavaDesignPatterns* umfasst Metamodellelemente zur typisierten Spezifikation von Entwurfsmustern für Java. Entsprechend erweitert es das Metamodell *Java6*, indem aus diesem einige Elemente spezialisiert werden. Für die Zwecke des Generatorframeworks enthält es in den EPackages *accessor* und *singleton* EClasses für Accessor-Methoden als triviale Methodentypen und Singletons als einfaches Entwurfsmuster (vgl. Abbildung 31). Die Kennzeichnung von Modellelementen mit diesen Typen ermöglicht einen erhöhten Explikationsgrad bzw. Informationsgehalt im Modell, begünstigt eine vollautomatisierte Generierung des Quelltexts und führt zu einem reduzierten Anpassungsbedarf am Codegenerator. Der erhöhte Informationsgehalt drückt sich z. B. im Fall von Accessor-Methoden darin aus, dass diese als expliziter Teil des Modells typsicher durch andere Modellelemente referenziert werden können. Die Reduktionen des Anpassungsaufwands für den Generator äußert sich darin, dass der Codegenerator nicht adaptiert werden muss, falls sich die Anfoderungen an das Generat ändern. Dies ist z. B. der Fall, falls nicht für jede Objektvariable Accessor-Methoden zu generieren sind. Durch die Explikation der Accessor-Methoden im Modell sind in diesem Fall Anpassungen auf das Modell beschränkt und betreffen nicht den Codegenerator.

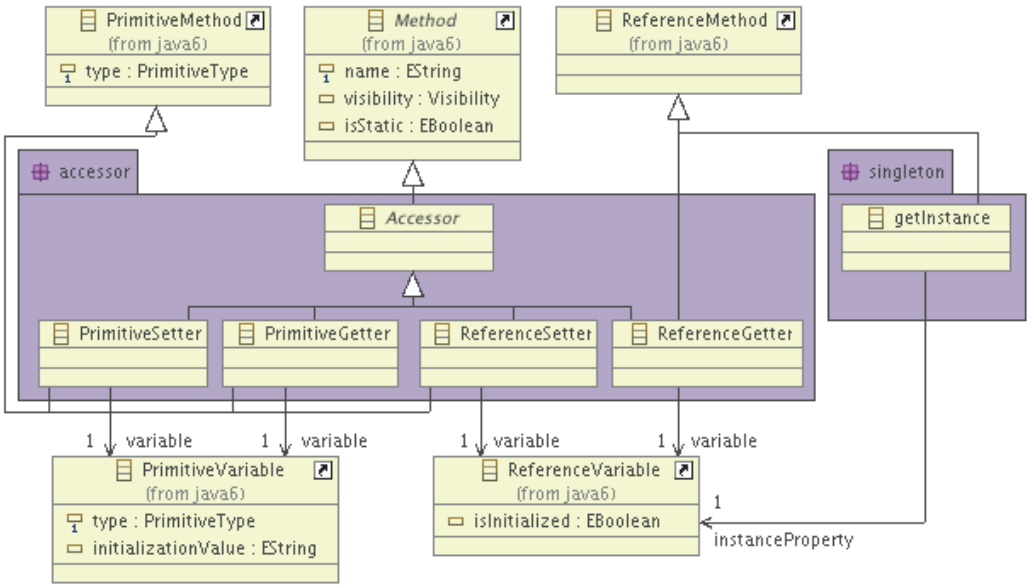

Abbildung 31: Metamodell *JavaDesignPatterns*

Bzgl. der Accessor-Methoden wird unterschieden zwischen Get-Methoden (Getter) zum Abfragen von Objektvariablen und Set-Methoden zu deren Modifikation, die in einer weiteren Unterscheidung in Accessoren für Objektvariablen mit primitivem Datentyp und solchen mit Referenzdatentyp unterteilt sind. Jede der vier möglichen Kombinationen spezialisiert die abstrakte EClass *Accessor*, die ihrerseits eine Spezialisierung von *Method* aus dem Metamodell *Java6* ist. Die EClass *PrimitiveGetter* typisiert Get-Methoden, die den Wert primitiver Objektvariablen vom Typ *PrimitiveVariable* zurückgeben. Die EClass *PrimitiveSetter* typisiert entsprechend Set-Methoden, die eine *PrimitiveVariable* modifizieren und erhält im Modell üblicherweise als Rückgabedatentyp den Wert *void*. Die EClass *ReferenceGetter* typisiert als Pendant für die EClass *PrimitiveGetter* Get-Methoden, die Instanzen vom Typ *ReferenceVariable* zurückgeben. Abschließend typisiert die EClass *ReferenceSetter* Set-Methoden für Instanzen von *ReferenceVariable*. Abhängig von dem Rückgabedatentyp der Methoden spezialisieren diese entweder *PrimitiveMethod* oder *ReferenceMethod* aus dem Metamodell *Java6*. Als Besonderheit ist die EClass *ReferenceSetter* eine Spezialisierung von *PrimitiveMethod*, da als Rückgabedatentyp der primitive Datentyp *void* vorausgesetzt wird. Als Restriktion für sämtliche Accessor-Methoden muss die referenzierte *PrimitiveVariable* bzw. *ReferenceVariable* derselben *Class* zugeordnet sein, in der auch die Accessor-Methode enthalten ist. Darüber hinaus muss bei den Get-Methoden der Rückgabe-Datentyp der Methode dem Typ der referenzierten Variablen entsprechen und den Set-Methoden jeweils ein Parameter hinzugefügt werden, der ebenfalls identisch typisiert ist.

```
1  public class Singleton{
2    private static Singleton instance;
3
4    private Singleton(){}
5
6    public static Singleton getInstance(){
7      if(instance == null)
8        instance = new Singleton();
9      return instance;
10   }
11 }
```

Listing 10: Beispiel für eine Singleton-Klasse

Das Singleton-Entwurfsmuster wird durch die EClass *getInstance* als Spezialisierung von *ReferenceMethod* typisiert. Eine Singleton-Klasse zeichnet sich in Java erstens durch eine private Klassenvariable mit der Singleton-Klasse als Datentyp, zweitens durch einen privaten Konstruktor und drittens durch eine Klassenmethode zur Initialisierung der Klassenvariable und deren Rückgabe aus (vgl. Listing 10 und [GHJV94, S. 127 ff.]). Die ersten beiden Charakteristika können durch die sprachlichen Ausdrucksmittel des Metamodells *Java6* formuliert werden, das dritte Charakteristikum dagegen ist explizit zu modellieren.

Im Detail muss dazu im Modell erstens eine Methode als *getInstance*-Methode gekennzeichnet werden, um dem Codegenerator die gewünschte Erzeugung des Methodenrumpfs zu signalisieren und zweitens die *getInstance*-Methode die Klassenvariable für das Singleton-Objekt referenzieren. Als Restriktion müssen sämtliche der genannten Variablen und Methoden im Modell als Teil derselben *Class* modelliert werden und konsistente Sichtbarkeiten, Typen, Parameter etc. aufweisen.

Im Unterschied zu den Metamodellen für Programmiersprachen repräsentieren die Metamodellelemente in diesen EPackages nicht direkt die sprachlichen Ausdrucksmittel der Plattform, sondern typisieren vielmehr bestimmte Programmkonstrukte als Resultat der Anwendung der sprachlichen Ausdrucksmittel. So ist z. B. die *getInstance*-Methode kein neues sprachliches Konzept, sondern lediglich ein bestimmter Methodentyp, welcher eine bestimmte Art der Programmierung, eben ein Muster repräsentiert. Dass die Metamodellelemente keine neuen Ausdrucksmittel der Plattformen sind, sondern lediglich die Anwendung existierender Ausdrucksmittel nach einem bestimmten Schema, drückt sich auch darin aus, dass die Metamodellemente sämtlich Spezialisierungen von *PrimitiveMethod* bzw. *ReferenceMethod* und damit Typen für Methodenrümpfe sind.

Metamodell *Hibernate3*

Das Metamodell *Hibernate3* adressiert in Verbindung mit dem nachfolgend beschriebenen Metamodell *Hibernate30Config_XSD_Adapted* das ORM-Framework Hibernate 3[31]. Es deckt mit den vier EPackages *classes*, *designpatterns*, *annotations* und *config* (vgl. Abbildungen 32 und 33) Entity-Klassen inkl. Annotationen, Entwurfsmuster für Data Access Objects (DAO) und Hilfsklassen ab und bindet das Metamodell *Hibernate30Config_XSD_Adapted* in das Dateisystem ein.

Das EPackage *classes* umfasst Metamodellelemente für Entity-Klassen, die bei Hibernate in Entsprechung zu der Java Persistence API (JPA) als Entity Beans in Form regulärer Javaklassen ohne externe Abhängigkeiten (Plan Old Java Object , POJO) [Ric06, S. 12 ff.] implementiert und leichtgewichtig um Annotationen von JPA und Hibernate angereichert werden[32]. Ein POJO ist zwar die Instanz einer gewöhnlichen Klasse, wird aber in der Verwendungsrichtung als Entity Bean im Modell für eine typsichere Referenzierung nicht als *Class*, sondern als deren Spezialisierung *PersistentPojo* typisiert. Ebenso werden Variablen durch die EClasses *PersistentPojoPrimitiveVariable* und *PersistentPojoReferenceVariable* als Spezialisierungen von *Variable* gekennzeichnet. Die spezielle *ReferenceMethod ToString*

[31] `http://www.hibernate.org/`

[32] `http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/#mapping-declaration`

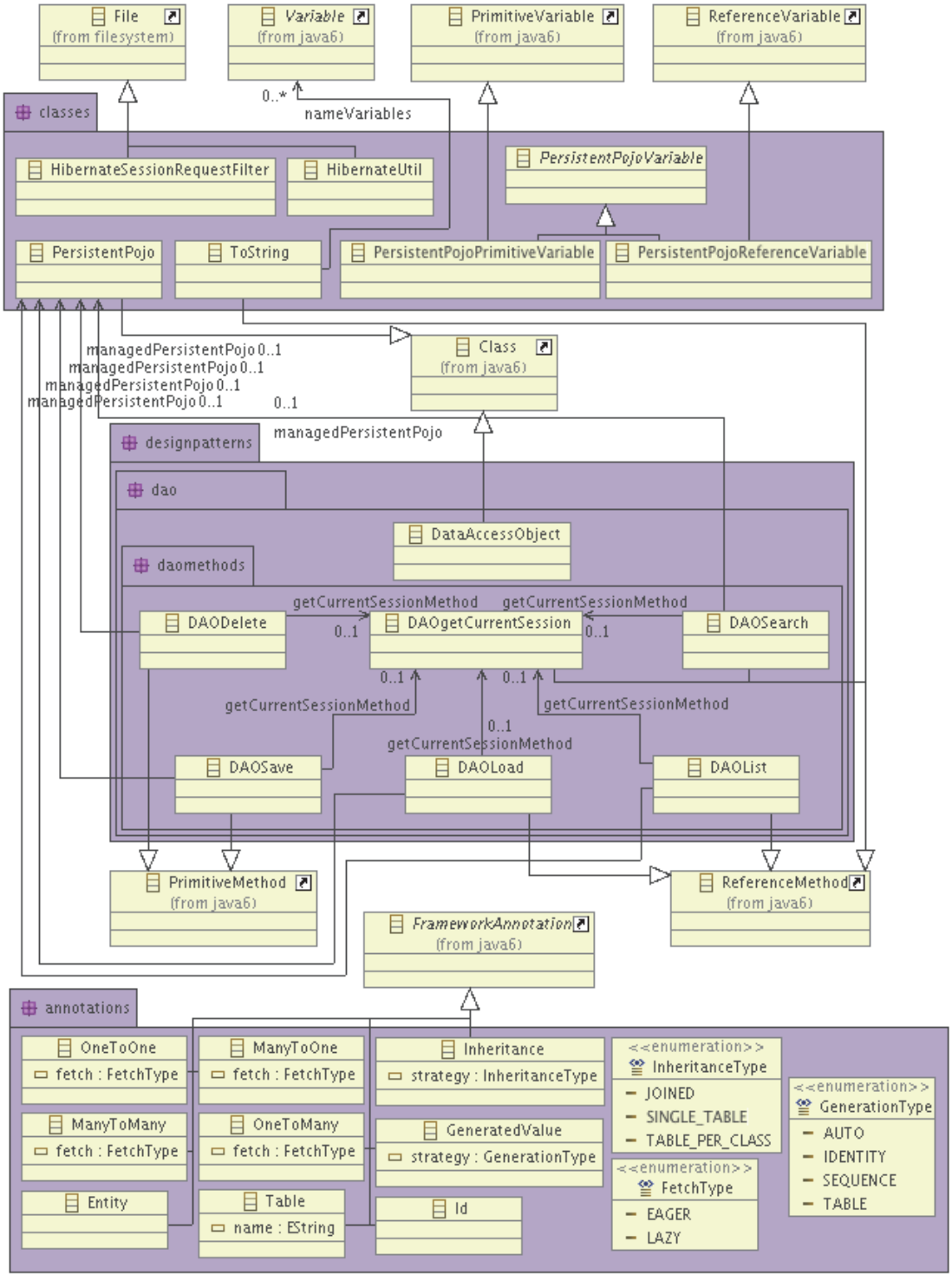

Abbildung 32: Ausschnitt aus dem Metamodell *Hibernate3*

dient der Spezifikation des Rumpfes der gleichnamigen Methode und erfasst mit der EReference *nameVariables* Variablen, deren textuelle Repräsentationen in konkatenierter Form die jeweilige Entity Bean z. B. in der Bedienoberfläche bezeichnen. Die EClasses *HibernateSessionRequestFilter* und *HibernateUtil* typisieren gleichnamige Hilfsklassen bzw. -dateien für Hibernate, die als Spezialisierungen von *File* im Modell nicht weiter detailliert werden, deren Verortung im Modell des Dateisystems aber als Auslöser für eine Erzeugung durch den Codegenerator dient.

Das EPackage *designpatterns* enthält in seinen Sub-EPackages Metamodellelemente zur Spezifikation von DAOs. Die EClass *DataAccessObject* typisiert als spezielle Form einer *Class* DAOs, die Methoden für die Interaktion mit der Datenbank über Hibernate enthalten. Die Methoden sind typisiert durch die EClasses *DAOSave*, *DAOLoad*, *DAODelete* zum Speichern, Laden und Löschen von Datensätzen sowie *DAOList* und *DAOSearch* zum Laden einer Liste von Datensätzen ohne und mit Bezug zu einem Suchkriterium. Die EClasses referenzieren als zusätzliche Informationen für die Generierung der Methodenrümpfe jeweils die zu verwaltende *PersistentPojo* und eine Methode vom Typ *DAOgetCurrentSession*, mit der die aktuelle Datenbanksitzung ermittelt wird.

Das EPackage *annotations* beinhaltet EClasses, die als Spezialisierungen der EClass *FrameworkAnnotation* aus dem Metamodell *Java6* die Annotationen von JPA und Hibernate repräsentieren. Sie dienen der Anreicherung von *PersistentPojos*, z. B. zur Kennzeichnung einer Klasse als Entity-Klasse (*Entity*), zur Deklaration einer Variable als Schlüsselvariable (*Id*) i.V.m. der Festlegung einer Strategie für die automatische Generierung von Schlüsselwerten (*GeneratedValue* und *GenerationType*) oder dem Mapping der Multiplizität von Assoziationen zwischen *PersistentPojos* (*OneToOne*, *OneToMany*, *ManyToOne*, *ManyToMany*) i.V.m. der Festlegung der anzuwendenden Ladestrategie (*FetchType*).

Metamodell *Hibernate30Config_XSD_Adapted*

Das Metamodell *Hibernate30Config_XSD_Adapted* repräsentiert mit Mitteln von Ecore die XML-Schemadefinition für Konfigurationsdateien des ORM-Frameworks Hibernate 3. Das Metamodell ist automatisiert erstellt aus einer XML-Schemadefinition, die in Ermangelung einer offiziellen XSD-Datei ihrerseits automatisiert mit dem Schema-Converter *trang*[33] aus einer im Umfang von Hibernate 3 verfügbaren Document Type Definition (DTD) erzeugt ist. Das Metamodell (vgl. Abbildung 33) orientiert sich entsprechend an der Syntax von Konfigurationsdateien für Hibernate (vgl. Listing 11) und spiegelt deren Semantik wider. Im Folgenden werden die wesentlichen Metamodellelemente erläutert, ein vollständiger Überblick über die Syntax und Semantik findet sich in der offiziellen Dokumentation von

[33] `http://www.thaiopensource.com/relaxng/trang.html`

Hibernate[34].

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate␣Configuration␣DTD␣3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.url">jdbc:mysql://localhost/database</property>
    <property name="connection.username">user</property>
    <property name="connection.password">password</property>
    ...
    <mapping class="model.crm.Address"></mapping>
    ...
  </session-factory>
</hibernate-configuration>
```

Listing 11: Beispiel für eine Hibernate-Konfigurationsdatei

Das XML-Dokument wird durch die EClass *DocumentRoot* repräsentiert, die mit einer Containment-Beziehung die EClass *HibernateConfigurationType* referenziert, welche das Modellelement für das XML-Wurzelelement einer Hibernate-Konfigurationsdatei typisiert. Zur Einbindung des *DocumentRoot* in das Modell des Dateisystems ist in dem Metamodell *Hibernate3* als Spezialisierung von *File* die EClass *HibernateConfigFile* enthalten, die das *DocumentRoot* referenziert (vgl. Abbildung 34).

Ein Modellelement vom Typ *HibernateConfigurationType* enthält obligatorisch ein Element vom Typ *SessionFactoryType*, das die Session-Factory konfiguriert, mit der zur Laufzeit Sitzungsobjekte zur objektorientierten Interaktion mit der Datenbank erzeugt werden, sowie optional ein Element vom Typ *SecurityType* für die Definition von Zugriffsrechten. Eine Instanz von *SessionFactoryType* kann beliebig viele Instanzen der EClasses *PropertyType* für generische Konfigurationsoptionen, *MappingType* für die Zuordnung von Javaklassen zu Datenbanktabellen, *ClassCacheType* und *CollectionCacheType* für die Konfiguration des Cachings von Objekten bzw. Datensätzen sowie Collections über diese und *EventType* i.V.m *ListenerType* für die Registrierung von Klassen als Event-Listener enthalten.

Das automatisiert generierte Metamodell wurde aus technischen und konzeptionellen Gründen adaptiert:

- Die EClass *PropertyType* enthält das EAttribute *mixed* vom Typ *EFeatureMapEntry*, das automatisch erzeugt ist, da in der XML-Schemadefinition der komplexe Typ des Schemaelements *property* als mixed deklariert ist (vgl. Listing 12 und [SBPM08, S. 215-217]). Dies bewirkt, dass im XML-Dokument innerhalb eines Property-Elements XML-Subelemente und Text in beliebiger Reihenfolge enthalten sein können (vgl. [W3C04a, Kapitel 2.5.2]). Da Xpand technischen Einschränkungen beim Umgang mit mixed-Elementen unterworfen ist, wird die EClass *PropertyType* um eine EReference

[34] Vgl. `http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/`

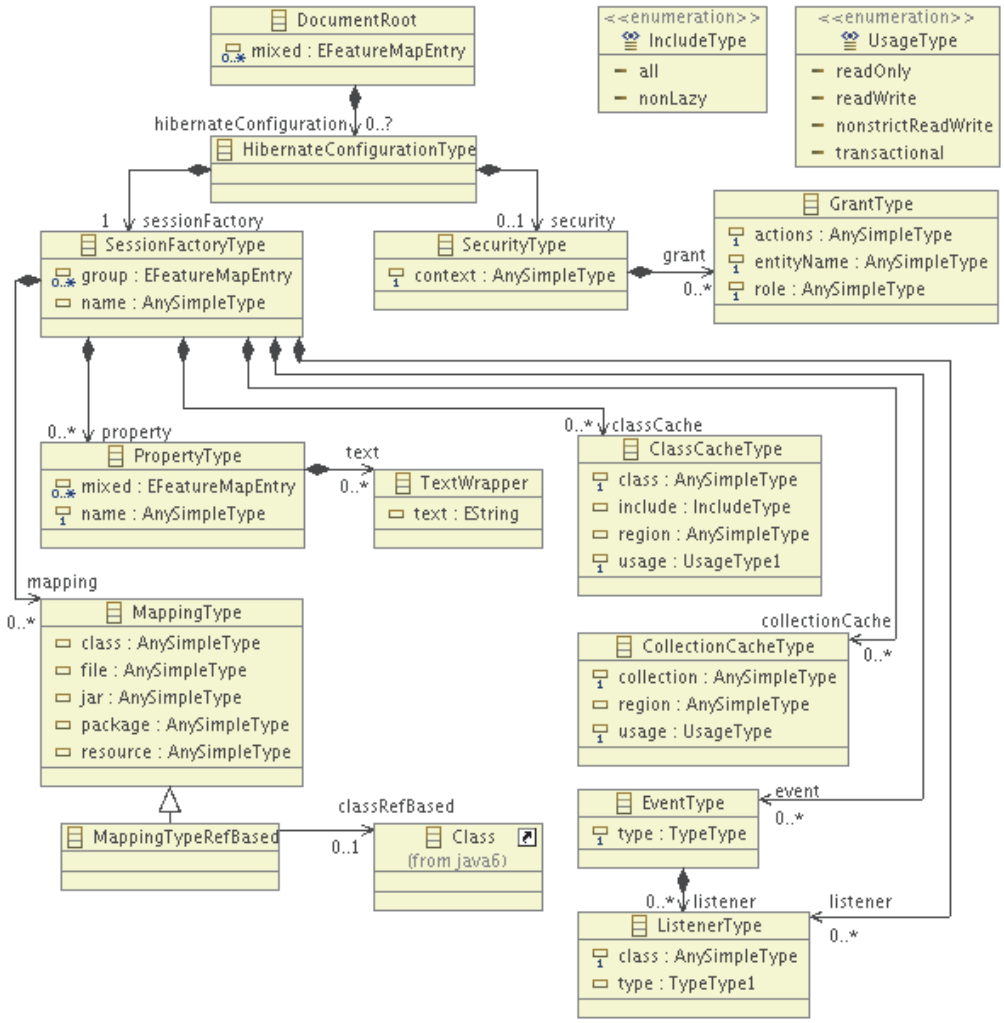

Abbildung 33: Ausschnitt aus dem Metamodell *Hibernate30Config_XSD_Adapted*

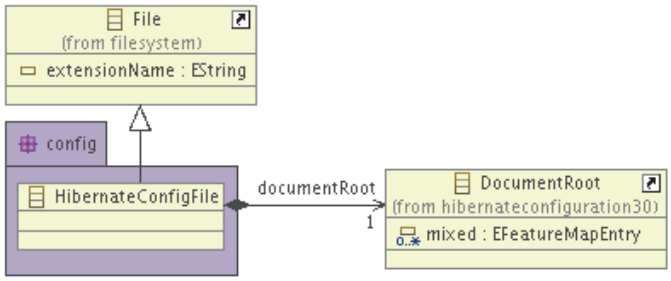

Abbildung 34: Ausschnitt aus dem Metamodell *Hibernate3*

text mit Verweis auf die EClass *TextWrapper* erweitert, die als Wrapperklasse für Text fungiert.

- Die EClass *MappingType* besitzt das EAttribute *class*, mit dem textuell auf den vollqualifizierten Namen derjenigen Entity-Klasse verwiesen wird, die auf eine Datenbanktabelle abgebildet werden soll. Es wird also keine navigierbare EReference auf die Repräsentation der Entity-Klasse durch ein Modellelement angelegt. Dies hat den Vorteil, dass das Metamodell auf diese Weise keine Abhängigkeit von dem Metamodell *Java6* aufweist, nachteilig wirkt sich jedoch aus, dass die Navigation im Modell und die Sicherung der Konsistenz des Modells erschwert wird. Zudem muss der vollqualifizierte Name der Entity-Klasse sowohl in der M2T-Transformationsdefinition für die Generierung des Quelltexts der Entity-Klasse, als auch in der M2M-Transformationsdefinition zur Festlegung der textuellen Referenz des Mappings zu der Entity-Klasse berechnet werden. Dies führt zu Redundanzen in der Berechnungslogik des Generators und damit potentiell zu Inkonsistenzen. Aus diesem Grund ist als alternative Lösung dem Metamodell die EClass *MappingTypeRefBased* als Spezialisierung der EClass *MappingType* hinzugefügt, die eine EReference *classRefBased* auf die Entity-Klasse besitzt. Dieser Lösungsansatz ist ein Beispiel dafür, wie das Modell eine integrierende Gesamtsicht auf das System bietet, ohne eine Abstraktionslücke zwischen dem PSM und dem Quelltext zu erzeugen.

```
1 <xs:element name="property">
2   <xs:complexType mixed="true">
3     <xs:attribute name="name" use="required"/>
4   </xs:complexType>
5 </xs:element>
```

Listing 12: Ausschnitt aus der XML-Schemadefinition für Hibernate-Konfigurationsdateien

Metamodell *JavaServerFaces1*

Das Metamodell *JavaServerFaces1* adressiert den Standard JavaServer Faces (JSF)[35] zur Entwicklung serverseitiger Web-Benutzerschnittstellen im Kontext von Java EE. JSF folgt dem MVC-Architekturmuster, das eine graphische Benutzerschnittstelle als ein System strukturiert, bei dem die Daten und die Geschäftslogik einer Domäne (Model) dem Benutzer in Form graphischer Ansichten präsentiert werden (Views), die in Abhängigkeit von Datenein- und ausgaben durch separate Logik gesteuert werden (Controller) (vgl. [KP88, Ree79a, Ree79b]). Das MVC-Architekturmuster zielt im Zusammenspiel mit dem

[35] `http://java.sun.com/j2ee/javaserverfaces/`

restlichen Softwaresystem auf eine Trennung von GUI und Geschäftslogik ab und soll innerhalb der GUI einer komplexen Vermischung von GUI-Schablonen mit Präsentationslogik vorbeugen. Entsprechend der MVC-Orientierung von JSF ist das Metamodell in die EPackages *facelets* für Views, *faces* für Controller-Klassen und *config* für die Konfiguration der Interaktion zwischen Models, Views und Controllern unterteilt (vgl. Abbildung 35).

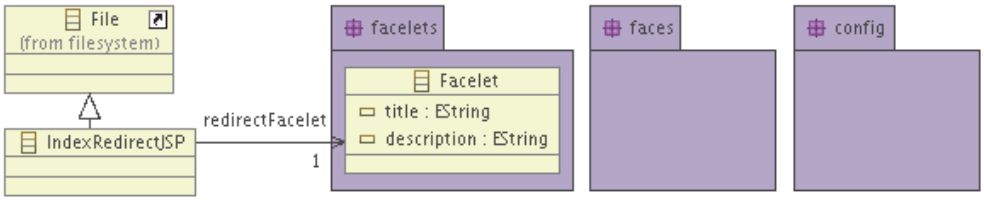

Abbildung 35: Metamodell *JavaServerFaces1*

Das EPackage *faces* enthält Metamodellelemente zur Spezifikation von Controller-Klassen (*Controller*), deren Methoden von den Views aufgerufen werden, die als *Facelets* bezeichnet werden und im Speziellen die Anbindung der Views an die Entity-Klassen (*PersistentPojo*) des Models herstellen (vgl. Abbildung 36). Die EClasses *CLoad*, *CList*, *CSave* und *CDelete* im Sub-EPackage *methodtypes* typisieren solche Methoden mit CRUD-Funktionalität, durch die *PersistentPojos* mittels DAOs geladen, modifiziert und für die Ausgabe durch *Facelets* präpariert werden. Um eine webseitenübergreifende Bearbeitung von *PersistentPojos* zu ermöglichen, werden diese vom jeweiligen Controller mit einer Variable referenziert, welche durch die EClass *CManagedPersistentPojo* im Sub-EPackage *variabletypes* typisiert wird und auf dasjenige *PersistentPojo* verweist, das durch den Controller verwaltet wird. Mit der Löschung oder optional Speicherung einer *PersistentPojo* ist eine Bereinigung dieser Variable verbunden, die durch eine Methode vom Typ *CReset* vorgenommen wird, welche von den EClasses *CSave* und *CDelete* referenziert wird. Die EClasses *CGetSelectItem*, *CSetSelectItem* und *CSelectItemCompleteList* typisieren abschließend Methoden, mit denen Auswahlmenüs in *Facelets* mit Auswahloptionen befüllt werden.

Die Metamodellelemente im EPackage *facelets* dienen der Spezifikation von *Facelets*, die im Rahmen von JSF als Template-Dateien für Webseiten die Rolle der Views einnehmen. Jedes *Facelet* ist als Spezialisierung von *File* Teil des Dateisystemmodells und enthält in der generischen Form als Inhalt einen Titel, eine Beschreibung sowie eine Referenz auf einen Ordner aus der Menühierarchie der Webapplikation (vgl. Abbildung 37). Spezielle *Facelets* zur Repräsentation von CRUD-Funktionalität werden typisiert durch die EClasses *CreateFacelet*, *EditFacelet*, *DeleteFacelet*, *DataSetFacelet* und *ListFacelet* aus dem Sub-EPackage *facelettypes*, die einerseits zur Interaktion mit der Persistenzschicht Methoden der jeweils zugeordneten Controller-Klasse referenzieren und andererseits für Übergänge

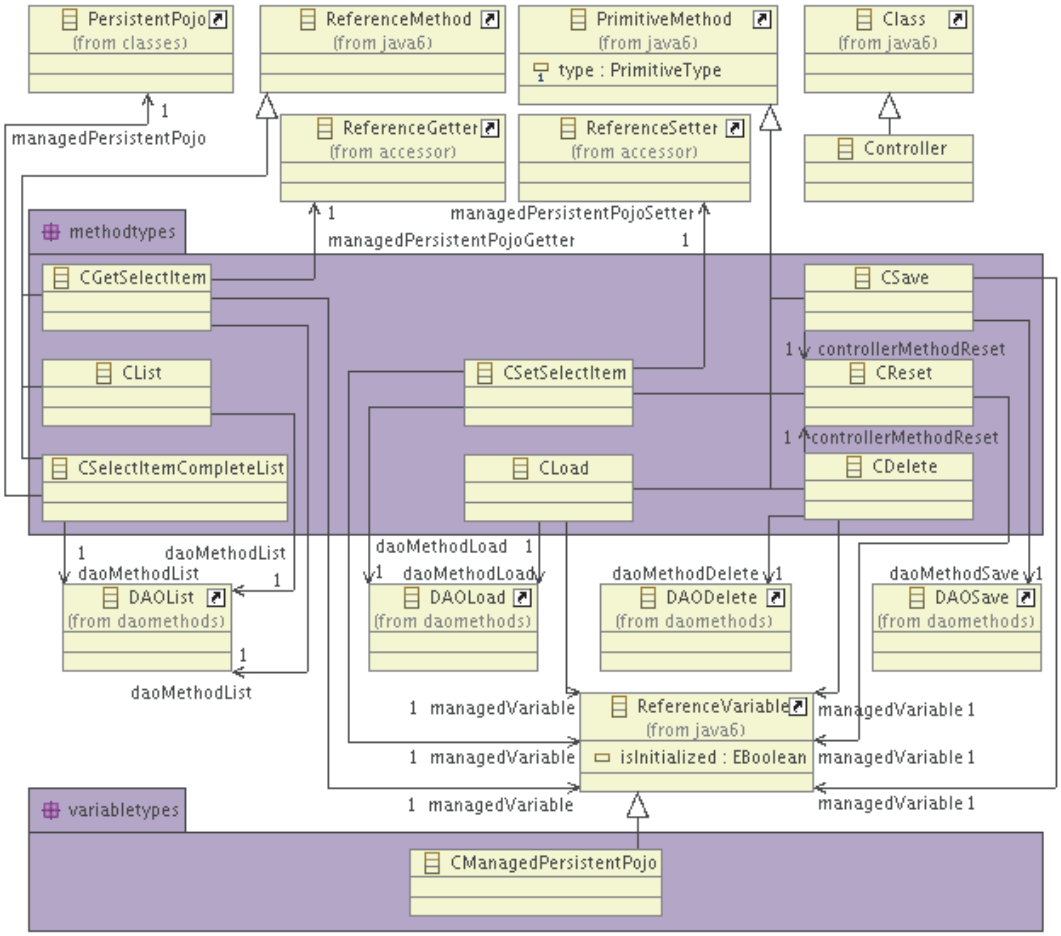

Abbildung 36: EPackage *faces* aus dem Metamodell *JavaServerFaces1*

zwischen Webseiten bzw. Facelets auf Transitionskanten aus dem Navigationsgraphen verweisen, welche durch die EClass *FacesConfigNavigationCaseType* typisiert werden (vgl. Kapitel 4.6.3.3). Die Facelets enthalten zur Datenein- und ausgabe Formular- bzw. Datenfelder, die im Modell in Form von *FaceletVariables* abgebildet werden. Jede *FaceletVariable* verweist auf ein Attribut über eine Referenz, die aus drei Teilen zusammengesetzt ist. Der erste Teil besteht aus einer EReference *controller* auf eine Controller-Klasse, die im JSF-Kontext als Managed Bean verwaltet wird und im Modell durch die EClass *FacesConfigManagedBeanType* typisiert wird. Der zweite Teil der Referenz besteht aus einer EReference auf eine in dieser Controller-Klasse enthaltene Variable vom Typ *CManagedPersistentPojo*, die auf das *PersistentPojo* verweist, das durch die Controller-Klasse verwaltet wird. Der dritte Teil der Referenz verweist auf eine *PersistentPojoReferenceVariable*, die in dem *PersistentPojo* enthalten ist und die über diesen dreiteiligen Navigationspfad der *FaceletVariable* zugeordnet wird. Die EClass *FaceletVariable* ist als abstrakt deklariert und wird durch die EClasses *FaceletScalarVariable* und *FaceletSelectItemVariable* spezialisiert, mit denen zwischen textuellen Eingabefeldern und solchen mit Auswahlmenü unterschieden wird. Bei Letzteren wird zur Befüllung des Auswahlmenüs mit Auswahloptionen über die EReference *selectItemCompleteListController* auf eine Controller-Klasse verwiesen, die einerseits unter der EReference *selectItemCompleteList* eine Methode vom Typ *CSelectItemCompleteList* zur Abfrage der auswählbaren Optionen bzw. Datensätze und andererseits unter der EReference *selectItem* eine Methode vom Typ *CGetSelectItem* zur Abfrage der bereits ausgewählten Optionen bzw. Datensätze referenziert. Auf eine EReference zu einer obligatorischen korrespondierenden Methode vom Typ *CSetSelectItem* zur Speicherung der Auswahl wird im Modell verzichtet, da sie in der Implementierung im Facelet ebenfalls nicht explizit referenziert wird, sondern der Verweis implizit aus dem Namen der Get-Methode abgeleitet wird, indem deren obligatorisches Namenspräfix *get* durch *set* ersetzt wird.

Das Zusammenspiel von Model, Views und Controllern wird durch eine XML-Konfigurationsdatei mit dem Namen *faces-config.xml* festgelegt, deren Inhalte im Modell mit den Ausdrucksmitteln des Metamodells *JavaServerFaces12Config_XSD_Adapted* beschrieben werden, welches mit der EClass *FacesConfigFile* aus dem Sub-EPackage *config* als Spezialisierung von *File* in das Dateisystemmodell eingebunden wird (vgl. Abbildung 38).

Metamodell *JavaServerFaces12Config_XSD_Adapted*

Die XML-Konfigurationsdatei *faces-config.xml* dient im JSF-Framework der Definition des Navigationsgraphen sowie der Verwaltung von Objekten in Form von Managed Beans. Die Syntax des XML-Dokuments ist durch eine XML-Schemadefinition als Teil des JSF-Frameworks spezifiziert, aus dem automatisch das Metamodell *JavaServerFaces12Config_-*

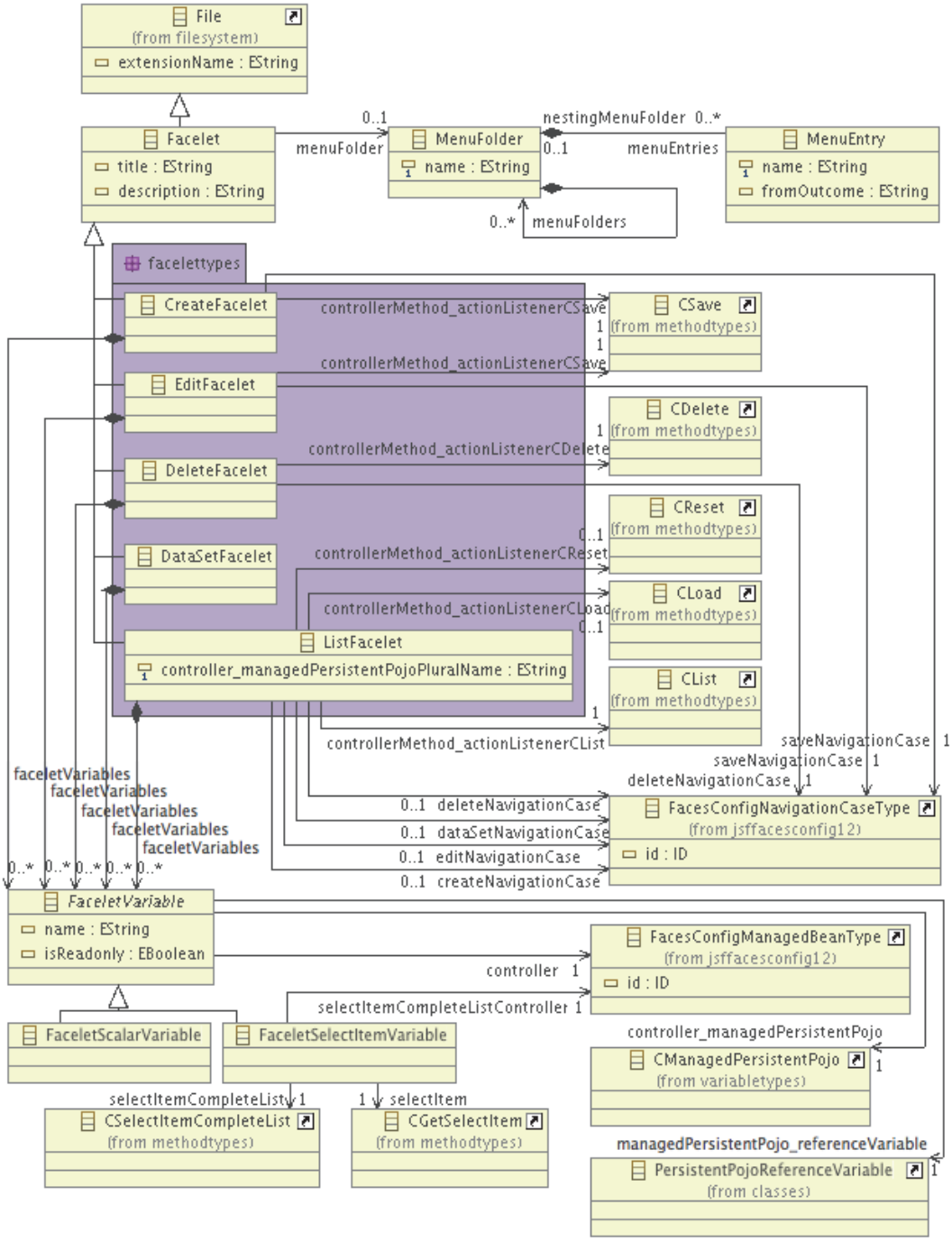

Abbildung 37: EPackage *facelets* aus dem Metamodell *JavaServerFaces1*

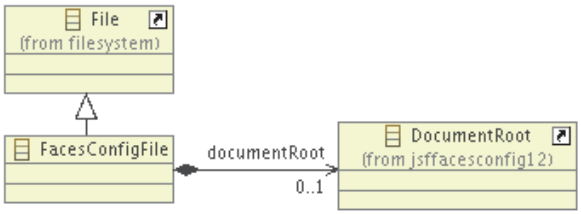

Abbildung 38: EPackage *config* aus dem Metamodell *JavaServerFaces1*

XSD_Adapted abgeleitet ist.

Das Wurzelelement des Metamodells ist die EClass *DocumentRoot*, welche auf die EClass *FacesConfigType* verweist, mit der das Wurzelelement des XML-Dokuments typisiert wird (vgl. Abbildung 39). Die EClass *FacesConfigType* referenziert unter anderem die EClasses *FacesConfigManagedBeanType* zur Definition von Managed Beans und *FacesConfigNavigationRuleType* zur Definition von Navigationsregeln.

Ein *FacesConfigManagedBeanType* bzw. eine Managed Bean verweist auf die verwaltete Javaklasse, indem textuell der vollqualifizierte Name der Javaklasse mit der EReference *managedBeanName* zu der EClass *JavaIdentifierType* als Spezialisierung von *String* genannt wird. Der Gültigkeitsbereich der Managed Bean wird mit der EReference *managedBeanScope* auf die EClass *FacesConfigManagedBeanScopeOrNoneType* als Spezialisierung von *String* auf die Werte *none*, *request*, *session* oder *application* festgelegt.

Im Navigationsgraphen sind *Facelets* Knoten, die durch Kanten als Kombinationen aus den EClasses *FacesConfigNavigationRuleType*, *FacesConfigFromViewIdType* und *FacesConfigNavigationCaseType* verbunden werden. So werden die Kanten des Navigationsgraphen im Kontext von Navigationsregeln bzw. Instanzen der EClass *FacesConfigNavigationRuleType* definiert. In jeder Navigationsregel werden ausgehend von einem Quellknoten bzw. -facelet, das mit dem Metamodellelement *FacesConfigFromViewIdType* als Spezialisierung von *String* textuell referenziert wird, mit der EClass *FacesConfigNavigationCaseType* die möglichen Zielknoten bzw. -facelets festgelegt und somit die Kanten definiert.

Die EClasses *FacesConfigManagedBeanType*, *FacesConfigFromViewIdType* und *FacesConfigNavigationCaseType* des automatisiert generierten Metamodells werden durch korrespondierend benannte EClasses mit dem Suffix *RefBased* spezialisiert, die als Alternative zu dem EAttribute *id* für textuelle Verweise auf Javaklassen bzw. *Facelets* eine EReference auf das jeweilige Modellelement besitzen. Dies vereinfacht die Navigation zwischen Modellelementen und stellt die Typsicherheit der Referenzen sicher.

Metamodell *WaslJavaEE*

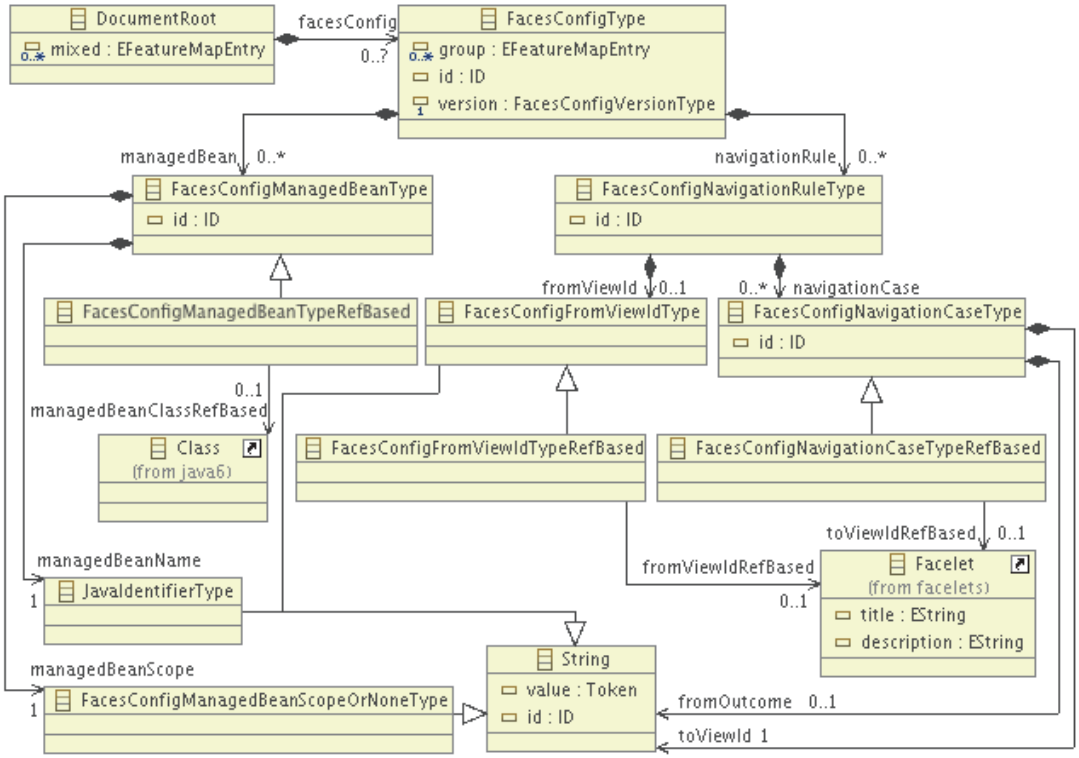

Abbildung 39: Ausschnitt aus dem Metamodell *JavaServerFaces12Config_XSD_Adapted*

Das Metamodell *WaslJavaEE* enthält die EClass *WaslJavaEEModel*, die das Wurzelelement der Containment-Hierarchie eines *WASL JavaEE*-Modells typisiert (vgl. Abbildung 40). Das Modell des Dateisystems wird durch eine mehrwertige Containment-EReference zu der EClass *DirectoryElement* eingebunden. Die Instanzen der EClasses *FrameworkClassifier*, *FrameworkAnnotation* und *MenuFolder* müssen durch gesonderte Containment-EReferences in die Containment-Hierarchie des Modells integriert werden, da sie weder Teil des Dateisystems sind, noch in anderen Modellelementen in der Containment-Hierarchie enthalten sind, sondern lediglich durch diese referenziert werden.

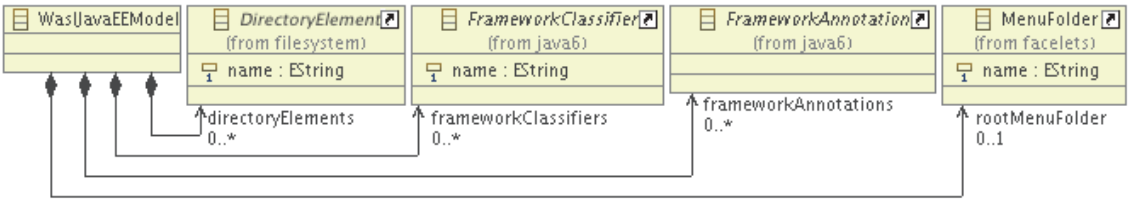

Abbildung 40: Metamodell *WaslJavaEE*

4.6.3.4 Metamodelle für Python 2

Metamodell *Python2*

Das Metamodell *Python2* bildet einige Sprachelemente der Programmiersprache Python in der Version 2 ab (vgl. Abbildung 41), mit Fokus auf objektorientierte Ausdrucksmittel für die Entwicklung von Webapplikationen im Rahmen der Google App Engine. Die Einbindung in das Dateisystem erfolgt über die EClass *PythonScript* als Spezialisierung der EClass *File*, die analog zu Modulen in Python[36] beliebig viele Instanzen der EClass *Class* enthalten kann. Importe anderer Pythonmodule werden mit dem EAttribute *imports* unstrukturiert als EStrings modelliert.

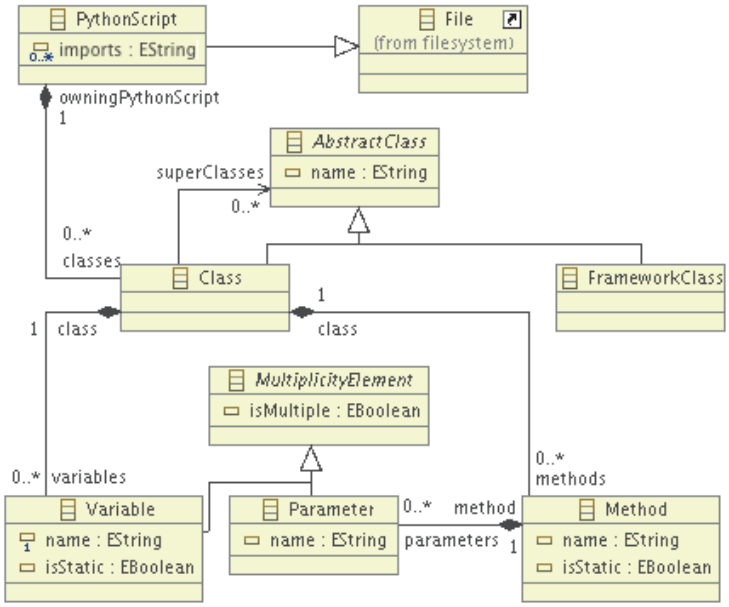

Abbildung 41: Metamodell *Python2*

Wie auch bei dem Metamodell *Java6* wird zwischen *Class* und *FrameworkClass* unterschieden, die beide durch die EClass *AbstractClass* generalisiert werden, über die mit der EReference *superClasses* Mehrfachvererbung modelliert werden kann. Eine *Class* kann Instanzen der EClasses *Variable* und *Method* enthalten, wobei Letztere durch *Parameter* ergänzt werden. Die Metamodellelemente *Variable* und *Parameter* besitzen über die Generalisierung *MultiplicityElement* eine Multiplizität und können somit als mehrwertig modelliert werden.

Die dynamische Typisierung von Variablen, Parametern und Methoden in Python drückt sich in der fehlenden Typisierung der EClasses *Variable*, *Parameter* und *Method*

[36] `http://docs.python.org/tutorial/modules.html`

aus. Es existieren im Unterschied zum Metamodell *Java6* also keine Typattribute oder Typreferenzen zu *AbstractClass*.

Metamodell *GoogleDatastore*

Das Metamodell *GoogleDatastore* adressiert das gleichnamige Framework zur Interaktion mit dem schemalosen Objektdatenspeicher der Google App Engine. Der Zugriff auf Datenspeicherentitäten bzw. Datensätze geschieht über korrespondierende Python-Objekte, deren Eigenschaften dynamisch zur Laufzeit deklariert werden können. Der Typ einer Entität im Datenspeicher wird aus der Python-Klasse des korrespondierenden Objekts abgeleitet und kategorisiert die Entität für Abfragen, determiniert aber kein Schema, sodass Entitäten mit gleichem Typ unterschiedliche Eigenschaften besitzen können[37]. Die Entitätstypen werden durch korrespondierende Modellklassen (Model Class) in Form von Python-Klassen definiert, die von der Klasse *db.Model* erben[38] (vgl. Listing 13).

Die Variablen bzw. Attribute einer Python-Modellklasse entsprechen den Eigenschaften der Entität im Datenspeicher. Der Typ jeder Variablen wird festgelegt, indem jeweils die Variable nach der Deklaration mit einer Instanz der Eigenschaftsklassen (Property Classes) des Google Datastore Frameworks wie z. B. *StringProperty* oder *IntegerProperty* initialisiert wird[39] (vgl. Listing 13).

Die Eigenschaftsklassen sind Unterklassen der Klasse *google.appengine.ext.db.Property*, die als solche jeweils einen nativen Basisdatentypen von Python referenzieren und zusätzliche Logik zur Festlegung des Standardwerts und der Belegung, sowie Logik zur Validierung und zur Interaktion mit der Datenbank umfassen[40]. Für die unterschiedlichen Datentypen sind von Seiten des Frameworks diverse Eigenschaftsklassen vordefiniert[41].

```
1  from google.appengine.ext import db
2
3  class Person(db.Model):
4    firstname = db.StringProperty()
5    lastname = db.StringProperty()
6    address = db.ReferenceProperty(Address)
7
8  class Address(db.Model):
9    street = db.StringProperty()
10   city = db.StringProperty()
11   postalcode = db.StringProperty()
```

Listing 13: Beispiel für Datastore-Modell mit den zwei Modellklassen *Person* und *Address*

[37] http://code.google.com/intl/en/appengine/docs/python/datastore/overview.html

[38] http://code.google.com/intl/en/appengine/docs/python/datastore/datamodeling.html

[39] http://code.google.com/intl/en/appengine/docs/python/datastore/datamodeling.html

[40] http://code.google.com/intl/en/appengine/docs/python/datastore/propertyclass.html

[41] http://code.google.com/intl/en/appengine/docs/python/datastore/typesandpropertyclasses.html

Im Modell werden die Python-Modellklassen durch Instanzen der EClass *ModelClass* repräsentiert (vgl. Abbildung 42). Sie sind entsprechend dem Metamodell *Python2* in Instanzen der EClass *PythonScript* enthalten, welche durch die EClass *ModelScript* spezialisiert wird. Da *ModelClass* das Metamodellelement *Class* spezialisiert, kann eine *ModelClass* somit Instanzen der EClass *Variable* besitzen. Um die Typisierung der Variablen einer Modellklasse repräsentieren zu können, wird die EClass *Variable* über die intermediäre Sub-EClass *Property* durch eine Vielzahl an EClasses aus dem EPackage *propertyclasses* für die unterschiedlichen Eigenschaftsklassen spezialisiert. Diese sind z. B. *StringProperty* sowie *IntegerProperty* für primitive Datentypen und *PostalAddressProperty* und *GeoPtProperty* für abstrakte Datentypen. Die EClass *ReferenceProperty* dient der Modellierung von Referenzen zwischen Modellklassen und verweist mit der EReference *referencedModelClass* auf eine solche (vgl. Listing 13, Zeile 6). Die EClass *ListProperty* dient der Modellierung von Listen und verweist per EReference *itemType* auf die Eigenschaftsklasse, welche den Typ der Listenelemente angibt.

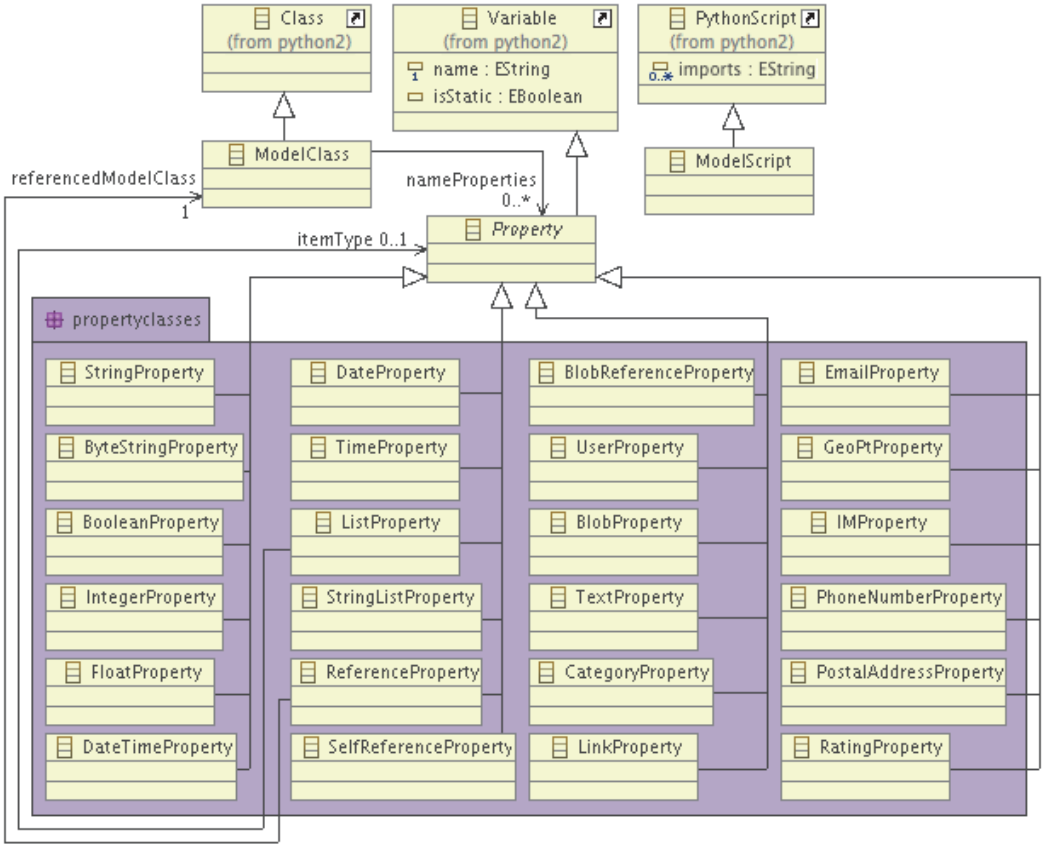

Abbildung 42: Metamodell *GoogleDatastore*

Da in der Implementierung jede Modellklasse von der vordefinierten Klasse *db.Model*

erbt, muss als Restriktion auch im Modell jede *ModelClass* mit den Mitteln des Metamodells *Python2* als Unterklasse einer Instanz der EClass *FrameworkClass* mit dem Namen *db.Model* spezifiziert werden.

Metamodell *Django*

Das Metamodell *Django* adressiert einen aufgabenrelevanten Ausschnitt der Funktionalität des Webframeworks Django[42] als Teilkomponente der Google App Engine[43]. Das zentrale Metamodellelement ist die EClass *Template* im EPackage *template* als Spezialisierung der EClass *File*, mit dem im Modell Django-Templates repräsentiert werden (vgl. Abbildung 43). Jedes Template hat einen Titel und kann beliebige Kombinationen von *DjangoComponents* aus dem Sub-EPackage *djangocomponents* und Block-, Inline- sowie Flowelementen aus dem Metamodell *XHTML1-Strict_XSD_Adapted* enthalten. Letzere repräsentieren mit ihren Spezialisierungen die verschiedenen Elementtypen von XHTML, sodass im Modell beliebige XHTML-Inhalte in Django-Templates modelliert werden können. Die feingranularen Metamodellelemente für XHTML-Elementtypen werden durch die grobgranularen *DjangoComponents* ergänzt, die in aggregierter und abstrakter Form Präsentationsbausteine repräsentieren.

Das EAttribute *group* der EClass *Template* wird als eine FeatureMap realisiert, die eine spezielle EList mit Elementen vom Typ FeatureMap.Entry ist (vgl. [SBPM08, S. 186-196]) und über diesen Elementtyp in geordneter Form die Elemente der abgeleiteten (derived) EReferences bzw. Features *inline*, *block*, *flow* und *djangoComponent* enthalten kann. Die FeatureMap dient im Fall des EAttribute *group* dazu, in einer einzelnen Liste geordnet Instanzen verschiedener EClasses zu sammeln, die keine gemeinsame exklusive Super-EClass besitzen. Im Fall des Metamodells *Django* weisen die EClass *DjangoComponent* sowie die EClasses *Inline*, *Block* und *Flow* keine solche gemeinsame exklusive Super-EClass auf. Einerseits repräsentiert eine *DjangoComponent* kein spezielles XHTML-Element und soll somit nicht eine der EClasses aus dem XHTML-Metamodell spezialisieren. Andererseits soll den drei XHTML-EClasses keine Spezialisierungsbeziehung zu einer metamodellfremden Super-EClass hinzugefügt werden, da dies eine Metamodell-Interdependenz erzeugt und somit die Selbstständigkeit und lose Kopplung des XHTML-Metamodells einschränkt. Aufgrund der fehlenden gemeinsamen Super-EClass wird die Verwendung einer polymorphen mehrwertigen Containment-EReference verhindert. Eine alternative Aufteilung der Sammlung auf vier eigenständige EReferences ist zwar flexibel bzgl. der referenzierbaren Elementtypen, ermöglicht aber keine geordnete Erfassung der Elemente, wie sie für die

[42] `http://www.djangoproject.com/`

[43] `http://code.google.com/intl/de-DE/appengine/articles/django.html`

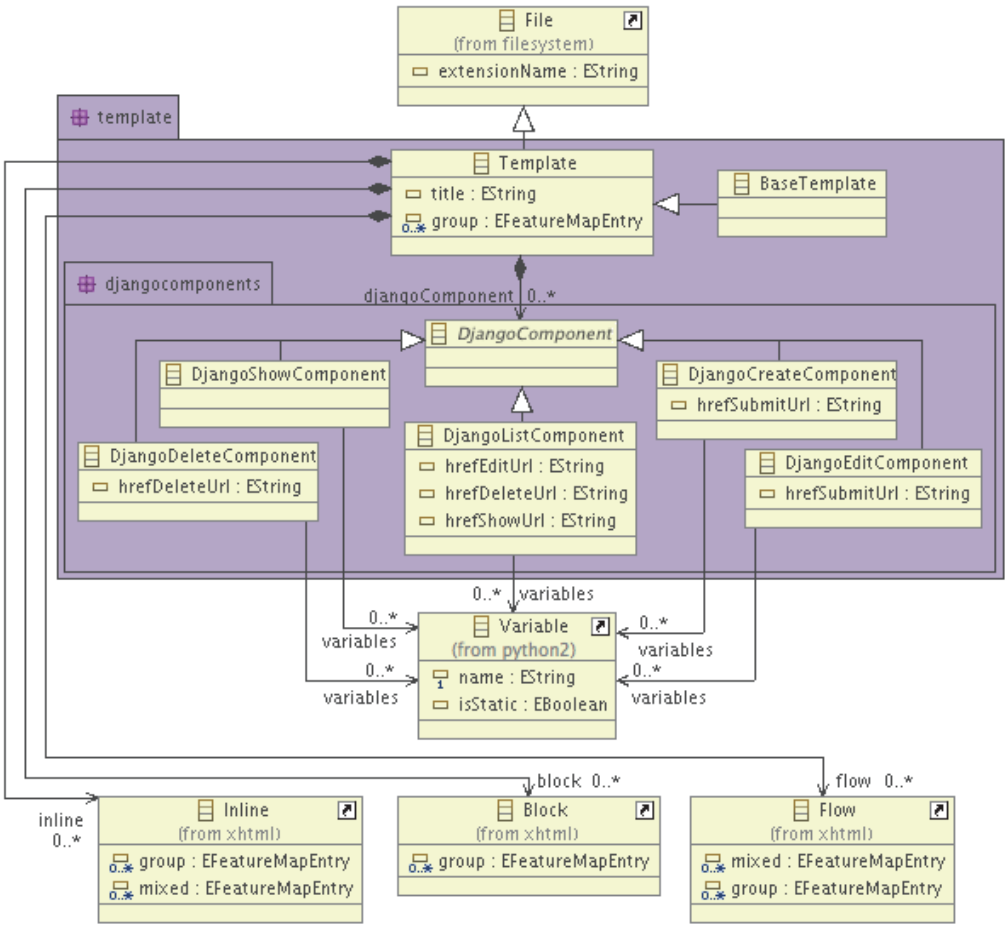

Abbildung 43: Metamodell *Django*

Abbildung der Reihenfolge von Elementen in der Template-Datei notwendig ist.

Die abstrakte EClass *DjangoComponent* wird durch die EClasses *DjangoCreateComponent*, *DjangoShowComponent*, *DjangoListComponent*, *DjangoEditComponent* und *DjangoDeleteComponent* zur Abbildung von CRUD-Funktionalität spezialisiert, die zum einen die Variablen für die Datenein- und ausgabe referenzieren und zum anderen diverse textuelle URLs für die Navigation nach Aktionsausführungen enthalten.

Metamodell *GoogleWebapp*

Das Metamodell *GoogleWebapp* adressiert das webapp-Framework[44] der Google App Engine, das die Entwicklung von WSGI-kompatiblen HTTP-Request-Handlern in Form von Pythonklassen ermöglicht. Das Metamodell enthält Elemente zur Modellierung von Pythonskripten, die entsprechend dem MVC-Architekturschema des webapp-Frameworks als Controller das Zusammenspiel von Django-Templates als Views und Modellklassen bzw. dem Model steuern (vgl. Abbildung 44). Die Modellelemente für Skripte dieser Art werden im Modell durch die EClass *WebappApplicationScript* als Spezialisierung von *PythonScript* typisiert, und können somit entsprechend dem Metamodell *Python2* Instanzen der EClass *Class* und in diesen Instanzen der EClass *Method* enthalten.

In der Implementierung wird ein *WebappApplicationScript* als ein Python-Skript realisiert, das erstens RequestHandler-Klassen als Unterklassen der Framework-Klasse *webapp.RequestHandler* enthält, zweitens ein Objekt der Klasse WSGIApplication erzeugt, mit dem URLs auf diese RequestHandler-Klassen abgebildet werden, und drittens die Ausführung der WSGIApplication in einer *main*-Funktion initiiert (vgl. Listing 14). Bei einem HTTP-Request wird entsprechend zur der URL die zugeordnete RequestHandler-Klasse ermittelt und abhängig von der HTTP-Methode die korrespondierende Methode mit dem obligatorischen Namen *get* oder *post* aufgerufen. Im Modell werden RequestHandler-Klassen durch Instanzen von *Class* repräsentiert, die im *WebappApplicationScript* enthalten sind. Für die Abbildung von URLs auf die Klassen dient die EClass *UrlMapping*, die eine textuelle Darstellung der URL durch das EAttribute *url* in Beziehung zu *Class* setzt.

Eine RequestHandler-Methode wird im Modell durch Instanzen der EClass *RequestHandlerMethod* repräsentiert, die entsprechend dem MVC-Architekturschema die Ausgabe der View in Form eines *Template* steuert und CRUD-Operationen auf einer *ModelClass* ausführt. Die Methoden für die CRUD-Funktionalität einer RequestHandler-Klasse werden im einzelnen durch die EClasses *RequestHandlerMethodCreateGet* bzw. *-Post*, *RequestHandlerMethodShow*, *RequestHandlerMethodList*, *RequestHandlerMethodEditGet* bzw. *-Post* und *RequestHandlerMethodDelete* als Spezialisierungen der abstrakten

[44] `http://code.google.com/intl/en/appengine/docs/python/tools/webapp/`

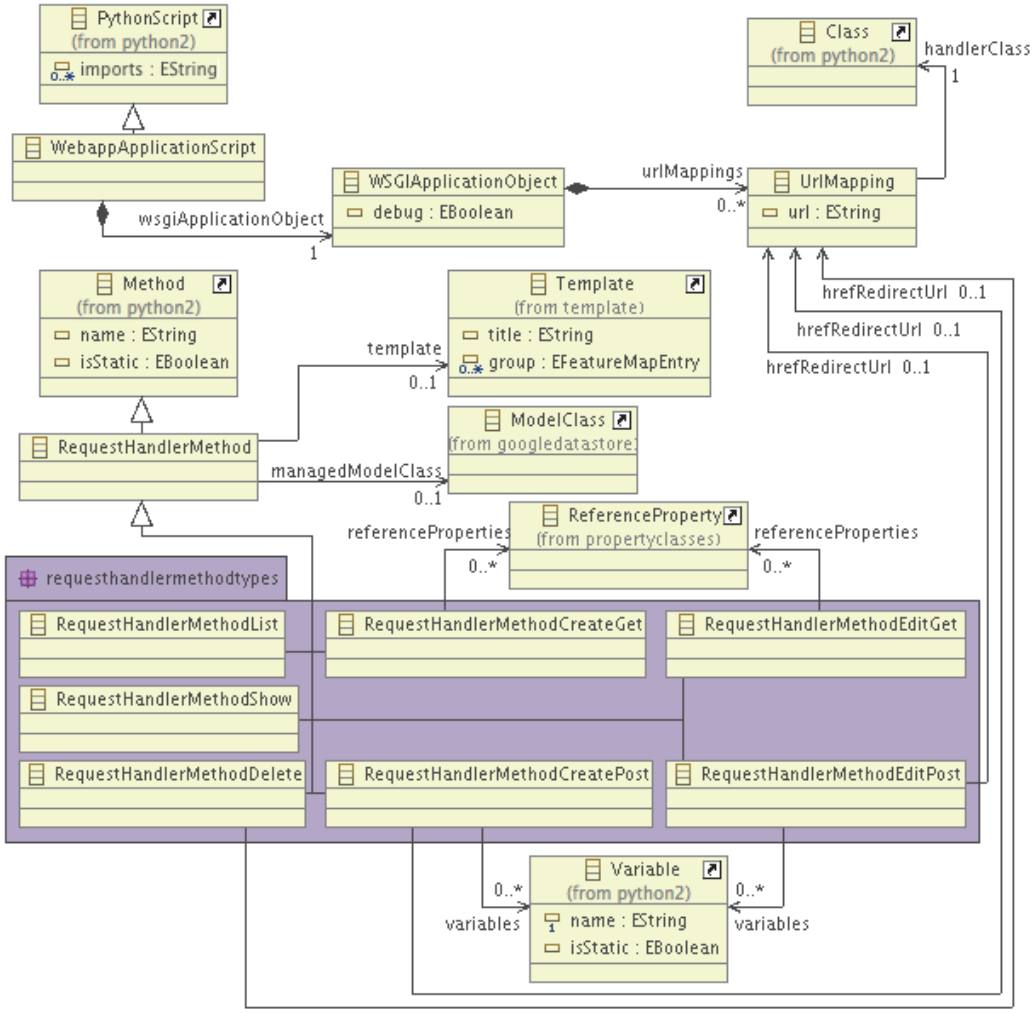

Abbildung 44: Metamodell *GoogleWebapp*

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app

#--- 1. request handler
class ListAddresses(webapp.RequestHandler):
  def get(self):
    query = model.Address.all()
    records = query.fetch(1000)

    template_values = {
      'records': records
    }

    path = os.path.join('./..', 'templates/address/address_list.html')
    self.response.out.write(template.render(path, template_values))

#--- 2. application preparation
application = webapp.WSGIApplication([('/crm/address/list_addresses', ListAddresses)],
        debug=True)

#--- 3. application initialization
def main():
  run_wsgi_app(application)

if __name__ == "__main__":
  main()
```

Listing 14: Beispiel für die Implementierung eines *WebappApplicationScript*

EClass *RequestHandlerMethod* typisiert. Im Fall der Create- und Edit-Methoden wird zwischen Versionen für die HTTP-Methoden *GET* und *POST* unterschieden, da entweder Daten für die Ausgabe eines Formulars geladen werden, oder Formular-Eingabedaten zu speichern sind. Falls bei einem Formular ein Auswahlmenü darzustellen ist, wird das zugrunde liegende *ReferenceProperty* der *ModelClass* explizit referenziert, um Auswahloptionen zu berechnen. Bei der Speicherung per POST werden vordefinierte *Variables* der *ModelClass* modifiziert, die ebenfalls explizit referenziert werden. Durch die optionale EReference *hrefRedirectUrl*, die jeweils von *RequestHandlerMethodCreatePost*, *RequestHandlerMethodEditPost* und *RequestHandlerMethodDelete* ausgeht, kann ein HTTP-Redirect nach der Speicherung oder Löschung eines Datensatzes modelliert werden.

Metamodell *GoogleAppengine*

Das Metamodell *GoogleAppengine* dient der Modellierung von YAML-Dateien für die Google App Engine, mit denen die Laufzeitumgebung der Google App Engine in Form des Webservers, des Datenbankmanagementsystems etc. konfiguriert werden kann[45]. Die Syntax der für die Webapplikation zentralen Konfigurationsdatei *app.yaml* (vgl. Listing 15) wird durch die Metamodellelemente des EPackage *app* im EPackage *config* repräsentiert (vgl. Abbildung 45).

Die EClass *ApplicationYaml* wird als Spezialisierung von *File* an das Dateisystemmodell angebunden, besitzt EAttributes für Namens- und Versionsinformationen der Web-

[45] `http://code.google.com/intl/en/appengine/docs/python/config/`

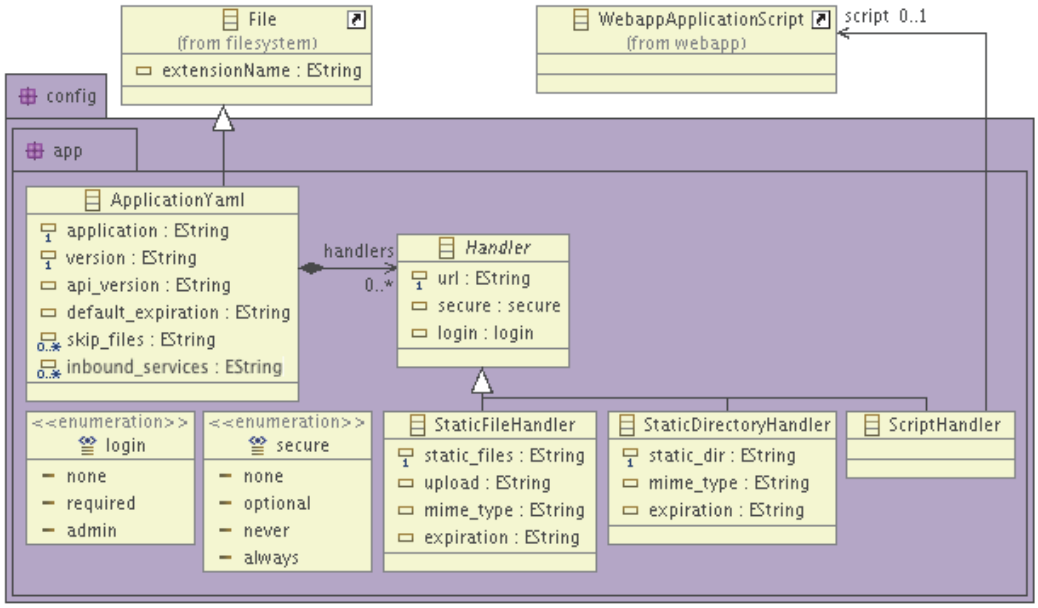

Abbildung 45: Ausschnitt aus dem Metamodell *GoogleAppengine*

```
1  application: waslpython-generated-app
2  version: 1
3  runtime: python
4  api_version: 1
5
6  handlers:
7  - url: /stylesheets
8    static_dir: stylesheets
9  - url: /crm/address/(.*)
10   script: app/address.py
```

Listing 15: Beispiel für eine *app.yaml*-Konfigurationsdatei

applikation und referenziert HTTP-Request-Handler über die EReference *handlers* zu der EClass *Handler*. Die *Handler*-Typen *StaticFileHandler*, *StaticDirectoryHandler* und *ScriptHandler* dienen der Zuordnung von URLs oder URL-Mustern zu Dateien, Verzeichnissen oder *WebappApplicationScripts* und spezifizieren somit das Routing von HTTP-Requests in der Webapplikation. Zudem können mit den EAttributes *secure* und *login* Anforderungen an die Verschlüsselung des HTTP-Datenverkehrs und die Authentifizierung modelliert werden.

Metamodell *WaslPython*

Das Metamodell *WaslPython* enthält die EClass *WaslPythonModel*, mit dem die Containment-Hierarchie eines *WASL Python*-Modells typisiert wird (vgl. Abbildung 46). Neben dem Modell des Dateisystems werden Instanzen der EClass *FrameworkClass* gesondert in

die Containment-Hierarchie einbezogen, da sie weder in das Dateisystem integriert, noch in einem anderen Modellelement enthalten sind.

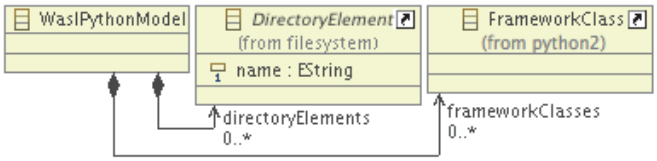

Abbildung 46: Metamodell *WaslPython*

4.6.3.5 Metamodelle für PHP

Metamodell *PHP5*

Das Metamodell *PHP5* bildet Sprachelemente der Skriptsprache PHP in der Version 5 für objektorientierte und prozedurale Programmierung ab (vgl. Abbildung 47). PHP-Skripte werden als Instanzen des Metamodellelements *PHPScript* modelliert, das eine Spezialisierung von *File* ist. Korrespondierend zu den prozeduralen oder objektorientierten Arten von PHP-Dateien kann ein *PHPScript* beliebig viele Instanzen der EClasses *Class*, *Interface*, *Namespace* und *Function* enthalten, wobei *Namespaces* ihrerseits *NamespaceElements* in Form der Spezialisierungen *Class*, *Interface* und *Function* enthalten können.

Ähnlich zu dem Metamodell *Java6* wird zwischen eigenentwickelten Klassen sowie Interfaces, und solchen aus Frameworks unterschieden, sämtliche Arten aber in ein gemeinsames Typsystem integriert. So werden einerseits *Class* und *Interface* als *Classifier*, sowie *FrameworkClass* und *FrameworkInterface* als *FrameworkClassifier* generalisiert. Andererseits werden *Class* und *FrameworkClass* als *AbstractClass*, sowie *Interface* und *FrameworkInterface* als *AbstractInterface* generalisiert. Durch die EReference *superClass* zwischen *Class* und *AbstractClass* wird die Einfachvererbung von Klassen sowohl von eigenentwickelten als auch von solchen aus Frameworks ermöglicht. Analog kann ein *Interface* über die EReference *superInterfaces* spezialisiert werden und über die EReference *implementedInterfaces* durch Instanzen von *Class* implementiert werden.

Classifier werden durch Instanzen von *Variable* und *Method* verfeinert. Die Unterscheidung zwischen den EClasses *Method* und *Function* ist nötig, da Erstere im Kontext von Klassen und Letztere im Kontext von PHP-Skripten existieren. Entsprechend besitzt auch nur die EClass *Method* das EAttribute *isStatic* für die Kennzeichnung von Klassenmethoden und das EAttribute *visibility* für die Sichtbarkeit. Beide werden durch die EClass *AbstractFunction* generalisiert, welche die EClass *Parameter* referenziert. Den EClasses *Parameter* und *Variable* kann zudem über die EClass *MultiplicityElement* eine Ein- oder Mehrwertigkeit zugeordnet werden.

Die meisten der EClasses sind Spezialisierungen der EClass *CommentableElement*, die mit Kommentaren in Form von Instanzen der EClasses *BlockComment* für Blockkommentare und *InlineComment* für Zeilenkommentare angereichert werden kann. Dies ist insbesondere relevant für die Modellierung von Annotationen für Frameworks wie z. B. Doctrine 2.

Da PHP 5 eine Programmiersprache mit dynamischer Typisierung ist, werden Variablen, Parameter und Methoden bzw. Funktionen im Modell nicht explizit typisiert. Im Vergleich zu dem Metamodell *Java6* entfällt somit auch die Notwendigkeit für eine EClass

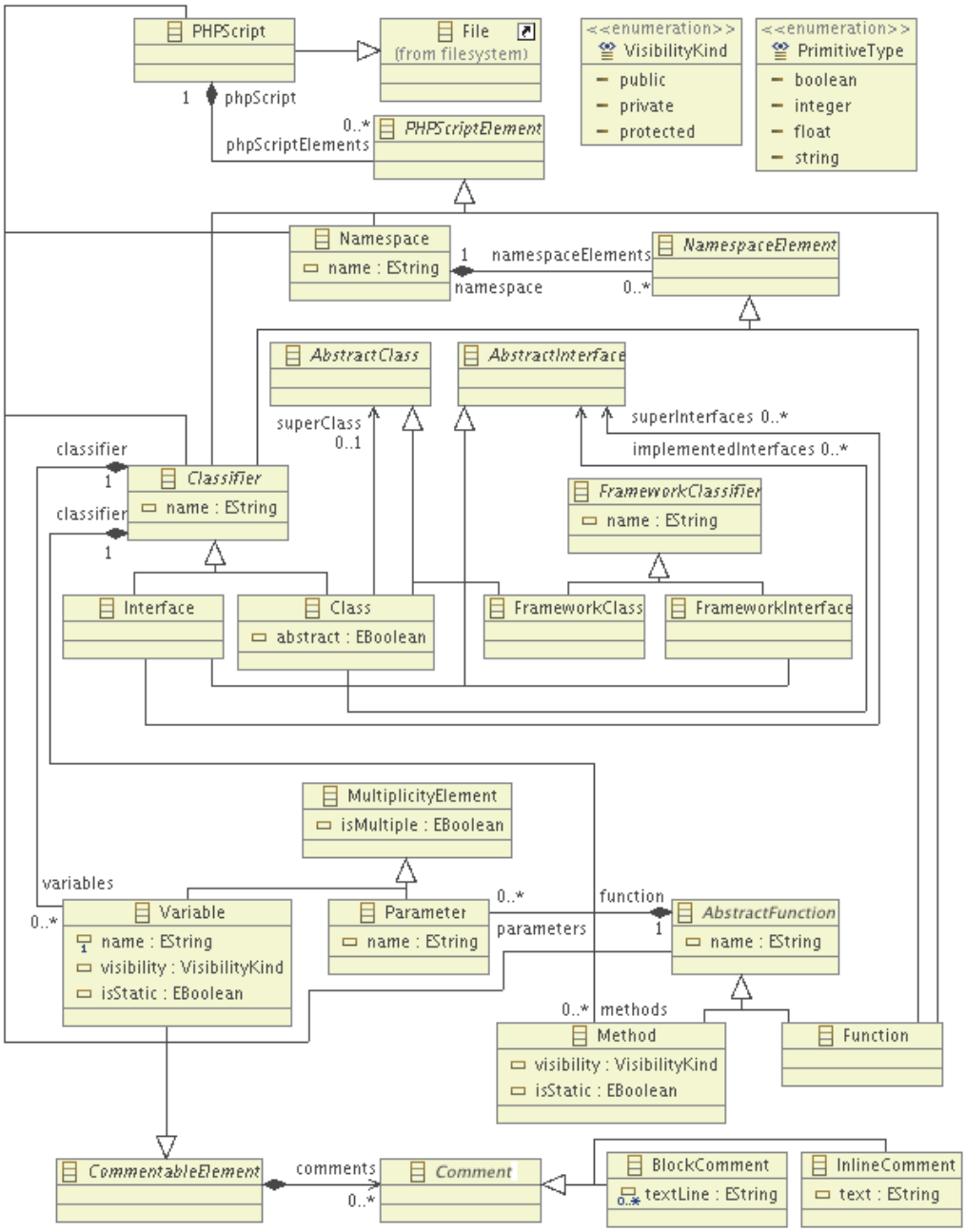

Abbildung 47: Metamodell *PHP5*

AbstractClassifier als Ziel für Typ-EReferences. Die EClass *MultiplicityElement* wird entsprechend nicht für eine Ermittlung des Typs von Variablen und Parametern ausgewertet, findet aber bei der Generierung von Quelltext für deren Initialisierung zur Programmlaufzeit Verwendung. Aufgrund der dynamischen Typisierung werden auch die Basisdatentypen aus der EEnum *PrimitiveType* nicht als Typen referenziert, stehen aber in dieser für Erweiterungen des Metamodells durch weitere Metamodelle bereit.

Metamodell *PHPDesignPatterns*

Das Metamodell *PHPDesignPatterns* ist basierend auf dem Metamodell *PHP5* konzeptionell der Ankerpunkt für Entwurfsmuster und umfasst in der grundlegenden Fassung als triviales Muster im weitesten Sinne Metamodellelemente zur Repräsentation von Accessor-Methoden. Diese sind im EPackage *accessor* enthalten und umfassen die abstrakte EClass *Accessor* als Spezialisierung von *Method*, die durch die EClasses *Getter* und *Setter* spezialisiert wird (vgl. Abbildung 48). Beide referenzieren obligatorisch die auszugebende bzw. zu modifizierende *Variable*. Restriktionen bei der Instanziierung sind, dass die Klasse der *Variable* mit derjenigen der *Accessor*-Methode identisch zu sein hat und die *Setter*-Methode einen *Parameter* besitzen muss.

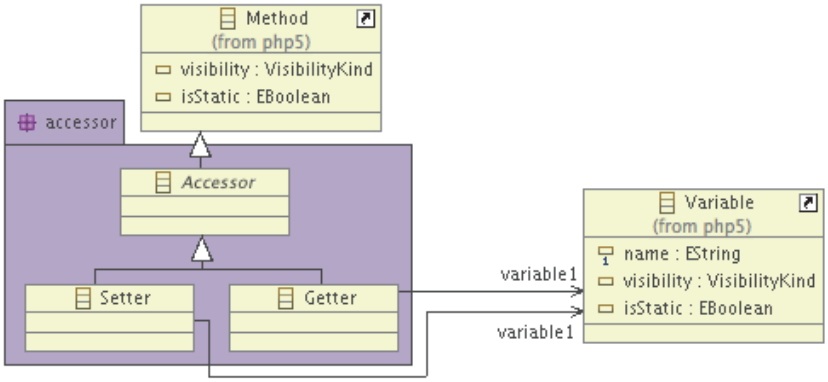

Abbildung 48: Metamodell *PHPDesignPatterns*

Metamodell *Doctrine2*

Das Metamodell *Doctrine2* adressiert in Verbindung mit dem nachfolgend beschriebenen Metamodell *Doctrine2Mapping_XSD* die PHP-Bibliotheken des ORM-Frameworks Doctrine 2[46]. Das Metamodell ist entsprechend zu dessen Libraries in die EPackages *common*, *dbal* und *orm* unterteilt, von denen das EPackage *orm* mit EClasses ausdefiniert ist

[46] `http://www.doctrine-project.org/`

und die Ausdrucksmittel der Doctrine Library *Object Relational Mapper* repräsentiert (vgl. Abbildung 49).

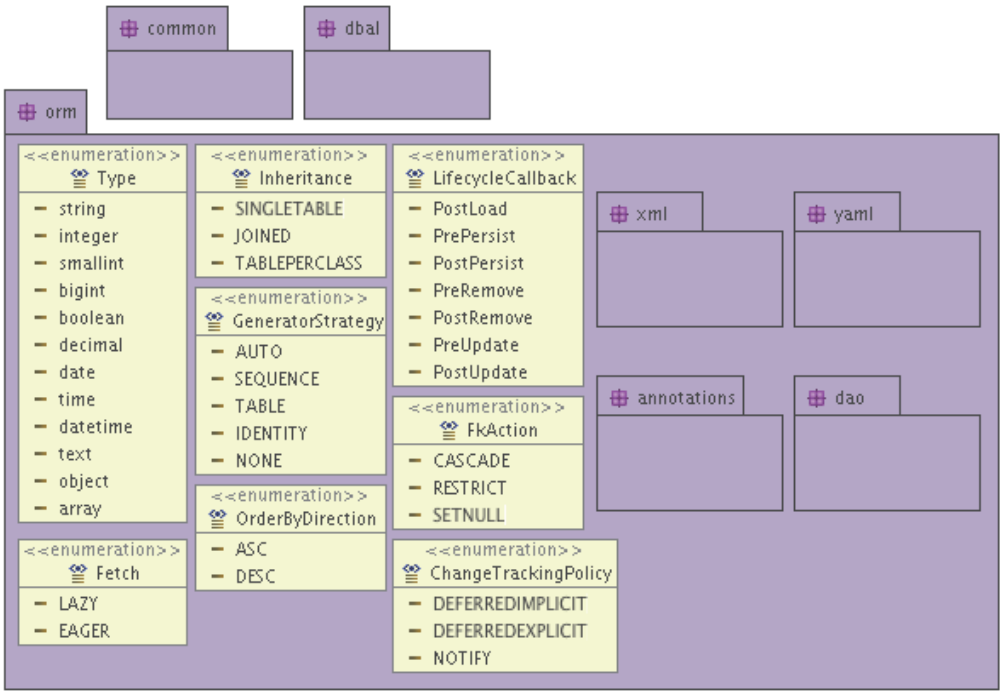

Abbildung 49: Paketstruktur des Metamodells *Doctrine2*

In dem EPackage *orm* befinden sich das EPackage *dao* zur Modellierung von Datenzugriffsklassen sowie die EPackages *annotations*, *xml* und *yaml* zur Definition von Mappings zwischen Entitätsklassen und Datenbanktabellen. In Doctrine 2 wird die Abbildung von Entitätsklassen auf Tabellen durch Annotationen an PHP-Klassen, XML-Dokumenten, YAML-Dateien oder PHP-Anweisungen definiert, von denen die ersten drei Alternativen im Metamodell durch gleichnamige EPackages repräsentiert werden. Da die Varianten eine ähnliche Semantik aufweisen und sich lediglich bzgl. der Syntax unterscheiden, wird im Folgenden auf die EPackages *xml* zur Repräsentation von XML-Mappings und *annotations* für Annotationen eingegangen. Das EPackage *orm* enthält zudem mehrere Enumerationen, wie z. B. *Type* zur Auflistung von Basisdatentypen und *Fetch* zur Repräsentation von Ladestrategien, die von den EClasses der Sub-EPackages referenziert werden.

Das EPackage *annotations* umfasst Metamodellelemente für Annotationen, die im Quelltext als Zeichenketten in PHP-Blockkommentaren realisiert werden. Dazu wird die EClass *BlockComment* aus dem Metamodell *PHP5* durch die EClass *DocBlock* spezialisiert, die über die EReference *annotations* Instanzen der abstrakten EClass *Annotation* sowie deren Spezialisierungen referenziert.

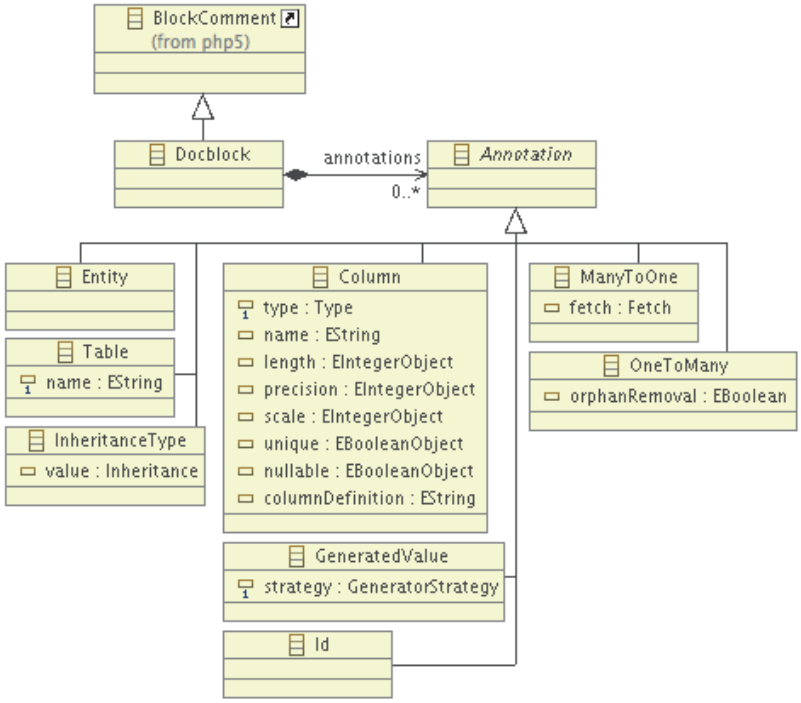

Abbildung 50: Ausschnitt aus dem EPackage *annotations* aus dem Metamodell *Doctrine2*

Diese Spezialisierungen sind EClasses für die verschiedenen Arten von Annotationen wie z. B. *Entity* zur Kennzeichnung von Entitätsklassen i.V.m *Table* zur Zuordnung von Datenbanktabellen zu Entitätsklassen und *InheritanceType* zur Festlegung der Abbildungsstrategie für Vererbungshierarchien (vgl. Listing 16). Variablen werden mit der Annotation *Column* typisiert und Tabellenspalten zugeordnet, durch *Id* i.V.m. *GeneratedValue* als Schlüsselattribut mit automatisch erzeugtem Wert deklariert und durch *ManyToOne* oder *OneToMany* als Beziehungsattribut zwischen zwei Entitätsklassen mit entsprechender Kardinalität festgelegt. Eine Beschreibung der Semantik der restlichen Metamodellelemente des EPackage findet sich in der Beschreibung der semantisch verwandten Elemente aus dem Metamodell *Doctrine2Mapping_XSD* (vgl. Kapitel 4.6.3.5).

Das EPackage *dao* umfasst Metamodellelemente zur Modellierung von Datenzugriffsklassen für Doctrine 2. Eine Datenzugriffsklasse wird mittels der EClass *DataAccessObject* modelliert, die eine Spezialisierung von *Class* ist. Deren Logik zum Datenabruf wird durch die EClasses *Fetch* und *FetchAll* repräsentiert, welche die EClass *DAOMethod* spezialisieren. Die beiden Methodentypen referenzieren jeweils durch die EReference *managedEntityClass* diejenige *EntityClass*, unter deren Instanzen ein Datensatz gesucht werden soll (*Fetch*) oder deren Datensätze insgesamt zurückgegeben werden sollen (*FetchAll*). Die *DataAccessObjects* dienen ausschließlich der Abfrage von Datensätzen und nicht der Modifi-

```
/**
 * @Entity
 * @Table(name = "person")
 * @InheritanceType("SINGLE_TABLE")
 */
class Person
{
  /**
  * @Column(type="integer", name="id")
  * @Id
  * @GeneratedValue(strategy="AUTO")
  */
  private $id;
}
```

Listing 16: Beispiel für Doctrine-Annotationen an einer PHP-Klasse

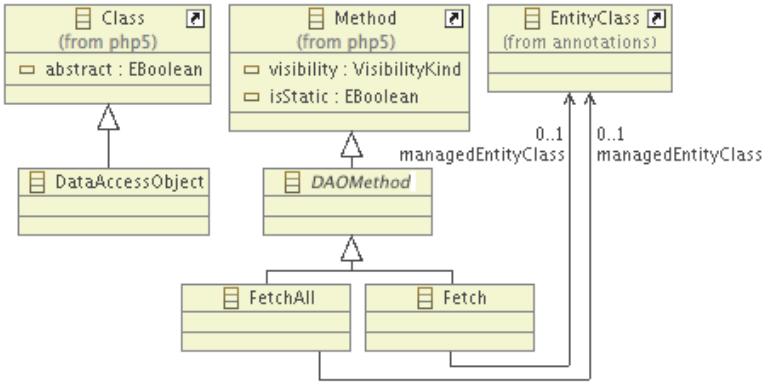

Abbildung 51: EPackage *dao* aus dem Metamodell *Doctrine2*

kation, da Abfragen beliebig komplex definiert werden können, das Speichern und Löschen von Datensätzen mit Doctrine 2 dagegen durch triviale Aufrufe auf dem EntityManager von Doctrine 2 initiiert wird. Das Metamodell enthält somit keine Methodentypen zur Modifikation von Datensätzen, bietet aber mit der EClass *DataAccessObject* einen Ankerpunkt zur Injektion von Typen für beliebig komplexe Abfragemethoden.

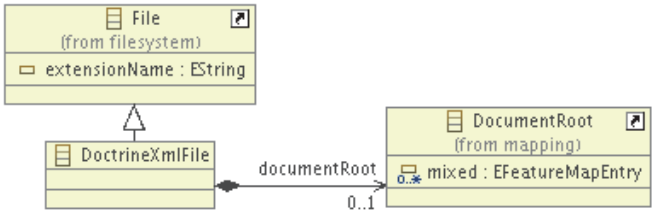

Abbildung 52: EPackage *xml* aus dem Metamodell *Doctrine2*

Das EPackage *xml* enthält selbst keine Metamodellelemente zur Definition von Mappings, sondern verweist über das Metamodellelement *DoctrineXmlFile* als Spezialisierung

von *File* auf die EClass *DocumentRoot* aus dem Metamodell *Doctrine2Mapping_XSD*, das der Modellierung von XML-Dokumenten dient und auf diese Weise in das Modell des Dateisystems eingebunden wird (vgl. Abbildung 52).

Metamodell *Doctrine2Mapping_XSD*

Das Metamodell *Doctrine2Mapping_XSD* repräsentiert vollständig die Konfigurationsparameter von Doctrine 2 zur Abbildung von Entitätsklassen auf Datenbanktabellen mittels XML-Konfigurationsdokumenten. Es ist automatisiert aus der XML-Schemadefinition *doctrine-mapping.xsd*[47] generiert und spiegelt somit deren Syntax und Sematik wider, die im Detail in der Doctrine-Dokumentation[48] beschrieben ist und auf deren wesentliche Konzepte im Folgenden eingegangen wird.

Das Metamodell enthält als Wurzelelement der Containment-Hierarchie die EClass *DocumentRoot*, welche auf die EClass *DoctrineMappingType* als Wurzelelement des XML-Dokuments verweist (vgl. Abbildung 53). Letztere referenziert die EClass *Entity*, durch deren Instanzen ein Doctrine-Mapping primär beschrieben wird. Jede Entitätsdefinition verweist optional auf eine Datenbanktabelle (EAttribute *table*) als Alternative zu einer Zuordnung per Namenskonvention und enthält Informationen zu Tabellenspalten bzw. Feldern (*Field*), zu Schlüsselattributen (*Id*) i.V.m. mit Strategien für die automatisierte Erzeugung von Schlüsselwerten (*Generator*, *SequenceGenerator*), zur Einzigartigkeit von Attributswerten (*UniqueConstraints*, *UniqueConstraint*), zu Indexen (*Indexes*, *Index*) und zu Beziehungen zwischen Entitätsdefinitionen (*OneToOne*, *ManyToOne*, *OneToMany*, *ManyToMany*). Mit der EClass *JoinColumn* kann optional ein Fremdschlüssel explizit definiert werden, welcher bei der Verwendung von *OneToOne* und *ManyToOne* in der XML-Schemadefinition einer Objektbeziehung zugrunde liegt. Im Fall von *ManyToMany* ist zudem mit der EClass *JoinTable* eine Kreuztabelle i.V.m. einem Verweis auf die Fremdschlüsselattribute über die EClass *JoinColumns* zu spezifizieren. Die Sortierung der Entitäten bei mehrwertigen Objektbeziehungen von *OneToMany* und *ManyToMany* wird mittels der EClasses *OrderBy* und *OrderByField* im Modell spezifiziert.

Metamodell *Web*

Das Metamodell *Web* adressiert exemplarisch PHP-Skripte für die Präsentationsschicht einer Webapplikation, die ohne Bezug zu einem bestimmten Präsentations-Framework implementiert sind. Dazu enthält das Metamodell als Spezialisierungen der EClass *PHPScript* EClasses zur Repräsentation der verschiedenen Arten von PHP-Skripten (vgl. Abbildung

[47] `http://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd`
[48] `http://www.doctrine-project.org/docs/orm/2.1/en/reference/xml-mapping.html`

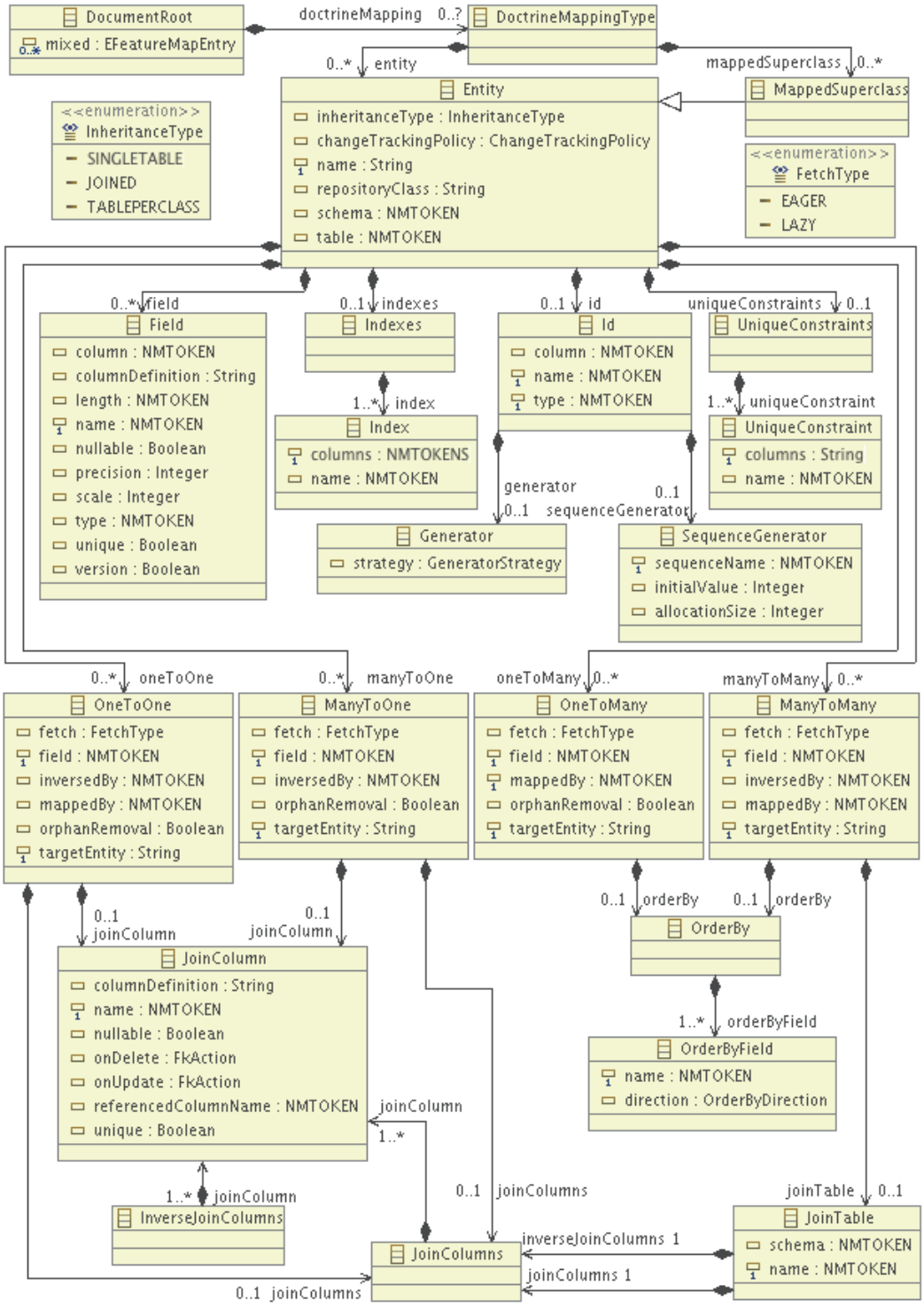

Abbildung 53: Ausschnitt aus dem Metamodell *Doctrine2Mapping_XSD*

54). Von diesen typisiert die EClass *PHPWebPageScript* PHP-Webseitenskripte, die im Modell über EReferences zu den EClasses *Block*, *Flow* und *Inline* aus dem Metamodell *XHTML1-Strict_XSD_Adapted* mit XHTML-Inhalten und über die EReference zu *PHPComponent* mit vordefinierten aggregierten PHP-Funktionsbausteinen befüllt werden können. Analog zu der EClass *Template* aus dem Metamodell *Django* (vgl. Kapitel 4.6.3.4) werden auch bei der EClass *PHPWebPageScript* diese Elemente in dem EAttribute *group* geordnet in einer FeatureMap gesammelt. Spezielle Arten von Webseitenskripten für seitenübergreifende Menü- und Rahmentemplates werden als Spezialisierung von *PHPWebPageScript* durch die EClasses *PHPMenuIncludeScript*, *PHPHeaderScript* und *PHPFooterScript* typisiert.

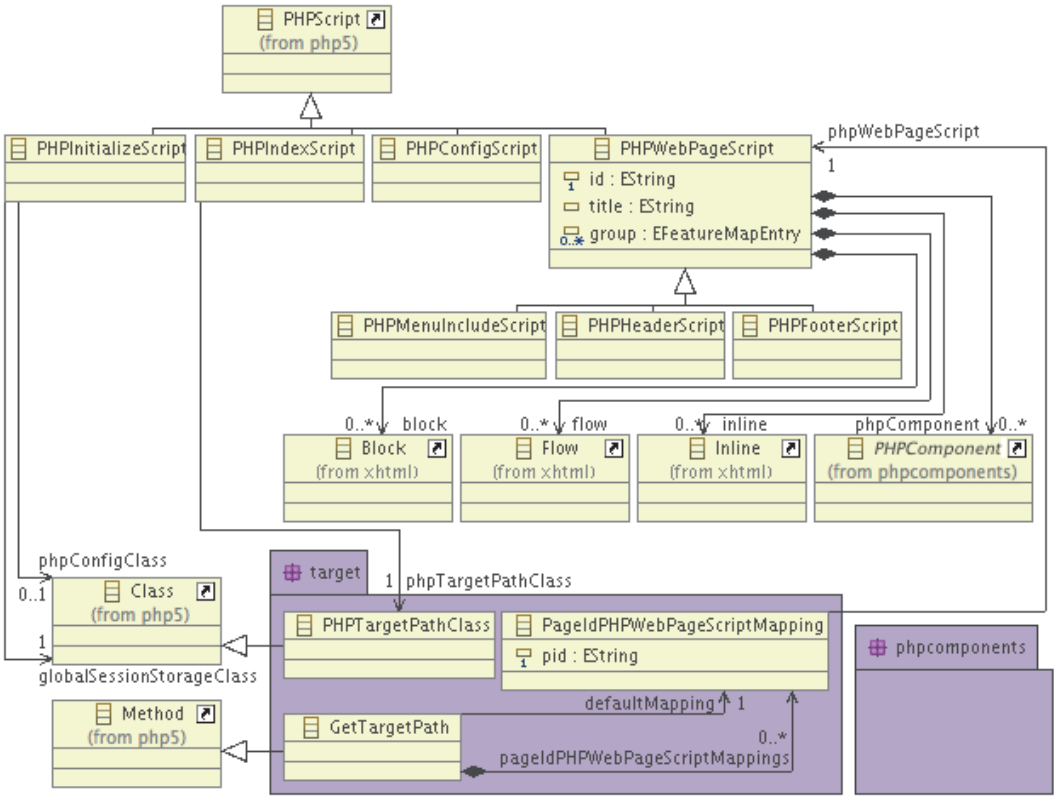

Abbildung 54: Metamodell *Web*

Im Sinne des MVC-Architekturmusters sind *PHPWebPageScripts* Views, deren gemeinsamer Controller mit den EClasses *PHPInitializeScript* und *PHPIndexScript* modelliert wird. Jeder HTTP-Request wird an das *PHPIndexScript* gerichtet, das mit dem *PHPInitializeScript* auf Basis einer Konfiguration in einem *PHPConfigScript* den Speicher präpariert und anschließend das passende *PHPWebPageScript* ausgibt. Das passende *PHPWebPageScript* wird abhängig von dem Wert eines HTTP-Request-Parameters mit dem Namen *pid*

ermittelt. Dazu ruft das *PHPIndexScript* auf einer Klasse vom Typ *PHPTargetPathClass* die Methode *GetTargetPath* auf, welche mit der EClass *PageIdPHPWebPageScriptMapping* eindeutige Zuordnungen von *pids* zu *PHPWebPageScripts* enthält und das passende *PHPWebPageScript* für eine *pid* zurückgibt. Bei einem HTTP-Request ohne Ausprägung des *pid*-Parameters wird dasjenige *PHPWebPageScript* ausgegeben, auf dessen Mapping durch die EReference *defaultMapping* verwiesen wird.

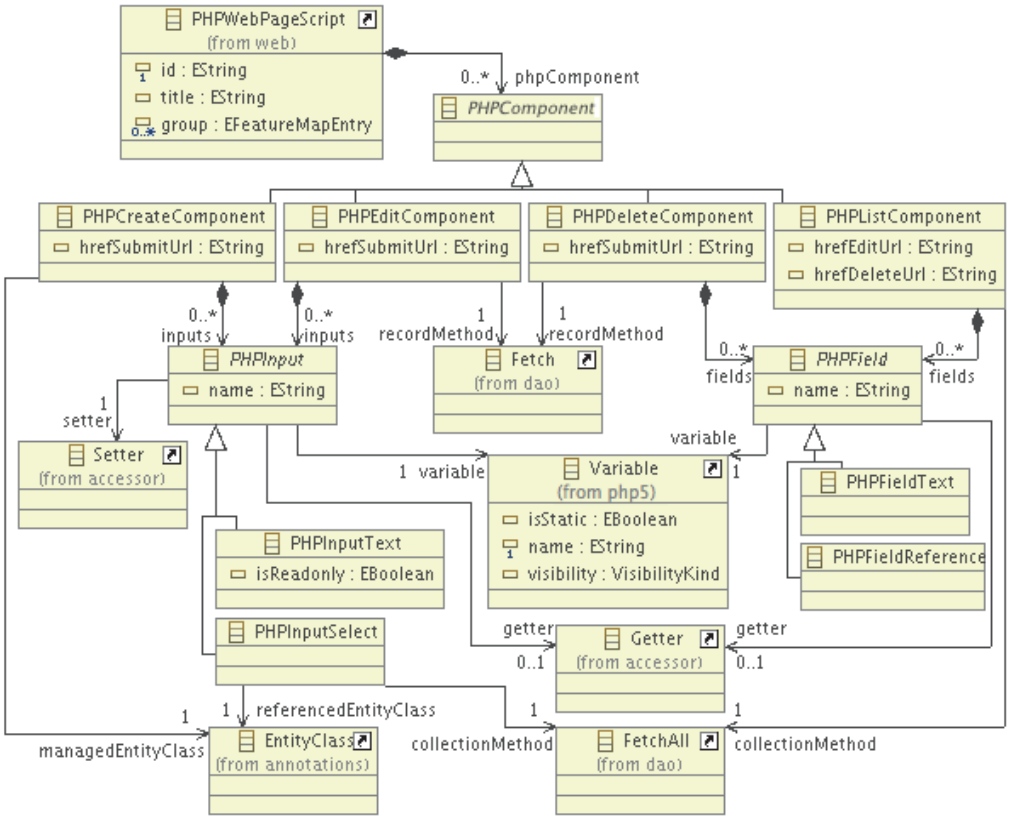

Abbildung 55: EPackage *phpcomponents* aus dem Metamodell *Web*

Für die Modellierung von CRUD-Funktionalität sind im EPackage *phpcomponents* die PHP-Funktionsbausteine *PHPCreateComponent*, *PHPEditComponent*, *PHPDeleteComponent* und *PHPListComponent* als Spezialisierungen von *PHPComponent* vordefiniert (vgl. Abbildung 55). Die EClass *PHPCreateComponent* repräsentiert PHP-Funktionalität für ein Formular, dessen Felder als *PHPInputs* modelliert werden und als dessen Resultat das Objekt einer *EntityClass* erzeugt wird. Auf die *EntityClass* verweist die EReference *managedEntityClass*, sodass ein entsprechendes Objekt der Klasse erzeugt und mit den Eingabewerten ausgeprägt werden kann. Analog dient die EClass *PHPEditComponent* der Repräsentation von PHP-Funktionalität für ein Formular zur Modifikation des Objekts einer *EntityClass*. Im Unterschied zu der *PHPCreateComponent* wird im Modell nicht

auf eine verwaltete *EntityClass* verwiesen, sondern mit der EReference *recordMethod* eine DAO-*Fetch*-Methode referenziert, die das Objekt der *EntityClass* zurückgibt. Die EClass *PHPDeleteComponent* repräsentiert PHP-Funktionalität zur Darstellung der Daten des Objekts einer *EntityClass* i.V.m. dessen Löschung. Für den Zugriff auf das Objekt wird ebenfalls mit der EReference *recordMethod* eine DAO-Fetch-Methode referenziert. Eine Auswahl an Attributen des Objekts zur Ausgabe wird mit Instanzen von *PHPField* vorgenommen, die der Komponente zugeordnet werden. Abschließend repräsentiert die EClass *PHPListComponent* PHP-Funktionalität zur Auflistung von *EntityClass*-Objekten, für die ebenfalls eine Auswahl der auszugebenden Attribute über eine Liste an *PHPFields* vorgenommen werden kann. Die Objekte werden über die DAO-*FetchAll*-Methode unter der EReference *collectionMethod* abgerufen.

Jeder *PHPInput* und jedes *PHPField* referenziert zur Benennung die korrespondierende *Variable* und zum Abruf des Attributwerts den *Getter* für die Variable. Ein *PHPInput* verweist zudem auf den entsprechenden *Setter* zur Modifikation des Werts. Bei *PHPInputs* wird als Spezialisierungen zwischen *PHPInputText* für textuelle Eingabefelder und *PHPInputSelect* für Auswahlmenüs unterschieden. Letztere referenzieren ihrerseits die auswählbare *EntityClass*, deren Objekte mit der DAO-*FetchAll*-Methode unter der EReference *collectionMethod* abgerufen werden. Analog wird bei *PHPFields* zwischen *PHPFieldText* für textuelle und *PHPFieldReference* für referenzielle Felder unterschieden.

Der Navigationsgraph manifestiert sich mit einem geringen Abstraktionsgrad einerseits in statischen Hyperlinks, die als XHTML-Elemente in den *PHPWebPageScripts* enthalten sind, und als parametrisierte Hyperlinks, die dynamisch mit der *pid* und dem Schlüssel des ausgewählten Objekts bzw. Datensatzes versehen werden und in den verschiedenen *PHPComponents* als die EAttributes *hrefSubmitUrl*, *hrefEditUrl* und *hrefDeleteUrl* annotiert sind.

Metamodell *WaslPHP*

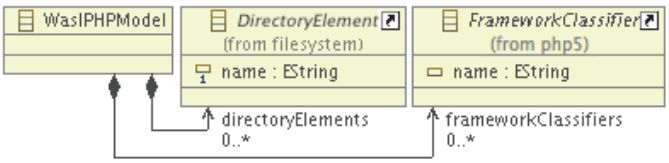

Abbildung 56: Metamodell *WaslPHP*

Das Metamodell *WaslPHP* enthält die EClass *WaslPHPModel*, welche das Wurzelelement der Containment-Hierarchie eines *WASL PHP*-Modells typisiert (vgl. Abbildung 56).

Es referenziert mit der Containment-EReference *directoryElements* das Dateisystemodell und mit der Containment-EReference *frameworkClassifiers FrameworkClassifier*, die sonst generell in keinem anderen Modellelement enthalten sind.

4.7 Modell-zu-Modell-Transformationsdefinitionen

Im Folgenden wird auf die M2M-Transformationsdefinitionen des WASL-Generatorframeworks eingegangen. Sie sind in unidirektionaler Form mit der Transformationssprache QVTO implementiert und nehmen Bezug auf die Elemente der vorab beschriebenen Metamodelle.

4.7.1 Transformationsdefinition *WaslData2WaslGeneric*

Die Transformationsdefinition *WaslData2WaslGeneric* bildet die Elemente des Metamodells von *WASL Data* auf solche des Metamodells von *WASL Generic* ab und ermöglicht somit die Transformation eines *WASL Data*-Modells in ein *WASL Generic*-Modell. Da Letzteres über das Datenmodell hinaus umfangreichere funktionale Aspekte der Webapplikation abdeckt, wird es auf Basis der Inhalte des *WASL Data*-Modells automatisiert mit vordefinierten Mustern angereichert.

Im Rahmen der QVTO-Transformation wird das Metamodell von *WASL Data* unter dem Modelltyp *DATA* und das Metamodell von *WASL Generic* unter dem Modelltyp *GENERIC* referenziert (vgl. Listing 17). Die Transformation nimmt als eingehenden Parameter ein DATA-Modell entgegen und gibt ein GENERIC-Modell zurück, indem die korrespondierende Main-Operation durch Aufruf der Mapping-Operation *WaslDataModel::toGenericModel* für jedes Wurzel-Modellelement vom Typ *WaslDataModel* ein Modellelement vom Typ *WaslGenericModel* erzeugt. Für die nachgelagerten Mapping-Operationen (Mappings) wird zudem die Bibliothek *WaslData* importiert, in der Queries für komplexere Abfragen auf dem gleichnamigen Metamodell vordefiniert sind.

```
1 import WaslData;
2 modeltype DATA uses 'wasldata';
3 modeltype GENERIC uses 'waslgeneric';
4
5 transformation WaslData2WaslAbstract(in wasldata : DATA, out GENERIC);
6
7 main() {
8   wasldata.rootObjects()[DATA::WaslDataModel]->map toGenericModel();
9 }
```

Listing 17: Transformation *WaslData2WaslGeneric*

Das Mapping *WaslDataModel::toGenericModel* nimmt ein *WaslDataModel* entgegen und gibt ein *WaslGenericModel* zurück, in dessen Containment-Hierarchie die EReferences *dataModel*, *navigationModel* und *viewModel* anhand der Mappings *DataModel::toDataModel*, *DataModel::toNavigationModel* und *DataModel::toViewModel* mit Untermodellen belegt werden (vgl. Listing 18).

Die Mappings zur Abbildung eines *WASL Data*-Datenmodells auf ein *WASL Generic*-Datenmodell sind trivial, da die beiden Metamodelle isomorph sind. Das Mapping

```
mapping DATA::WaslDataModel::toGenericModel() : GENERIC::WaslGenericModel{
  dataModel := self.dataModel.map toDataModel();
  navigationModel := self.dataModel.map toNavigationModel(self);
  viewModel := self.dataModel.map toViewModel(self);
}
```

Listing 18: Mapping *WaslDataModel::toGenericModel*

DataModel::toDataModel bildet ein *DataModel* von *WASL Data* auf ein *DataModel* von *WASL Generic* ab und transferiert mit dem Mapping *Package::toPackage* das darin enthaltene Wurzelpackage (vgl. Listing 19). Dessen Inhalte werden ebenfalls strukturgleich nachgebildet, beispielsweise im Fall von *Entities* durch Aufruf des Mappings *Entity::toEntity*.

```
mapping DATA::data::DataModel::toDataModel() : GENERIC::data::DataModel{
  ownedPackage := self.ownedPackage.map toPackage();
}

mapping DATA::data::Package::toPackage() : GENERIC::data::Package{
  name := self.name;
  ownedPackages := self.ownedPackages.map toPackage();
  ownedEntities := self.ownedEntities.map toEntity();
  ownedAssociations := self.ownedAssociations.map toAssociation();
}

mapping DATA::data::Entity::toEntity() : GENERIC::data::Entity{
  name := self.name;
  generalization := self.generalization.map toEntity();
  ...
}
```

Listing 19: Auswahl aus den Mappings für die Transformation des Datenmodells

Das *NavigationModel* wird auf Basis des Datenmodells mit dem Mapping *DataModel::toNavigationModel* konstruiert (vgl. Listing 20). Dem *NavigationModel* wird unter der EReference *rootNodeGroup* eine *NodeGroup* hinzugefügt. Diese erhält mit dem Mapping *Entity::toNodeGroup* für jedes *Entity* aus der *Package*-Hierarchie jeweils eine *NodeGroup*, in der *Nodes* für die verschiedenen CRUD-Operationen zur Bearbeitung des *Entity* enthalten sind. Dem Navigationsmodell wird zudem ein *MenuFolder* hinzugefügt, der für jedes *Entity* mit dem Mapping *Entity::toMenuEntry* ein *MenuEntry* erhält. Als *indexNode* wird als Ausgangspunkt für eine nachträgliche manuelle Modifikation ein *Node* festgelegt, der mit dem Mapping *Entity::toListNode* aus dem initial angelegten *Entity* des Datenmodells abgeleitet wird und somit unter der EReference *logicTuple* ein *ListTuple* aufweist.

Das Mapping *Entity::toNodeGroup*, das für ein *Entity* eine *NodeGroup* mit *Nodes* für CRUD-Funktionalität erstellt, ruft dazu die Mappings *Entity::toListNode*, *Entity::toCreateNode*, *Entity::toEditNode*, *Entity::toDeleteNode* und *Entity::toDataSetNode* auf, und fügt nach jedem Aufruf den jeweils resultierenden *Node* zu der EReference *ownedNodeElements* hinzu (vgl. Listing 21). Die Kanten des Navigationsgraphen werden gebildet, indem *Links* unter der EReference *ownedLinks* zwischen den *Nodes*, z. B. zum Wechsel von einem Knoten mit List-Funktionalität zu einem Knoten mit Create-Funktionalität,

```
1  mapping DATA::data::DataModel::toNavigationModel(in model : DATA::WaslDataModel) : GENERIC::
           navigation::NavigationModel{
2    rootNodeGroup := object GENERIC::navigation::NodeGroup {
3      name := model.dataModel.ownedPackage.name;
4      ownedNodeElements := self.ownedPackage.unfoldPackageToEntities().map toNodeGroup(model)
                ->asOrderedSet();
5    };
6    rootMenuFolder := object GENERIC::navigation::MenuFolder {
7      name := model.dataModel.ownedPackage.name;
8      ownedMenuElements := self.ownedPackage.unfoldPackageToEntities().map toMenuEntry(model)
                ->asOrderedSet();
9    };
10   indexNode := self.ownedPackage.unfoldPackageToEntities()->first().map toListNode(model);
11 }
```

Listing 20: Mapping *DataModel::toNavigationModel*

```
1  mapping DATA::data::Entity::toNodeGroup(in model : DATA::WaslDataModel) : GENERIC::
           navigation::NodeGroup{
2    name := self.name;
3    ownedNodeElements := self.map toListNode(model);
4    ownedNodeElements += self.map toCreateNode(model);
5    ownedNodeElements += self.map toEditNode(model);
6    ownedNodeElements += self.map toDeleteNode(model);
7    ownedNodeElements += self.map toDataSetNode(model);
8    ownedLinks := object GENERIC::navigation::Link {
9      name := 'l_list_create';
10     source := self.map toListNode(model);
11     destination := self.map toCreateNode(model);
12   };
13   ...
14 }
```

Listing 21: Mapping *Entity::toNodeGroup*

gespeichert werden.

Ein Knoten mit List-Funktionalität wird mit dem Mapping *Entity::toListNode* aus einem *Entity* konstruiert, indem der Name des *Node* als Konkatenation des Strings „list“ mit dem Namen des *Entity* in der Pluralform festgelegt wird, und dem *Node* als *logicTuple* ein *ListTuple* hinzugefügt wird (vgl. Listing 22). Dessen EReference *view* referenziert zur graphischen Repräsentation der Logik eine *ListView*, die durch Aufruf des Mappings *Entity::toListView* erzeugt wird, und mit der EReference *entity* als Resultat des Mappings *Entity::toEntity* auf die transferierte Version des zugrunde liegenden *Entity* verweist.

```
1  mapping DATA::data::Entity::toListNode(in model : DATA::WaslDataModel) : GENERIC::navigation
           ::Node{
2    name := 'list ' + self.name.getPlural();
3    logicTuple := object GENERIC::logic::logictupletypes::ListTuple{
4      view := self.map toListView(model);
5      entity := self.map toEntity();
6    };
7  }
```

Listing 22: Mapping *Entity::toListNode*

Analog werden mit den Mappings *Entity::toCreateNode*, *Entity::toEditNode*, *Entity::toDeleteNode* und *Entity::toDataSetNode Nodes* konstruiert, die ein *CreateTuple*, *EditTuple*, *DeleteTuple* bzw. *DataSetTuple* enthalten, welches ebenfalls auf die Kopie des *Entity*

verweist und eine korrespondierende *CreateView*, *EditView*, *DeleteView* bzw. *DataSetView* referenziert.

Das *ViewModel* wird auf Basis des Datenmodells mit dem Mapping *DataModel::toViewModel* konstruiert (vgl. Listing 23). Dem *ViewModel* wird eine *ViewGroup* hinzugefügt, welche für die *Entities* des Datenmodells jeweils eine *ViewGroup* mit *Views* für die Darstellung der CRUD-Funktionalität enthält.

```
1 mapping DATA::data::DataModel::toViewModel(in model : DATA::WaslDataModel) : GENERIC::
          presentation::ViewModel{
2   rootViewGroup := object GENERIC::presentation::ViewGroup{
3     name := model.dataModel.ownedPackage.name;
4     ownedViewElements := self.ownedPackage.unfoldPackageToEntities().map toViewGroup(model);
5   }
6 }
```

Listing 23: Mapping *DataModel::toViewModel*

Diese *ViewGroups* werden durch das Mapping *Entity::toViewGroup* erstellt, welches die Mappings *Entity::toListView*, *Entity::toCreateView*, *Entity::toEditView*, *Entity::toDeleteView* und *Entity::toDataSetView* zur Konstruktion der *Views* aus einem *Entity* aufruft (vgl. Listing 24). Da diese Mappings bereits vorab für das jeweilige *Entity* durch die Mappings *Entity::toListNode* (vgl. Listing 22), *Entity::toCreateNode*, *Entity::toEditNode*, *Entity::toDeleteNode* und *Entity::toDataSetNode* aufgerufen werden, werden bei dem Aufruf durch das Mapping *Entity::toViewGroup* die bereits existierenden Modellelemente in der EReference *ownedViewElements* lediglich re-referenziert und über die *ViewGroup* in die Containment-Hierarchie des Modells eingebunden.

```
1 mapping DATA::data::Entity::toViewGroup(in model : DATA::WaslDataModel) : GENERIC::
          presentation::ViewGroup{
2   name := self.name;
3   ownedViewElements := self.map toListView(model);
4   ownedViewElements += self.map toCreateView(model);
5   ownedViewElements += self.map toEditView(model);
6   ownedViewElements += self.map toDeleteView(model);
7   ownedViewElements += self.map toDataSetView(model);
8 }
```

Listing 24: Mapping *Entity::toViewGroup*

Entsprechend enthalten die *Views* auch bereits eine Referenz auf das jeweilige *LogicTuple*, sodass in den Mappings *Entity::toListView*, *Entity::toCreateView*, *Entity::toEditView*, *Entity::toDeleteView* und *Entity::toDataSetView* lediglich der Name, der Titel, die Beschreibung und die darzustellenden *Properties* des *Entity* festgelegt werden (vgl. Listing 25). Letztere werden durch die Aufrufe der Mappings *ValueProperty::toValueProperty* und *ReferenceProperty::toReferenceProperty* ebenfalls lediglich re-referenziert, da die Modellelemente der *Properties* bereits bei dem Transfer des zugrunde liegenden *Entity* durch das Mapping *Entity::toEntity* angelegt werden.

```
1 mapping DATA::data::Entity::toListView(in model : DATA::WaslDataModel) : GENERIC::
          presentation::viewtypes::ListView{
2   name := self.name.firstToLower() + '_list';
3   title := self.name;
4   description := 'On this page all records of the type ' + self.name + ' are listed.';
5   properties := self.getValueProperties().map toValueProperty();
6   properties += self.getReferenceProperties().map toReferenceProperty();
7 }
```

Listing 25: Mapping *Entity::toListView*

4.7.2 Transformationsdefinition *WaslGeneric2WaslJavaEE*

Die Transformationsdefinition *WaslGeneric2WaslJavaEE* bildet die Elemente des Metamodells von *WASL Generic* auf solche der Metamodelle von *WASL JavaEE* ab und ermöglicht somit die Transformation eines *WASL Generic*-Modells in ein *WASL JavaEE*-Modell. Dabei werden die plattformunabhängigen Muster des Eingangsmodells in plattformspezifische Muster der Realisierung übersetzt und somit der Abstraktionsgrad gesenkt.

Für die QVTO-Transformation wird das Metamodell von *WASL Generic* unter dem Modelltyp *GENERIC* und die Metamodelle von *WASL JavaEE* unter ihrem jeweiligen Namen in Majuskel-Schreibweise referenziert, um sie optisch in den Mappings von den Namen der EPackages und EClasses abzuheben (vgl. Listing 26). Die Transformation nimmt GENERIC-Modelle entgegen und gibt für jedes ein WASLJAVAEE-Modell zurück. Die Main-Operation der Transformation ruft dazu für jedes Wurzel-Modellelement vom Typ *WaslGenericModel* das Mapping *WaslGenericModel::toWaslJavaEEModel* auf, das ein Modellelement vom Typ *WaslJavaEEModel* zurückgibt. Für die nachfolgenden Mappings wird zudem die Bibliothek *WaslGeneric* mit vordefinierten Queries auf dem gleichnamigen Metamodell importiert.

```
1  import WaslGeneric;
2  modeltype GENERIC uses 'waslgeneric';
3  modeltype FILESYSTEM uses 'filesystem';
4  modeltype WASLJAVAEE uses 'wasljavaee';
5  modeltype JAVA6 uses 'java6';
6  ...
7
8  transformation WaslGeneric2WaslJavaEE(in waslgeneric : GENERIC, out WASLJAVAEE);
9
10 main() {
11   waslgeneric.rootObjects()[GENERIC::WaslGenericModel]->map toWaslJavaEEModel();
12 }
```

Listing 26: Transformation *WaslGeneric2WaslJavaEE*

Das Mapping *WaslGenericModel::toWaslJavaEEModel* konstruiert ein *WaslJavaEEModel* inkl. dessen Grobstruktur des Dateisystemmodells, welches auf der Wurzelebene das *Directory classes* für Javaklassen und das *Directory war* für das Web Application Archive (WAR) enthält (vgl. Listing 27). Dem Wurzel-*Directory classes* werden über das Mapping *DataModel::toModelDirectory* das *Directory model* für Entity-Klassen, über das Mapping *DataModel::toDaoDirectory* das *Directory dao* für Datenzugriffsklassen, sowie ohne Aufruf von Mappings die *Directories web* für JSF-Controller-Klassen und *util* für eine Hibernate-Hilfsklasse hinzugefügt. Das Wurzel-*Directory war* enthält konform zu der Servlet-Spezifikation [Ora11, S. 98 ff.] das Sub-*Directory WEB-INF* mit den Sub-Sub-*Directories lib* für Hilfsbibliotheken und *classes* für die kompilierten Javaklassen aus dem Wurzel-*Directory classes*. Das Dateisystemmodell wird über Aufrufe verschiedener Mappings mit

Instanzen von *File*-Typen der verschiedenen Metamodelle befüllt und integriert diese so in eine gemeinsame Sicht, die sich vom Dateisystem bis in die Klassen und deren Methoden erstreckt. Darüber hinaus fügt das Mapping *WaslGenericModel::toWaslJavaEEModel* der Containment-Hierarchie des *WaslJavaEEModel* Instanzen der EClass *FrameworkClass* sowie mit dem Mapping *MenuFolder::toMenuFolder* den Wurzelordner der Menüstruktur hinzu.

```
1  mapping GENERIC::WaslGenericModel::toWaslJavaEEModel() : WASLJAVAEE::WaslJavaEEModel{
2    directoryElements := object FILESYSTEM::Directory{
3      name := 'classes';
4      directoryElements := self.dataModel.map toModelDirectory(self);
5      directoryElements += self.dataModel.map toDaoDirectory(self);
6      directoryElements += object FILESYSTEM::Directory{
7        name := 'util';
8        directoryElements := self.dataModel.map toHibernateUtil(self);
9      };
10     directoryElements += object FILESYSTEM::Directory{
11       name := 'web';
12       directoryElements := self.dataModel.map toHibernateSessionRequestFilter(self);
13       directoryElements += self.navigationModel.map toControllerDirectory(self);
14     };
15   };
16   directoryElements += object FILESYSTEM::Directory{
17     name := 'war';
18     directoryElements := self.navigationModel.map toIndexRedirectJSP(self);
19     directoryElements += self.viewModel.map toFaceletsDirectory(self);
20     directoryElements += object FILESYSTEM::Directory{
21       name := 'WEB-INF';
22       directoryElements := object FILESYSTEM::Directory{
23         name := 'lib';
24       };
25       directoryElements += object FILESYSTEM::Directory{
26         name := 'classes';
27         directoryElements := self.dataModel.map toHibernateConfigFile(self);
28         directoryElements += self.map toLog4jConfigFile(self);
29       };
30       directoryElements += self.map toWebXMLFile(self);
31       directoryElements += self.navigationModel.map toFacesConfigFile(self);
32     };
33   };
34   frameworkClassifiers := self.map toFrameworkInterfaceSerializable(self);
35   frameworkClassifiers += GENERIC::data::PrimitiveDataTypes::String.map toFrameworkClass();
36   ...
37   rootMenuFolder := self.navigationModel.rootMenuFolder.map toMenuFolder(self);
38 }
```

Listing 27: Mapping *WaslGenericModel::toWaslJavaEEModel*

Das *DataModel* von *Wasl Generic* wird mit dem Mapping *DataModel::toModelDirectory* in das *Directory model* transformiert, welches für jedes *Package* aus dem Quellmodell durch das Mapping *Package::toDirectory* ein Sub-*Directory* erhält (vgl. Listing 28). Der Aufruf des Mappings *Package::toDirectory* wird im Mapping *Package::toDirectory* rekursiv fortgesetzt, sodass die *Package*-Hierarchie in eine *Directory*-Hierarchie übersetzt wird. Den *directoryElements* wird zudem für jedes *Entity* mit dem Mapping *Entity::toPersistentPojoFile* ein *ClassifierFile* hinzugefügt, welches das *PerstistentPojo* bzw. die Entitäts-Klasse enthält.

Das Mapping *Entity::toPersistentPojoFile* erstellt das *ClassifierFile* mit dem Namen des *Entity* (vgl. Listing 29). Es erhält ein *PersistentPojo*, welches aus dem *Entity* durch

```
1  mapping GENERIC::data::DataModel::toModelDirectory(in model : GENERIC::WaslGenericModel) :
           FILESYSTEM::Directory{
2    name := 'model';
3    directoryElements := self.ownedPackage.map toDirectory(model);
4  }
5
6  mapping GENERIC::data::Package::toDirectory(in model : GENERIC::WaslGenericModel) :
           FILESYSTEM::Directory{
7    name := self.name;
8    directoryElements := self.ownedPackages.map toDirectory(model);
9    directoryElements += self.ownedEntities.map toPersistentPojoFile(model);
10 }
```

Listing 28: Mappings *DataModel::toModelDirectory* und *Package::toDirectory*

das Mapping *Entity::toPersistentPojo* konstruiert wird (vgl. Listing 30). Dabei werden Spezifika von Entitäts-Klassen wie z. B. die Implementierung des Interface *Serializable*, die Variable *serialVersionUID* oder die Annotation *Entity* expliziert. Die Explikation im Modell erweitert zwar dessen Umfang und Komplexität, ist aber vorteilhaft, da die Gestalt des Generats aus dem Modell heraus kontrolliert werden kann und die Mappings typsicher bzgl. der Zielsprache implementiert werden können. Zudem können einem *PersistentPojo* bzw. generell einer *Class* beliebige andere Variablen, Methoden, Annotationen etc. hinzugefügt werden, sodass die Metamodelle für andere Arten von Klassen wiederverwendet werden können.

```
1  mapping GENERIC::data::Entity::toPersistentPojoFile(in model : GENERIC::WaslGenericModel) :
           JAVA6::ClassifierFile{
2    name := self.name;
3    extensionName := 'java';
4    classifier := self.map toPersistentPojo(model);
5    imports := 'javax.persistence.*';
6  }
```

Listing 29: Mapping *Entity::toPersistentPojoFile*

```
1  mapping GENERIC::data::Entity::toPersistentPojo(in model : GENERIC::WaslGenericModel) :
           HIBERNATE3::classes::PersistentPojo{
2    name := self.name;
3    nestingPackage := 'model.' + self.owningPackage.getQualifiedPackageName();
4    implementedInterfaces := model.map toFrameworkInterfaceSerializable(model);
5    variables := object JAVA6::PrimitiveVariable{
6      name := 'serialVersionUID';
7      isStatic := true;
8      type := JAVA6::PrimitiveType::long;
9      visibility := JAVA6::Visibility::private;
10     ...
11   };
12   variables += self.getValueProperties().map toPersistentPojoVariable(model);
13   ...
14   methods := self.getValueProperties().map toPersistentPojoGetter(model);
15   methods += self.getValueProperties().map toPersistentPojoSetter(model);
16   ...
17   annotations := object HIBERNATE3::annotations::Entity{};
18   ...
19 }
```

Listing 30: Mapping *Entity::toPersistentPojo*

Diese Vorteile treffen z. B. auch auf die Generierung von Accessoren zu. So lässt bei-

spielsweise die explizite Erstellung eines *ReferenceGetter* für ein *ValueProperty* durch das Mapping *ValueProperty::toPersistentPojoGetter* Einflussmöglichkeiten auf sämtliche Details wie die Sichtbarkeit zu (vgl. Listing 31).

```
1  mapping GENERIC::data::ValueProperty::toPersistentPojoGetter(in model : GENERIC::
           WaslGenericModel) : JAVADESIGNPATTERN::accessor::ReferenceGetter{
2    name := 'get' + self.name.firstToUpper();
3    type := self.primitiveType.map toFrameworkClass();
4    visibility := JAVA6::Visibility::public;
5    variable := self.map toPersistentPojoVariable(model);
6    ...
7  }
```

Listing 31: Mapping *ValueProperty::toPersistentPojoGetter*

Es wird sowohl jedes *ValueProperty* als auch jedes *ReferenceProperty* auf den Typ *PersistentPojoReferenceVariable* abgebildet, da eine partielle Abbildung auf den Typ *PersistentPojoPrimitiveVariable* induziert, dass erstens referenzierende Modellelemente für beide Arten von Variablen ausgelegt sein müssen und zweitens die Zahlendatentypen von *WASL Generic* auf Basisdatentypen von Java abgebildet würden, der String-Datentyp von *WASL Generic* dagegen auf die String-Frameworkklasse. Die damit verbundene Komplexitätssteigerung wird vermieden, indem sämtliche Basisdatentypen von *WASL Generic* auf die Wrapperklassen von Java für Basisdatentypen wie z. B. *java.lang.Integer* abgebildet werden.

Die Modellelemente für die Wrapperklassen werden durch das Mapping *PrimitiveDataTypes::toFrameworkClass* als Instanzen von *FrameworkClass* konstruiert, deren Name durch Aufruf der Query *PrimitiveDataTypes::toPrimitiveTypeName* in Abhängigkeit vom Basisdatentypen in *WASL Generic* typsicher vordefiniert ist (vgl. Listing 32).

```
1  mapping GENERIC::data::PrimitiveDataTypes::toFrameworkClass() : JAVA6::FrameworkClass{
2    name := self.toPrimitiveTypeName();
3  }
4
5  query GENERIC::data::PrimitiveDataTypes::toPrimitiveTypeName() : String{
6    return if self = GENERIC::data::PrimitiveDataTypes::String then 'java.lang.String' else
7      if self = GENERIC::data::PrimitiveDataTypes::Integer then 'java.lang.Integer' else
8        if self = GENERIC::data::PrimitiveDataTypes::RealNumber then 'java.lang.Float' else
9            ...
10       endif
11     endif
12   endif
13 }
```

Listing 32: Mapping *PrimitiveDataTypes::toFrameworkClass* und Query *PrimitiveDataTypes::toPrimitiveTypeName*

Mit dem Mapping *DataModel::toDaoDirectory* wird für das *DataModel* ein *Directory dao* angelegt (vgl. Listing 33), das mit dem Mapping *DataModel::toMainDAOFile* ein *File* für eine applikationsweite Datenzugriffsklasse enthält (vgl. Listing 34).

Die *DataAccessObject*-Klasse wird mit dem Mapping *DataModel::toMainDAO* erstellt

```
1  mapping GENERIC::data::DataModel::toDaoDirectory(in model : GENERIC::WaslGenericModel) :
           FILESYSTEM::Directory{
2    name := 'dao';
3    directoryElements := self.map toMainDAOFile(model);
4  }
```

Listing 33: Mapping *DataModel::toDaoDirectory*

```
1  mapping GENERIC::data::DataModel::toMainDAOFile(in model : GENERIC::WaslGenericModel) :
           JAVA6::ClassifierFile{
2    name := 'MainDAO';
3    ...
4    classifier := self.map toMainDAO(model);
5  }
```

Listing 34: Mapping *DataModel::toMainDAOFile*

und umfasst Methoden mit CRUD-Funktionalität, welche durch Aufruf der Mappings *Entity::toDAOMethodDAOLoad*, *Entity::toDAOMethodDAOSave*, etc. individuell für jedes *PersistentPojo* angelegt werden (vgl. Listing 35). Abhängig von der Anzahl der *Persistent-Pojos* ist alternativ die Erstellung individueller Datenzugriffsklassen sinnvoll und durch eine minimale Adaption der Transformation realisierbar. Die Datenzugriffsklasse wird als Singleton definiert, indem erstens die *Constructor*-Methode die Sichtbarkeit *private* erhält, zweitens mit dem Mapping *DataModel::toMainDAO_instance* eine statische und private Klassenvariable für das DataAccessObject angelegt wird (vgl. Listing 36) und drittens mit dem Mapping *DataModel::toMainDAO_getInstance* die Methode *getInstance* zur Ausgabe der Instanz erstellt wird (vgl. Listing 37).

```
1  mapping GENERIC::data::DataModel::toMainDAO(in model : GENERIC::WaslGenericModel) :
           HIBERNATE3::designpattern::dao::DataAccessObject{
2    name := 'MainDAO';
3    nestingPackage := 'dao';
4    variables := self.map toMainDAO_instance(model);
5    ...
6    methods := object JAVA6::Constructor{
7      name := 'MainDAO';
8      visibility := JAVA6::Visibility::private;
9      type := self.map toMainDAO(model);
10   };
11   methods += self.map toMainDAO_getInstance(model);
12   methods += self.ownedPackage.unfoldPackageToEntities().map toDAOMethodDAOLoad(model);
13   methods += self.ownedPackage.unfoldPackageToEntities().map toDAOMethodDAODelete(model);
14   methods += self.ownedPackage.unfoldPackageToEntities().map toDAOMethodDAOSave(model);
15   methods += self.ownedPackage.unfoldPackageToEntities().map toDAOMethodDAOList(model);
16   ...
17 }
```

Listing 35: Mapping *DataModel::toMainDAO*

Jede *DAOLoad*-Methode wird durch das Mapping *Entity::toDAOMethodDAOLoad* mit den Attributen ausgeprägt, die für die Generierung des Methodenrumpfs benötigt werden. Zu diesen gehören die EReferences *managedPersistentPojo* und *getCurrentSessionMethod* für die Information, über welchen Hibernate-EntityManager für welche Entitäts-Klasse Daten abzurufen sind (vgl. Listing 38).

```
mapping GENERIC::data::DataModel::toMainDAO_instance(in model : GENERIC::WaslGenericModel) :
          JAVA6::ReferenceVariable{
  name := 'instance';
  visibility := JAVA6::Visibility::private;
  isStatic := true;
  type := self.map toMainDAO(model);
}
```

Listing 36: Mapping *DataModel::toMainDAO_instance*

```
mapping GENERIC::data::DataModel::toMainDAO_getInstance(in model : GENERIC::WaslGenericModel
        ) : JAVADESIGNPATTERN::singleton::getInstance{
  name := 'getInstance';
  isStatic := true;
  visibility := JAVA6::Visibility::public;
  type := self.map toMainDAO(model);
  instanceProperty :=  self.map toMainDAO_instance(model);
}
```

Listing 37: Mapping *DataModel::toMainDAO_getInstance*

Im *Directory jsfcontroller* (vgl. Listing 39) wird für jedes *Entity* durch das Mapping *Entity::toController* eine JSF-Controllerklasse erstellt, welche Methoden für den Datenfluss zwischen Datenzugriffsobjekten und Facelets sowie optional weitere Präsentationslogik enthält (vgl. Listing 40). Zu diesen gehören sowohl Methoden zur Modifikation einzelner Entitäts-Klassen bzw. *PersistentPojos*, als auch solche zur Aufbereitung von Daten für Auswahllisten, z. B. erstellt durch den Aufruf des Mappings *Entity::toControllerMethodCSelectItemCompleteList*.

Jede der CRUD-Methoden wird durch ein korrespondierendes Mapping konstruiert, welches über die Schnittstelle der Methode hinaus Informationen für den Methodenrumpf wie z. B. im Fall des Mappings *toControllerMethodCLoad* die EReference *daoMethodLoad* für einen Verweis auf die entsprechende Lade-Methode des Datenzugriffsobjekts hinzufügt (vgl. Listing 41). Inter-Methodenreferenzen dieser Art sind ein Beispiel für die Anreicherung des Modells mit feingranularen Informationen, durch die das Modell an Aussagekraft gewinnt, implizite Annahmen über den Kontext des Methodenrumpfs vermieden werden und der Codegenerater vereinfacht wird.

Die XML-basierte Konfiguration des JSF-Frameworks wird durch das Mapping *Naviga-*

```
mapping GENERIC::data::Entity::toDAOMethodDAOLoad(in model : GENERIC::WaslGenericModel) :
          HIBERNATE3::designpattern::dao::daomethods::DAOLoad{
  name := 'load' + self.name;
  type := self.map toPersistentPojo(model);
  managedPersistentPojo := self.map toPersistentPojo(model);
  getCurrentSessionMethod := model.dataModel. map toMainDAO_getCS(model);
  parameters := object JAVA6::PrimitiveParameter {
    name := 'id';
    type := JAVA6::PrimitiveType::int;
    isMultiple := false;
  };
}
```

Listing 38: Mapping *Entity::toDAOMethodDAOLoad*

```
1  mapping GENERIC::navigation::NavigationModel::toControllerDirectory(in model : GENERIC::
           WaslGenericModel) : FILESYSTEM::Directory{
2    name := 'jsfcontroller';
3    directoryElements := model.dataModel.ownedPackage.unfoldPackageToEntities().map
             toControllerFile(model);
4  }
5
6  mapping GENERIC::data::Entity::toControllerFile(in model : GENERIC::WaslGenericModel) :
           JAVA6::ClassifierFile{
7    name := self.name + 'Controller';
8    extensionName := 'java';
9    classifier := self.map toController(model);
10 }
```

Listing 39: Mappings *NavigationModel::toControllerDirectory* und *Entity::toControllerFile*

```
1  mapping GENERIC::data::Entity::toController(in model : GENERIC::WaslGenericModel) : JSF1::
           faces::Controller{
2    name := self.name + 'Controller';
3    nestingPackage := 'web.jsfcontroller';
4    variables := self.map toControllerVariableCManagedPersistentPojo(model);
5    methods += self.map toControllerMethodCLoad(model);
6    methods += self.map toControllerMethodCDelete(model);
7    methods += self.map toControllerMethodCSelectItemCompleteList(model);
8    ...
9  }
```

Listing 40: Mapping *Entity::toController*

tionModel::toFacesConfigFile festgelegt, das ein Modellelement vom Typ *FacesConfigFile* konstruiert (vgl. Listing 42). Dieses enthält ein *DocumentRoot*, das den *FacesConfigType* als Wurzelelement des XML-Dokuments enthält. In diesem werden durch Aufruf der Mappings *Entity::toManagedBeanType* und *Node::toNavigationRuleType* die verwalteten Javaklassen und Navigationsregeln angelegt.

Zur Konstruktion des Navigationsgraphen wird für jeden *Node* mit dem Mapping *Node::toNavigationRuleType* eine Instanz der EClass *FacesConfigNavigationRuleType* angelegt (vgl. Listing 43). Diese erhält unter der EReference *fromViewId* einen Verweis auf den Quell-Knoten im Graphen bzw. das *Facelet*, das aus derjenigen *View* abgeleitet ist, die dem *LogicTuple* des *Node* zugeordnet ist. Für jeden ausgehenden *Link* des *Node* wird mit dem Mapping *Link::toNavigationCaseTypeRefBased* ein *FacesConfigNavigationCaseType-*

```
1  mapping GENERIC::data::Entity::toControllerMethodCLoad(in model : GENERIC::WaslGenericModel)
           : JSF1::faces::methodtypes::CLoad{
2    name := 'load';
3    type := JAVA6::PrimitiveType::void;
4    parameters := object JAVA6::ReferenceParameter{
5      name := 'event';
6      type := model.viewModel.map toFrameworkClassActionEvent();
7      isMultiple := false;
8    };
9    daoMethodLoad := self.map toDAOMethodDAOLoad(model);
10   managedVariable := self.map toControllerVariableCManagedPersistentPojo(model);
11   visibility := JAVA6::Visibility::public;
12 }
```

Listing 41: Mapping *Entity::toControllerMethodCLoad*

```
1  mapping GENERIC::navigation::NavigationModel::toFacesConfigFile(in model : GENERIC::
           WaslGenericModel) : JSF1::config::FacesConfigFile{
2    name := 'faces-config';
3    extensionName := 'xml';
4    documentRoot := object JSF12CONFIG::DocumentRoot{
5      facesConfig := object JSF12CONFIG::FacesConfigType{
6        managedBean := model.dataModel.ownedPackage.unfoldPackageToEntities().map
                 toManagedBeanType(model);
7        navigationRule += self.rootNodeGroup.unfoldToNodes().map toNavigationRuleType(model);
8      }
9    }
10 }
```

Listing 42: Mapping *NavigationModel::toFacesConfigFile*

```
1  mapping GENERIC::navigation::Node::toNavigationRuleType(in model : GENERIC::WaslGenericModel
           ) : JSF12CONFIG::FacesConfigNavigationRuleType{
2    fromViewId := object JSF12CONFIG::FacesConfigFromViewIdTypeRefBased{
3      fromViewIdRefBased := self.logicTuple.getPrimaryView().toFacelet(model);
4    };
5    navigationCase := self.outgoingLinks.map toNavigationCaseTypeRefBased(model);
6    navigationCase += model.navigationModel.rootMenuFolder.unfoldMenuFolderToMenuEntities().
             map toNavigationCaseTypeRefBased(model, self);
7  }
```

Listing 43: Mapping *Node::toNavigationRuleType*

RefBased angelegt (vgl. Listing 44). Letzterer verweist auf den Ziel-Knoten im Graphen bzw. dessen *Facelet*. Abschließend werden die Navigationsfälle durch solche für Menüeinträge ergänzt.

Analog wird die Hibernate-Konfigurationsdatei durch das Mapping *DataModel::toHibernateConfigFile* erzeugt, das ein Modellelement vom Typ *DocumentRoot* aus dem Metamodell *Hibernate30Config_XSD_Adapted* enthält, in dem das XML-Dokument mit seiner Baumstruktur nachgebildet wird (vgl. Listing 45).

Die XHTML-Templates in Form von Facelets sind in dem *Directory facelets* enthalten, das mit dem Mapping *ViewModel::toFaceletsDirectory* generiert wird und für jede *ViewGroup* im *ViewModel* durch Aufruf des Mappings *ViewGroup::toDirectory* rekursiv Sub-*Directories* erhält (vgl. Listing 46).

Für jede der *Views*, die in den *ViewGroups* enthalten sind, wird jeweils ein korrespondierendes *Facelet* konstruiert. Da die Containment-EReference *ownedViewElements* der *ViewGroup* polymorph *Views* unterschiedlichen Typs aufnimmt und für diese verschiedene Mappings definiert sind, wird die Fähigkeit von QVTO zum späten Binden anhand der

```
1  mapping GENERIC::navigation::Link::toNavigationCaseTypeRefBased(in model : GENERIC::
           WaslGenericModel) : JSF12CONFIG::FacesConfigNavigationCaseTypeRefBased{
2    fromOutcome := object JSF12CONFIG::String{
3      value := self.name;
4    };
5    toViewIdRefBased := self.destination.logicTuple.getPrimaryView().toFacelet(model);
6  }
```

Listing 44: Mapping *Link::toNavigationCaseTypeRefBased*

```
1  mapping GENERIC::data::DataModel::toHibernateConfigFile(in model : GENERIC::WaslGenericModel
           ) : HIBERNATE3::config::HibernateConfigFile{
2    name := 'hibernate.cfg';
3    extensionName := 'xml';
4    documentRoot := object HIBERNATE30CONFIG::DocumentRoot{
5      hibernateConfiguration := object HIBERNATE30CONFIG::HibernateConfigurationType{
6        sessionFactory := object HIBERNATE30CONFIG::SessionFactoryType{
7          ...
8        };
9      };
10   };
11 }
```

Listing 45: Mapping *DataModel::toHibernateConfigFile*

```
1  mapping GENERIC::presentation::ViewModel::toFaceletsDirectory(in model : GENERIC::
           WaslGenericModel) : FILESYSTEM::Directory{
2    name := 'facelets';
3    directoryElements := self.rootViewGroup.map toDirectory(model);
4  }
5
6  mapping GENERIC::presentation::ViewGroup::toDirectory(in model : GENERIC::WaslGenericModel)
           : FILESYSTEM::Directory{
7    name := self.name.toDirectoryName();
8    directoryElements := self.getOwnedViewGroups().map toDirectory(model);
9    directoryElements += self.getOwnedViews().toFacelet(model);
10 }
```

Listing 46: Mappings *ViewModel::toFaceletsDirectory* und *ViewGroup::toDirectory*

EClass durch Überladen der Query *View::toFacelet* mit den spezifischen Queries *ListView::toFacelet*, *EditView::toFacelet*, etc. genutzt. Die passende Query für eine *View* wird zur Laufzeit automatisch selektiert und durch diese das korrespondierende spezifische Mapping wie z. B. *EditView::toEditFacelet* aufgerufen (vgl. Listing 47). Die intermediären Queries dienen dazu, den spezifischen Rückgabetyp des jeweiligen Mappings wie z. B. *EditFacelet* auf den generischen Typ *Facelet* umzuinterpretieren, da das Überladen von Queries und Mappings identische Rückgabetypen voraussetzt.

```
1  query GENERIC::presentation::viewtypes::EditView::toFacelet(in model : GENERIC::
           WaslGenericModel) : JSF1::facelets::Facelet {
2    return self.map toEditFacelet(model).oclAsType(JSF1::facelets::Facelet);
3  }
```

Listing 47: Mapping *EditView::toFacelet*

Die Mappings für die verschiedenen *Facelets* konstruieren die benötigten Referenzen auf die zugrunde liegenden Modellelemente, wie z. B. im Fall des Mappings *EditView::toEditFacelet* einen Verweis auf die von JSF verwaltete *ManagedBean* unter der EReference *controller* oder den eintretenden *NavigationCase* unter der EReference *saveNavigationCase* für den Fall einer Speicheroperation (vgl. Listing 48).

Die Transformation umfasst insgesamt 84 Mappings, von denen jedes in der dargestellten Form eine EClass von *WASL Generic* einer EClass von *WASL JavaEE* zuordnet. Das Ergebnis der Transformation ist ein strukturähnliches Modell einer Webapplikation für Ja-

```
1  mapping GENERIC::presentation::viewtypes::EditView::toEditFacelet(in model : GENERIC::
           WaslGenericModel) : JSF1::facelets::facelettypes::EditFacelet{
2    name := self.name;
3    extensionName := 'jsp';
4    title := self.title;
5    controller := self.editTuple.entity.map toManagedBeanType(model);
6    faceletVariables := self.getValueProperties().map toFaceletScalarVariable_Edit(model, self
           );
7    saveNavigationCase := self.editTuple.node.getOutgoingLinks_ToListTuple()->first().map
           toNavigationCaseTypeRefBased(model);
8    ...
9  }
```

Listing 48: Mapping *EditView::toEditFacelet*

va EE, das geeignet ist, Quelltext für eine lauffähige Implementierung der Webapplikation vollständig automatisiert zu generieren. Aufgrund des Detailgrads des Modells und der geringen Abstraktionslücke zum Quelltext werden zudem potentiell Modifikationen durch Modellierer mit Plattformbezug ermöglicht.

4.7.3 Transformationsdefinition *WaslGeneric2WaslPython*

Die Transformationsdefinition *WaslGeneric2WaslPython* bildet Metamodellelemente des Metamodells von *WASL Generic* auf solche der Metamodelle von *WASL Python* ab, um das plattformspezifische Modell einer lauffähigen Webapplikation für die Google App Engine zu erzeugen.

Die QVTO-Transformation nimmt GENERIC-Modelle entgegen und gibt für jedes ein WASLPYTHON-Modell zurück, das durch Aufruf des Mappings *WaslGenericModel::toWaslPythonModel* in der Main-Operation erzeugt wird (vgl. Listing 49). Sie weist mit insgesamt 36 Mappings im Vergleich zu der Transformation *WaslGeneric2WaslJavaEE* einen geringeren Umfang auf, da auch die Implementierung vergleichsweise kompakt und von Konventionen geprägt ist. Die Bibliothek *WaslGeneric* wird wie bei der Transformationsdefinition *WaslGeneric2WaslJavaEE* eingebunden und somit wiederverwendet. Dies ist möglich, da die Bibliothek keine EClasses von *WASL PSM* referenziert.

```
1  import WaslGeneric;
2  modeltype GENERIC uses 'waslgeneric';
3  modeltype FILESYSTEM uses 'filesystem';
4  modeltype WASLPYTHON uses 'waslpython';
5  modeltype PYTHON2 uses 'python2';
6  ...
7
8  transformation WaslGeneric2WaslPython(in waslgeneric : GENERIC, out WASLPYTHON);
9
10 main() {
11   waslgeneric.rootObjects()[GENERIC::WaslGenericModel]->map toWaslPythonModel();
12 }
```

Listing 49: Mapping *WaslGeneric2WaslPython*

Das im *WaslPythonModel* enthaltene Dateisystemmodell wird in vergleichsweise einfach strukturierter Form konstruiert. Es enthält im Wesentlichen das *Directory app* für *WebappApplicationScripts*, das *Directory templates* für Django-Templatedateien, das *ModelScript model.py* für das Datenmodell und die *ApplicationYaml app.yaml* für die Applikationskonfiguration (vgl. Listing 50).

```
1 mapping GENERIC::WaslGenericModel::toWaslPythonModel() : WASLPYTHON::WaslPythonModel{
2   directoryElements := self.dataModel.map toModelScript(self);
3   directoryElements += self.navigationModel.map toAppDirectory(self);
4   directoryElements += self.viewModel.rootViewGroup.map toTemplatesRootDirectory(self);
5   directoryElements += self.navigationModel.map toApplicationConfig(self);
6   ...
7 }
```

Listing 50: Mapping *WaslGenericModel::toWaslPythonModel*

Das *ModelScript* wird mit dem Mapping *DataModel::toModelScript* angelegt und enthält für jedes *Entity* aus dem *DataModel* eine *ModelClass* (vgl. Listing 51).

Jede *ModelClass* wird durch das Mapping *Entity::toModelClass* erzeugt, das zur Re-

```
1  mapping GENERIC::data::DataModel::toModelScript(model : GENERIC::WaslGenericModel) :
           GOOGLEDATASTORE::ModelScript{
2    name := 'model';
3    extensionName := 'py';
4    imports := 'from google.appengine.ext import db';
5    classes := self.ownedPackage.unfoldPackageToEntities().map toModelClass(model);
6  }
```

Listing 51: Mapping *DataModel::toModelScript*

ferenzierung der Oberklasse das Mapping *DataModel::toFrameworkClassDbModel* aufruft und die Referenz unter der EReference *superClasses* speichert (vgl. Listing 52). Die *ValueProperties* und *ReferenceProperties* des *Entity* werden auf Instanzen der Spezialisierungen der EClass *Property* aus dem GoogleDatastore-Metamodell abgebildet.

```
1  mapping GENERIC::data::Entity::toModelClass(model : GENERIC::WaslGenericModel) :
           GOOGLEDATASTORE::ModelClass{
2    name := self.name;
3    superClasses := model.dataModel.map toFrameworkClassDbModel();
4    variables := self.ownedProperties.toDataStoreProperty(model);
5  }
6
7  mapping GENERIC::data::DataModel::toFrameworkClassDbModel() : PYTHON2::FrameworkClass{
8    name := 'db.Model';
9  }
```

Listing 52: Mappings *Entity::toModelClass* und *DataModel::toFrameworkClassDbModel*

Da im Quellmodell der Typ eines *ValueProperty* durch sein EAttribute *primitiveType* spezifiziert wird, im Zielmodell dagegen durch die typisierende EClass wie z. B. *StringProperty* oder *IntegerProperty*, wird mit der Query *ValueProperty::toDataStoreProperty* der Wert des *primitiveType* abgefragt und explizit das passende Mapping aufgerufen (vgl. Listing 53).

```
1  query GENERIC::data::ValueProperty::toDataStoreProperty(model : GENERIC::WaslGenericModel) :
            GOOGLEDATASTORE::Property{
2    return if self.primitiveType = GENERIC::data::PrimitiveDataTypes::String then self.map
              toStringProperty(model)
3      else if self.primitiveType = GENERIC::data::PrimitiveDataTypes::Integer then self.map
                toIntegerProperty(model)
4        else self.map toStringProperty(model)
5      endif
6    endif
7  }
8
9  mapping GENERIC::data::ValueProperty::toStringProperty(model : GENERIC::WaslGenericModel) :
           GOOGLEDATASTORE::propertyclasses::StringProperty{
10   name := self.name;
11 }
12
13 mapping GENERIC::data::ValueProperty::toIntegerProperty(model : GENERIC::WaslGenericModel) :
            GOOGLEDATASTORE::propertyclasses::IntegerProperty{
14   name := self.name;
15 }
```

Listing 53: Query *ValueProperty::toDataStoreProperty* mit assoziierten Mappings

Mit dem Mapping *NavigationModel::toAppDirectory* wird im Dateisystemmodell das *Directory app* angelegt, das durch Aufruf des Mappings *NodeGroup::toWebappApplication-*

Script für jede *NodeGroup* aus dem *NavigationModel* ein *WebappApplicationScript* enthält (vgl. Listing 54).

```
1  mapping GENERIC::navigation::NavigationModel::toAppDirectory(model : GENERIC::
           WaslGenericModel) : FILESYSTEM::Directory{
2    name := 'app';
3    directoryElements := self.rootNodeGroup.getOwnedNodeGroups().map toWebappApplicationScript
             (model);
4  }
```

Listing 54: Mapping *NavigationModel::toAppDirectory*

Die *Nodes* der *NodeGroup* bzw. deren *LogicTuples* werden in RequestHandler-Klassen transformiert, indem das Mapping *NodeGroup::toWebappApplicationScript* jeweils das Mapping *LogicTuple::toRequestHandlerClass* aufruft (vgl. Listing 55). Zusätzlich wird dem Skript ein *WSGIApplicationObject* mit *UrlMappings* hinzugefügt, die den RequestHandler-Klassen URLs zuordnen.

```
1  mapping GENERIC::navigation::NodeGroup::toWebappApplicationScript(model : GENERIC::
           WaslGenericModel) : GOOGLEWEBAPP::WebappApplicationScript{
2    name := self.name.toLower();
3    extensionName := 'py';
4    imports := 'import model';
5    ...
6    classes := self.getOwnedNodes().logicTuple.map toRequestHandlerClass(model);
7    wsgiApplicationObject := object GOOGLEWEBAPP::WSGIApplicationObject{
8      urlMappings := self.getOwnedNodes().map toUrlMapping(model);
9      ...
10   };
11 }
```

Listing 55: Mapping *NodeGroup::toWebappApplicationScript*

Jede RequestHandler-Klasse erhält entsprechend dem *LogicTuple* eine oder mehrere Methoden zur Behandlung des HTTP-GET- bzw. HTTP-POST-Requests. Dazu wird durch das Mapping *LogicTuple::toRequestHandlerClass* die Query *LogicTuple::toMethod* bzw. durch spätes Binden eine der spezialisierten Varianten *ListTuple::toMethod*, *CreateTuple::toMethod*, etc. aufgerufen (vgl. Listing 56).

```
1  mapping GENERIC::logic::LogicTuple::toRequestHandlerClass(model : GENERIC::WaslGenericModel)
             : PYTHON2::Class{
2    name := self.node.name.getRequestHandlerClassName();
3    methods := self.toMethod(model);
4    ...
5  }
```

Listing 56: Mapping *LogicTuple::toRequestHandlerClass*

Im Fall eines *EditTuple* ist dies die Query *EditTuple::toMethod*, die sowohl eine *RequestHandlerMethodEditGet*, als auch eine *RequestHandlerMethodEditPost* durch Aufruf der entsprechenden Mappings zurückgibt (vgl. Listing 57). Die Mappings prägen das EAttribute *name* sowie die EReference *parameters* entsprechend den Konventionen des web-

app-Frameworks aus und fügen weitere benötigte Angaben wie z. B. über die verwaltete *ModelClass* oder das auszugebende *Template* hinzu (vgl. Listing 58).

```
query GENERIC::logic::logictupletypes::EditTuple::toMethod(model : GENERIC::WaslGenericModel
        ) : OrderedSet(GOOGLEWEBAPP::RequestHandlerMethod){
  return self.map toRequestHandlerMethodEditGet(model).oclAsType(GOOGLEWEBAPP::
           RequestHandlerMethod)->asSequence()
    ->union(self.map toRequestHandlerMethodEditPost(model)->asSequence())
    ->asOrderedSet();
}
```

Listing 57: Mapping *EditTuple::toMethod*

```
mapping GENERIC::logic::logictupletypes::EditTuple::toRequestHandlerMethodEditGet(model :
        GENERIC::WaslGenericModel) : GOOGLEWEBAPP::requesthandlermethodtypes::
        RequestHandlerMethodEditGet{
  name := 'get';
  parameters := object PYTHON2::Parameter{
    name := 'self';
  };
  managedModelClass := self.entity.map toModelClass(model);
  template := self.view.map toTemplate(model);
  ...
}
```

Listing 58: Mapping *EditTuple::toRequestHandlerMethodEditGet*

Die *Views* des *WASL Generic*-Modells werden entsprechend ihrer Verortung in der *ViewGroup*-Hierarchie in eine *Directory*-Hierarchie eingeordnet, die in dem Wurzel-*Directory templates* enthalten ist. Jede der *Views* wird durch das Mapping *View::toTemplate* bzw. dessen Verfeinerungen in ein *Template* transformiert. Im Fall des Mappings *EditView::toTemplate* wird beispielsweise eine Mischung aus XHTML-Inhalten und einer *DjangoEditComponent* generiert, welche mit den benötigten Eigenschaften bzgl. der darzustellenden Variablen und einer Submit-URL ausgeprägt wird, anhand derer der Quelltext durch den Codegenerator erzeugt werden kann (vgl. Listing 59).

```
mapping GENERIC::presentation::viewtypes::EditView::toTemplate(model : GENERIC::
        WaslGenericModel) : DJANGO::template::Template{
  name := self.name;
  extensionName := 'html';
  title := self.title;
  inline := object XHTML1::PType{
    text := object XHTML1::TextWrapper{
      text := self.description;
    };
  };
  djangoComponent += object DJANGO::template::djangocomponents::DjangoEditComponent{
    variables := self.editTuple.entity.ownedProperties.toDataStoreProperty(model);
    hrefSubmitUrl := self.editTuple.node.toUrl();
  };
}
```

Listing 59: Mapping *EditView::toTemplate*

4.7.4 Transformationsdefinition *WaslGeneric2WaslPHP*

Die Transformationsdefinition *WaslGeneric2WaslPHP* bildet Elemente des Metamodells von *WASL Generic* auf solche der Metamodelle von *WASL PHP* ab und gibt als Resultat Modelle PHP-basierter Webapplikationen mit schwach ausgeprägtem Einsatz von Frameworks aus.

Die QVTO-Transformation bindet ebenfalls die Bibliothek *WaslGeneric* sowie die benötigten Metamodelle ein, nimmt als eingehenden Parameter ein GENERIC-Modell entgegen und gibt ein WASLPHP-Modell zurück, das basierend auf 66 Mappings erstellt wird. Dazu wird in der Main-Operation durch Aufruf der Mapping-Operation *WaslDataModel::toWaslPHPModel* für jedes Wurzel-Modellelement vom Typ *WaslGenericModel* ein Modellelement vom Typ *WaslPHPModel* erzeugt (vgl. Listing 60).

```
1  import WaslGeneric;
2  modeltype GENERIC uses 'waslgeneric';
3  modeltype FILESYSTEM uses 'filesystem';
4  modeltype WASLPHP uses 'waslphp';
5  modeltype PHP5 uses 'php5';
6  ...
7
8  transformation WaslGeneric2WaslPHP(in waslgeneric : GENERIC, out WASLPHP);
9
10 main() {
11   waslgeneric.rootObjects()[GENERIC::WaslGenericModel]->map toWaslPHPModel();
12 }
```

Listing 60: Transformation *WaslGeneric2WaslPHP*

Durch das Mapping *WaslGenericModel::toWaslPHPModel* wird im Wesentlichen ein Dateisystemmodell erstellt, das auf der Wurzelebene die *Directories lib* für Bibliotheken, *pages* für Seitentemplates und *doctrine* für das Doctrine-Datenmodell sowie das *PHPIndexScript* mit dem Namen *index.php* enthält (vgl. Listing 61). Das *Directory lib* umfasst die *Directories own* für selbsterstellte PHP-Bibliotheken und -Klassen inkl. Datenzugriffs- und MVC-Controllerklassen, *thirdparty* für Drittanbieter-Frameworks und *frame* für applikationsweite Seitenelemente, z. B. für das Menü und die Kopf- sowie Fußleiste der Webseiten.

Die Konfiguration für das ORM-Framework Doctrine 2 wird redundant in Form von YAML-Dateien, XML-Dateien und annotierten PHP-Klassen erzeugt, die mit dem Mapping *DataModel::toDoctrineDirectory* in Unterverzeichnissen des *Directory doctrine* angelegt werden (vgl. Listing 62).

Die *PHPScripts* der annotierten *EntityClasses* werden im Unterverzeichnis *entities* in einer *Directory*-Hierarchie abgelegt, die mit der *Package*-Hierarchie des Eingangsmodells korrespondiert (vgl. Listing 63).

Jedes *Entity* wird mit dem Mapping *Entity::toPHPScript* in ein *PHPScript* transformiert, das einen PHP-*Namespace* passend zu der relativen Verortung in der Dateisys-

```
mapping GENERIC::WaslGenericModel::toWaslPHPModel() : WASLPHP::WaslPHPModel{
  directoryElements := object FILESYSTEM::Directory{
    name := 'lib';
    directoryElements := object FILESYSTEM::Directory{
      name := 'own';
      directoryElements := self.dataModel.map toDAODirectory(self);
      directoryElements += object FILESYSTEM::Directory{
        name := 'logic';
        directoryElements := self.navigationModel.map toTargetPathPHPScript(self);
        ...
      };
      directoryElements += self.map toConfigScript(self);
    };
    directoryElements += object FILESYSTEM::Directory{
      name := 'thirdparty';
    };
    directoryElements += object FILESYSTEM::Directory{
      name := 'frame';
      directoryElements := self.map toPHPHeaderScript(self);
      directoryElements += self.map toPHPFooterScript(self);
      directoryElements += self.navigationModel.map toPHPMenuIncludeScript(self);
    };
    directoryElements += self.map toInitializeScript(self);
  };
  directoryElements += self.viewModel.rootViewGroup.map toPagesRootDirectory(self);
  directoryElements += self.dataModel.map toDoctrineDirectory(self);
  directoryElements += self.navigationModel.map toIndexPHPScript(self);
}
```

Listing 61: Mapping *WaslGenericModel::toWaslPHPModel*

```
mapping GENERIC::data::DataModel::toDoctrineDirectory(in model : GENERIC::WaslGenericModel)
        : FILESYSTEM::Directory{
  name := 'doctrine';
  directoryElements := self.map toYamlDirectory(model);
  directoryElements += self.map toXmlDirectory(model);
  directoryElements += self.map toEntitiesDirectory(model);
  ...
}
```

Listing 62: Mapping *DataModel::toDoctrineDirectory*

tem-Hierarchie aufweist und in diesem *Namespace* durch Aufruf des Mappings *Entity::toEntityClass* eine *EntityClass* erhält (vgl. Listing 64).

Die *EntityClass* besitzt eine *Docblock*-Annotation zur Kennzeichnung als *Entity* und zur Explikation der assoziierten *Table* in der Datenbank (vgl. Listing 65) sowie Variablen und Accessor-Methoden, welche aus den *ValueProperties* und *ReferenceProperties* des *Entity* abgeleitet werden und mit weiteren Annotationen versehen werden.

So wird beispielsweise im Fall eines *ValueProperty* die resultierende *Variable* zur Typisierung mit dem Mapping *ValueProperty::toDoctrineVariable* durch eine *Column*-Annotation ausgezeichnet (vgl. Listing 66). Die Zuordnung von Basisdatentypen des Metamodells von *WASL Generic* zu solchen des Metamodells *Doctrine2* geschieht durch die Query *PrimitiveDataTypes::primitive2Primitive*. Da primitive Datentypen als Elemente des Metamodells von *WASL Generic* definiert sind, können sie typsicher durch die Query referenziert werden.

Im *Directory dao* wird zudem für jedes *Entity* jeweils ein *PHPScript* mit einer Da-

```
1  mapping GENERIC::data::DataModel::toEntitiesDirectory(in model : GENERIC::WaslGenericModel)
           : FILESYSTEM::Directory{
2    name := 'entities';
3    directoryElements := self.ownedPackage.map toEntitiesDirectory(model);
4  }
5
6  mapping GENERIC::data::Package::toEntitiesDirectory(in model : GENERIC::WaslGenericModel) :
           FILESYSTEM::Directory{
7    name := self.name;
8    directoryElements := self.ownedPackages.map toEntitiesDirectory(model);
9    directoryElements += self.ownedEntities.map toPHPScript(model);
10 }
```

Listing 63: Mappings *DataModel::toEntitiesDirectory* und *Package::toEntitiesDirectory*

```
1  mapping GENERIC::data::Entity::toPHPScript(in model : GENERIC::WaslGenericModel) : PHP5::
           PHPScript{
2    name := self.name.firstToUpper();
3    extensionName := 'php';
4    phpScriptElements := object PHP5::Namespace{
5      name := self.owningPackage.getDoctrinePackageString('\\');
6      namespaceElements := self.map toEntityClass(model);
7    };
8  }
```

Listing 64: Mapping *Entity::toPHPScript*

tenzugriffsklasse generiert (vgl. Listing 68), das Methoden vom Typ *Fetch* und *FetchAll* enthält (vgl. Listing 69).

Das *Directory pages* wird mit der Dateisystemstruktur und *PHPWebPageScripts* als Inhalt generiert, indem ausgehend vom *ViewModel* dessen *ViewGroup*-Hierarchie rekursiv in eine *Directory*-Hierarchie überführt wird (vgl. Listing 70) und für jede *View* das Mapping *View::toPHPWebPageScript* bzw. durch spätes Binden dessen Verfeinerungen wie z. B. *EditView::toPHPWebPageScript* aufgerufen werden.

Das Mapping *EditView::toPHPWebPageScript* stellt den Inhalt des Skripts als Mischung aus XHTML-Inhalten und einer *PHPEditComponent* zusammen, welche aggregiert die benötigten Informationen für das Eingabeformular speichert (vgl. Listing 71). Darüber hinaus wird die *id* aus dem zugeordneten *Node* ermittelt, um einen selbstreferentiellen Verweis auf das *PHPWebPageScript* zur Verarbeitung von HTTP-POST-Requests nach einem

```
1  mapping GENERIC::data::Entity::toEntityClass(in model : GENERIC::WaslGenericModel) :
           DOCTRINE2::orm::annotations::EntityClass{
2    name := self.name;
3    comments := object DOCTRINE2::orm::annotations::Docblock{
4      annotations := object DOCTRINE2::orm::annotations::Entity{};
5      annotations += object DOCTRINE2::orm::annotations::Table{
6        name := self.getTableName();
7      };
8    };
9    variables := self.ownedProperties.map toDoctrineVariable(model);
10   methods := self.ownedProperties.map toGetter(model);
11   methods += self.ownedProperties.map toSetter(model);
12 }
```

Listing 65: Mapping *Entity::toEntityClass*

```
mapping GENERIC::data::ValueProperty::toDoctrineVariable(in model : GENERIC::
        WaslGenericModel) : PHP5::Variable{
  name := self.name;
  visibility := PHP5::VisibilityKind::private;
  comments := object DOCTRINE2::orm::annotations::Docblock{
    annotations := object DOCTRINE2::orm::annotations::Column{
      type := primitive2Primitive(self.primitiveType);
      name := self.getColumnName();
      length := self.getLength();
    };
    ...
  };
}
```

Listing 66: Mapping *ValueProperty::toDoctrineVariable*

```
query primitive2Primitive(in type : GENERIC::data::PrimitiveDataTypes) : DOCTRINE2::orm::
        Type {
  return if type = GENERIC::data::PrimitiveDataTypes::String then return DOCTRINE2::orm::
          Type::string else
    if type = GENERIC::data::PrimitiveDataTypes::Integer then return DOCTRINE2::orm::Type::
            integer else
      if type = GENERIC::data::PrimitiveDataTypes::Timestamp then return DOCTRINE2::orm::
              Type::string else
        ...
      endif
    endif
  endif
}
```

Listing 67: Query *PrimitiveDataTypes::primitive2Primitive*

Submit des Formulars generieren zu können.

```
mapping GENERIC::data::DataModel::toDAODirectory(in model : GENERIC::WaslGenericModel) :
        FILESYSTEM::Directory{
  name := 'dao';
  directoryElements := self.ownedPackage.unfoldPackageToEntities().map toDAOScript(model);
}
```

Listing 68: Mapping *DataModel::toDAODirectory*

```
1  mapping GENERIC::data::Entity::toDAOScript(in model : GENERIC::WaslGenericModel) : PHP5::
           PHPScript{
2    name := 'DAO' + self.name.firstToUpper();
3    extensionName := 'php';
4    phpScriptElements := object PHP5::Namespace{
5      name := 'lib\\own\\dao';
6      namespaceElements := object DOCTRINE2::orm::dao::DataAccessObject{
7        name := 'DAO' + self.name;
8        methods := self.map toDAOMethodFetch(model);
9        methods += self.map toDAOMethodFetchAll(model);
10     };
11   };
12 }
```

Listing 69: Mapping *Entity::toDAOScript*

```
1  mapping GENERIC::presentation::ViewGroup::toPagesRootDirectory(in model : GENERIC::
           WaslGenericModel) : FILESYSTEM::Directory{
2    name := 'pages';
3    directoryElements := self.getOwnedViewGroups().map toPHPWebPageScriptDirectory(model);
4    directoryElements += self.getOwnedViews().map toPHPWebPageScript(model);
5  }
6
7  mapping GENERIC::presentation::ViewGroup::toPHPWebPageScriptDirectory(in model : GENERIC::
           WaslGenericModel) : FILESYSTEM::Directory{
8    name := self.name.toDirectoryName();
9    directoryElements := self.getOwnedViewGroups().map toPHPWebPageScriptDirectory(model);
10   directoryElements += self.getOwnedViews().map toPHPWebPageScript(model);
11 }
```

Listing 70: Mappings *ViewGroup::toPagesRootDirectory* und *ViewGroup::toPHPWebPage-ScriptDirectory*

```
1  mapping GENERIC::presentation::viewtypes::EditView::toPHPWebPageScript(in model : GENERIC::
           WaslGenericModel) : WEB::PHPWebPageScript{
2    id := self.editTuple.node.toNodeId();
3    name := self.name;
4    extensionName := 'php';
5    title := self.title;
6    inline := object XHTML1::PType{
7      text := object XHTML1::TextWrapper{text := self.description;};
8    };
9    phpComponent += object WEB::phpcomponents::PHPEditComponent{
10     recordMethod := self.editTuple.entity.map toDAOMethodFetch(model);
11     inputs := self.editTuple.entity.getScalarProperties().toEditInput(model);
12     hrefSubmitUrl := self.editTuple.node.getOutgoingNodes_WithListTuple()->first().toUrl(
             model);
13   };
14 }
```

Listing 71: Mapping *EditView::toPHPWebPageScript*

4.8 Modell-zu-Text-Transformationsdefinitionen

Die Codegeneratoren des WASL-Generatorframeworks bestehen aus M2T-Transformationsdefinitionen, die jeweils individuell für eines der Metamodelle von *WASL PSM* spezifiziert sind. Jede der Transformationsdefinitionen ist eine Zusammenstellung von Templatedateien und Funktionsdefinitionen zur Erzeugung von Quelltext, die auch als Templatepaket bezeichnet wird.

Als Transformationssprache zur Formulierung der Transformationsdefinitionen dient die Templatesprache Xpand (vgl. 2.7.2). Der Wahl einer Transformationssprache liegen als Kriterien in erster Linie der Reifegrad und die Etablierung in der Praxis zugrunde, sowie in zweiter Linie die Standardkonformität. Als etablierter Bestandteil des *Eclipse Model to Text* (M2T) Projekts und mit einer ausgedehnten industriellen Entwicklungshistorie im Rahmen der openArchitectureWare werden die beiden ersten Kriterien zwar erfüllt, Standardkonformität dagegen nicht. Von einer Verwendung der MOF Model to Text Transformation Language als standardisierter Transformationssprache wird abgesehen, da deren Implementierung durch Acceleo 3 eine neuere Entwicklung ist. Die Verwendung von Xpand im Rahmen des WASL-Generatorframeworks ist exemplarisch für die Anwendung einer Templatesprache für die Codegeneratoren einzuordnen. Aufgrund der geringen Abstraktionslücke zwischen dem PSM und dem Quelltext weisen die M2T-Transformationsdefinitionen eine geringe Komplexität auf, da auf komplexe Generatorlogik verzichtet werden kann, und beanspruchen somit lediglich einen minimalen Funktionsumfang der Transformationssprache. Insofern kann die Transformationstechnik für den Codegenerator verhältnismäßig einfach substituiert werden, solange die Alternative ebenfalls kompatibel zu EMF ist.

Bei der Ausführung des Codegenerators wird initial aus dem Template *templates::Root.xpt* der *DEFINE-Block* mit dem Namen *Root* aufgerufen, der abhängig vom Modelltyp als Eingabe ein *WaslJavaEEModel*, *WaslPythonModel* oder *WaslPHPModel* erwartet. Entsprechend zu deren Spezifikation in den Metamodellen *WaslJavaEE*, *WaslPython* und *WaslPHP* weisen die Eingabemodelle jeweils die EReference *directoryElements* für *DirectoryElements* auf. Der DEFINE-Block des Templates expandiert für jedes *DirectoryElement* abhängig vom Typ den entsprechenden DEFINE-Block *DirectoryElement* aus dem Template *FilesystemTraverser.xpt*, das ebenfalls im Verzeichnis *templates* liegt (*templates::FilesystemTraverser.xpt*, vgl. Listing 72).

```
1 «DEFINE Root FOR wasljavaee::WaslJavaEEModel»
2   «EXPAND FilesystemTraverser::DirectoryElement FOREACH this.directoryElements»
3 «ENDDEFINE»
```

Listing 72: Template *templates::Root.xpt*

Das Template *templates::FilesystemTraverser.xpt* umfasst DEFINE-Blöcke, die sämtlich den Namen *DirectoryElement* aufweisen und für die diversen Spezialisierungen der EClass *File* in den verschiedenen Metamodellen definiert sind (vgl. Listing 73). Das Template traversiert rekursiv das Dateisystemmodell als Ausschnitt aus der Containment-Hierarchie des gesamten Modells. Dazu wird mit dem DEFINE-Block *DirectoryElement* für die EClass *Directory* die EReference *directoryElements* durchlaufen und für jedes darin enthaltene *DirectoryElement* rekursiv der DEFINE-Block *DirectoryElement* expandiert. Für *DirectoryElements*, deren Typ die EClass *File* spezialisiert, werden die alternativen DEFINE-Blöcke im Template expandiert, wie z. B. der DEFINE-Block *DirectoryElement* für den Typ *ClassifierFile*. Zur Laufzeit des Codegenerators wird durch spätes Binden automatisch derjenige DEFINE-Block für ein Modellelement gewählt, dessen Typ in der Typhierarchie zu dem Typ des Modellelements die geringste Distanz aufweist. Das Template dient der zentralen Konfiguration des Codegenerators, indem die Templatepakete für die verschiedenen Metamodelle ähnlich zu dem Metamodell *Filesystem* über die DEFINE-Blöcke registriert und integriert werden.

```
1  «DEFINE DirectoryElement FOR filesystem::Directory»
2    «EXPAND DirectoryElement FOREACH this.directoryElements»
3  «ENDDEFINE»
4
5  «DEFINE DirectoryElement FOR java6::ClassifierFile»
6    «EXPAND templates::java6::DirectoryElement::DirectoryElement»
7  «ENDDEFINE»
8
9  «DEFINE DirectoryElement FOR hibernate3::config::HibernateConfigFile»
10   «EXPAND templates::hibernate3::DirectoryElement::DirectoryElement»
11 «ENDDEFINE»
12
13 «DEFINE DirectoryElement FOR jsf1::facelets::Facelet»
14   «EXPAND templates::jsf1::DirectoryElement::DirectoryElement»
15 «ENDDEFINE»
16 ...
```

Listing 73: Template *templates::FilesystemTraverser.xpt*

Jedes Templatepaket für ein Metamodell muss per Konvention als Einstiegspunkt ein Template mit dem Namen *DirectoryElement* bereitstellen, das gleichnamige DEFINE-Blöcke für die verschiedenen Spezialisierungen der EClass *File* aus dem Metamodell enthält. Die DEFINE-Blöcke dienen als Schnittstelle gegenüber dem *FileTraverser*-Template und leiten die EXPAND-Aufrufe an nachgelagerte Templates zur Generierung der Quelltext-Dateien weiter. Somit ist sichergestellt, dass für jedes Templatepaket dieselbe Schnittstelle bereitgestellt wird. Beispielsweise enthält das Templatepaket *java6* für das Metamodell *Java6* ein solches Template, das den EXPAND-Aufruf zur Generierung eines *ClassifierFile* an das interne Template *template::java6::ClassifierFile* und dessen DEFINE-Block *Root* weiterleitet (vgl. Listing 74).

Jedes sonstige Template eines Templatepakets besitzt per Konvention einen DEFINE-

```
1  «DEFINE DirectoryElement FOR java6::ClassifierFile»
2    «EXPAND templates::java6::ClassifierFile::Root»
3  «ENDDEFINE»
```

Listing 74: Template *templates::java6::DirectoryElement.xpt*

Block namens *Root* für diejenige EClass, nach der das Template benannt ist. So enthält z. B. das Template *templates::java6::ClassifierFile.xpt* den DEFINE-Block *Root* für die EClass *java6::ClassifierFile* (vgl. Listing 75). Dies stellt sicher, dass die EClass eindeutig identifiziert wird, die das Template prägt und von diesem primär adressiert wird. Die EXPAND-Aufrufe innerhalb eines DEFINE-Blocks orientieren sich weitestgehend an den Containment-EReferences der adressierten EClass. Da z. B. die EClass *ClassifierFile* über die Containment-EReference *classifier* auf die EClass *Classifier* verweist, expandiert der DEFINE-Block *Root* für die EClass *ClassifierFile* den DEFINE-Block *Root* für die EClass *Classifier*, ruft diesen also zur Erzeugung des Klassenrumpfs auf.

Im Detail generiert das Template *templates::java6::ClassifierFile.xpt* eine Java-Klassendatei für jedes *ClassifierFile*, befüllt diese mit textuellen Referenzen auf das Java-Package sowie Java-Imports und expandiert wie beschrieben den DEFINE-Block *Root* im Template *templates::java6::Classifier*.

```
1  «EXTENSION templates::filesystem::Filesystem»
2
3  «DEFINE Root FOR java6::ClassifierFile»
4  «FILE this.getPath()»«IF this.classifier.nestingPackage != null»package «this.classifier.
           nestingPackage»;«ENDIF»
5
6  «FOREACH this.imports AS import»import «import»;
7  «ENDFOREACH»
8  «EXPAND templates::java6::Classifier::Root FOR this.classifier»
9  «ENDFILE»
10 «ENDDEFINE»
```

Listing 75: Template *templates::java6::ClassifierFile.xpt*

Der Aufruf *this.getPath* im FILE-Tag berechnet aus der Verortung des *ClassifierFile* im Dateisystemmodell eine textuelle Repräsentation des Pfades im Projektverzeichnis. Die Funktion *getPath(filesystem::File f)* ist in der Xtend-Datei *templates::filesystem::Filesystem.ext* definiert (vgl. Listing 76), und ruft die Funktion *getPathUntil(filesystem::File f, String stopDirectoryName)* mit dem *File* als erstem und einem leeren String als zweitem Parameter auf. Diese Funktion konkateniert den Pfad, indem der Name des *File* an den Pfad der *Directories* gehängt wird. Letzterer wird rekursiv durch wiederholten Aufruf der Funktion *getPathUntil(filesystem::Directory d, String stopDirectoryName)* ermittelt, die solange auf dem übergeordneten Verzeichnis aufgerufen wird, wie ein solches existiert und dieses nicht einen Namen aufweist, der dem String-Parameter *stopDirectoryName* entspricht. Der Parameter *stopDirectoryName* dient dazu, die Pfadberechnung

auf einer vordefinierbaren Dateisystemebene abbrechen zu lassen. Dies wird z. B. benötigt, wenn bei Projekten mit *WaslJavaEE*-Modellen textuelle Dateipfade relativ zu dem Ordner *war* zu konstruieren sind.

Die Funktion *getNestingDirectory(filesystem::DirectoryElement de)* liefert den Wert der EReference *nestingDirectory* eines *DirectoryElement* zurück. Die direkte Abfrage des Werts der EReference *nestingDirectory* gibt aufgrund einer fehlerhaften Implementierung der Ausführungsengine für die Transformationsdefinition keine Referenz aus, falls die EReference auf ein Modellelement verweist, dessen EClass nicht im selben Metamodell liegt, wie die EClass der EReference. Aus diesem Grund muss die Funktion *getNestingDirectory* verwendet werden, die den *eContainer* des *DirectoryElement* zurückgibt. Diese Implementierung ist praktikabel, da *nestingDirectory* die wesentliche Containment-EReference für ein *DirectoryElement* ist und somit wertmäßig mit der EReference *eContainer* übereinstimmt.

```
1  Boolean hasNestingDirectory(filesystem::DirectoryElement de) :
2    de.eContainer.metaType == filesystem::Directory;
3  filesystem::Directory getNestingDirectory(filesystem::DirectoryElement de) :
4    de.hasNestingDirectory() ? de.eContainer : null;
5
6  String getName(filesystem::DirectoryElement de) :
7    de.name;
8  String getName(filesystem::File f) :
9    f.extensionName.length > 0 ? f.name + '.' + f.extensionName : f.name;
10
11 String getPathUntil(filesystem::Directory d, String stopDirectoryName) :
12   d.hasNestingDirectory() && d.getNestingDirectory().getName() != stopDirectoryName ?
13     d.getNestingDirectory().getPathUntil(stopDirectoryName) + '/' + d.getName() :
14     d.getName();
15 String getPathUntil(filesystem::File f, String stopDirectoryName) :
16   f.hasNestingDirectory() && f.getNestingDirectory().getName() != stopDirectoryName ?
17     f.getNestingDirectory().getPathUntil(stopDirectoryName) + '/' + f.getName() :
18     f.getName();
19
20 String getPath(filesystem::File f) :
21   f.getPathUntil('');
22 ...
```

Listing 76: Xtend-Datei *templates::filesystem::Filesystem.ext*

Das Template *templates::java6::Classifier.xpt* deckt exemplarisch die Generierung von Javaklassen und -interfaces ab (vgl. Listing 77). Es adressiert entsprechend dem Template-Namen die EClass *Classifier* sowie deren Spezialisierungen *Class* und *Interface* aus dem Metamodell *Java6* durch DEFINE-Blöcke mit dem Namen *Root*. Der Rumpf des DEFINE-Blocks für die EClass *Classifier* ist nicht spezifiziert, da die EClass abstrakt ist und in Java keine Entsprechung existiert. Der DEFINE-Block für die EClass *Class* deckt initial die Expandierung des DEFINE-Blocks *Annotation* für jede Annotation der *Class* ab, deklariert anschließend optional die Java-Klasse als abstrakt, als Unterklasse, oder als Implementierung von Interfaces und expandiert abschließend DEFINE-Blöcke für Variablen- und Methodendefinitionen. Der DEFINE-Block *Interface* erzeugt Variablen in Form von Kon-

stanten und Methodensignaturen. Da sämtliche der DEFINE-Blöcke mit dem Namen *Root* bezeichnet sind, wird zur Laufzeit automatisch der passende DEFINE-Block für den Typ des Modellelements gewählt.

```
1  «DEFINE Root FOR java6::Classifier»«ENDDEFINE»
2
3  «DEFINE Root FOR java6::Class»
4  «EXPAND Annotation FOREACH this.annotations»
5  public «this.getAbstract()» class «this.name»«EXPAND Inheritance»«EXPAND Implements»{
6    «EXPAND Variable FOREACH this.variables»
7    «EXPAND Method FOREACH this.methods»
8  }
9  «ENDDEFINE»
10
11 «DEFINE Root FOR java6::Interface»
12 public «this.getAbstract()» interface «this.name»«EXPAND Inheritance»{
13   «EXPAND Variable FOREACH this.variables»
14   «EXPAND InterfaceMethod FOREACH this.methods»
15 }
16 «ENDDEFINE»
```

Listing 77: Ausschnitt aus dem Template *templates::java6::Classifier.xpt*

Spezielle Methoden wie z. B. solche für Datenzugriffsklassen werden inkl. der Methodenrümpfe durch separate DEFINE-Blöcke generiert, die in Framework-spezifischen Templatepaketen bzw. deren Templates definiert sind. So ist beispielsweise im Template *templates::hibernate3::DataAccessObject.xpt* der DEFINE-Block *Method* für die EClass *DAOList* aus dem Metamodell *Hibernate3* enthalten, der nach der Generierung der Methodensignatur ergänzend den Methodenrumpf erzeugt (vgl. Listing 78). Dieser umfasst eine Query, welche auf der Datenbanksitzung ausgeführt wird und deren Ergebnis von der Methode zurückgegeben wird. Wie im Beispiel finden generell bei der Generierung von Methodenrümpfen die zusätzlichen EAttributes und EReferences der Spezialisierungen von *Method* Verwendung.

```
1  «DEFINE Method FOR hibernate3::designpatterns::dao::daomethods::DAOList»
2    «this.getVisibility()» «this.getQualifiedTypeName()» «this.name»(«EXPAND templates::java6
           ::Classifier::Parameter FOREACH this.getParameterList() SEPARATOR ","»){
3      String queryString = "FROM «this.managedPersistentPojo.name»";
4      Query q = this.«this.getCurrentSessionMethod.name»().createQuery(queryString);
5      return («this.getQualifiedTypeName()») q.list();
6    }
7  «ENDDEFINE»
```

Listing 78: Ausschnitt aus dem Template *templates::hibernate3::DataAccessObject.xpt*

Deskriptive textuelle Dokumente wie z. B. JSF-Facelets lassen sich vergleichsweise einfach durch Templates generiert, da die klare Struktur der Dokumente meist wenig Interdependenzen der Dokumenteninhalte untereinander induziert und lediglich vordefinierte Dokumenteninhalte zu parametrisieren sind. So bildet der DEFINE-Block *Root* im Template *templates::jsf1::facelets::facelettypes::EditFacelet.xpt* für die EClass *EditFacelet* diese EClass auf ein JSF-Formular ab, das über die Expandierung des DEFINE-Blocks *FaceletVariable* z. B. mit Eingabefeldern für *FaceletScalarVariables* aus dem Modell angereichert

wird (vgl. Listing 79). Bei der Ausführung der DEFINE-Blöcke können die Platzhalter ohne komplexere Generatorlogik ausgeprägt werden, da die Werte bereits durch die vorgelagerte M2M-Transformationsdefiniton vorberechnet in den EAttributes und EReferences des Modellelements gespeichert sind, wie z. B. im Fall der EReference *FaceletVariable::managedPersistentPojo_referenceVariable*.

```
1  «DEFINE Root FOR jsf1::facelets::facelettypes::EditFacelet»
2    <h:form>
3      <h:panelGrid columns="2" border="1">
4      «EXPAND FaceletVariable FOREACH this.faceletVariables»
5      </h:panelGrid>
6      <h:commandButton action="..." value="Save"/>
7    </h:form>
8  «ENDDEFINE»
9
10 «DEFINE FaceletVariable FOR jsf1::facelets::FaceletScalarVariable»
11   <h:outputText value="«this.name»:" />
12   <h:inputText id="«this.managedPersistentPojo_referenceVariable.name»" value="..." «IF this
           .isReadonly»readonly="true" «ENDIF»/>
13 «ENDDEFINE»
```

Listing 79: Ausschnitt aus dem Template *templates::jsf1::facelets::facelettypes::EditFacelet.xpt*

5 Templategenerator *Metagen*

5.1 Problemstellung

Das Eclipse Modeling Framework bietet die Möglichkeit, XML-Schemadefinitionen mittels einer Meta-Transformationsdefinition in Ecore-basierte Metamodelle zu überführen [SBPM08, S. 197-259][04.04b], mit denen XML-Dokumente in Modellen abgebildet werden können. Beispiele für solche automatisch erzeugten Metamodelle im Rahmen des WASL-Generatorframeworks sind die Metamodelle *XHTML1-Strict_XSD_Adapted* und *Hibernate30Config_XSD_Adapted*.

Entsprechend zu der automatisierten Erstellung der Metamodelle bietet sich auch bei der Entwicklung der M2T-Transformationsdefinitionen für die Codegeneratoren prinzipiell Potential für eine automatisierte Erzeugung. Eine zu generierende Transformationsdefinition soll dabei Abbilder von XML-Dokumenten im Modell in wohlgeformte und valide (vgl. [W3C08]) textuelle textuelle XML-Dokumente überführen. Sowohl die Anforderung der Wohlgeformtheit als auch die der Validität kann bei der Generierung von M2T-Transformationsdefinitionen erfüllt werden, da die Kriterien für Wohlgeformtheit als generelle syntaktische Anforderung an XML-Dokumente ohne Bezug zu bestimmten Metamodellen bekannt sind, und die Kriterien für Validität in den XML-Schemadefinitionen bzw. den entsprechenden Metamodellen ausgedrückt sind.

Eine Generierung der Transformationsdefinitionen ist bei automatisch generierten Metamodellen im Vergleich zu manuell entworfenen besonders sinnvoll, da die zugrunde liegenden XML-Schemadefinitionen z. B. im Fall von XHTML vergleichsweise umfangreich sind und somit entsprechend komplexe M2T-Transformationsdefinitionen induzieren. Darüber hinaus liegt es nahe, dass auch die Entwicklung von Transformationsdefinitionen von den Vorteilen der modellgetriebenen Softwareentwicklung profitieren kann. Insofern existieren verschiedene Ansätze für die Generierung von Transformationsdefinitionen bzw. Generatoren.

5.2 Verwandte Ansätze

Diverse Ansätze zur automatisierten Generierung von M2M-Transformationsdefinitionen bieten Werkzeuge für das Matching von Metamodellen bzw. Metamodellelementen, die teilautomatisiert abstrakte Mappings zwischen Elementen verschiedener Metamodelle anlegen und aus diesen Transformationsdefinitionen ableiten. Implementierungen dieser Art, die über Heuristiken zur teilautomatisierten Erkennung semantischer Ähnlichkeiten zwischen Metamodellen verfügen, sind das Matching-Tool *MT4MDE/SAMT4MDE* [LHA06, LHBJ05, SLCA09], sowie diverse weitere Matching-Werkzeuge [KKK+07a, DV07,

ADMR05].

Einen alternativen Lösungsweg bietet das Generatorframework *Marius* [RKR+06], das auf die Modellierung und Generierung von Domain Specific Model Transformation Languages (DSMTL) für M2M-Transformationsdefinitionen abzielt und somit als Meta-Generatorframework einzustufen ist. Die generierten domänenspezifischen Transformationssprachen bieten abstrakte sprachliche Ausdrucksmittel zur Referenzierung von Konzepten aus den domänenspezifischen Metamodellen, wie z. B. sprachliche Schlüsselwörter zur Selektion bestimmter Sequenzen von Aktivitäten in Prozessmodellen. Analog zu Transformationsdefinitionen wird die Implementierung der domänenspezifischen Ausführungssemantik der Transformationssprache in Form von Java-Code mit Hilfe von JET-Templates generiert, die ihrerseits teilautomatisiert aus der Syntax der DSMTL abgeleitet werden.

Die modellgetriebene Software-Entwicklungsumgebung *HyperSenses*[49] nennt als Kernfunktionalität die Generierung von Generatoren [Del11a, Del11c]. Dem liegt zugrunde, dass als Code-Patterns bezeichnete Templates i.V.m. Metamodellen für die Erzeugung ausführbarer Codegeneratoren verwendet werden. Die Formulierung der Code-Patterns wird vom Entwicklungswerkzeug unterstützt, ist aber manuell durchzuführen (vgl. [Del11b]). Insofern ist Ähnlichkeit zu Xpand-basierten Templates gegeben, die ebenfalls i.V.m. Metamodellen Codegeneratoren konstituieren.

Den Ansätzen ist gemein, dass der Schwerpunkt auf M2M-Transformationen liegt, deren Definitionen teilautomatisiert generiert werden. Letzteres ist darin begründet, dass die teilgenerierten Transformationsdefinitionen über rein syntaktische Details hinaus semantisch eine erhöhte Komplexität aufweisen, sodass ein manuelles Eingreifen erforderlich ist.

5.3 Einordnung

Im Folgenden wird der Generator *Metagen* vorgestellt, welcher die vollautomatisierte Erstellung von M2T-Transformationsdefinitionen in Form von Xpand-Templates für die Generierung von XML-Dokumenten unterstützt. Die Voraussetzung für den Einsatz von *Metagen* ist, dass das jeweils zugrunde liegende Metamodell mittels EMF aus einer XML-Schemadefinition generiert ist. Die M2T-Transformationsdefinition wird mit *Metagen* anschließend generiert, indem für das Metamodell ein Xpand-Template angelegt wird (vgl. Abbildung 57), das für jede EClass aus dem Metamodell jeweils einen DEFINE-Block enthält. Jeder generierte Block steuert für die zugeordnete EClass bzw. für den XML-Elementtyp die Serialisierung der entsprechenden Modellelemente in das XML-Dokument. Da das Metamodell und somit auch die Transformationsdefinition von der XML-Sche-

[49] `http://www.d-s-t-g.com/`

madefinition abgeleitet sind, kann bei der Transformation ein wohlgeformtes und valides XML-Dokument erzeugt werden.

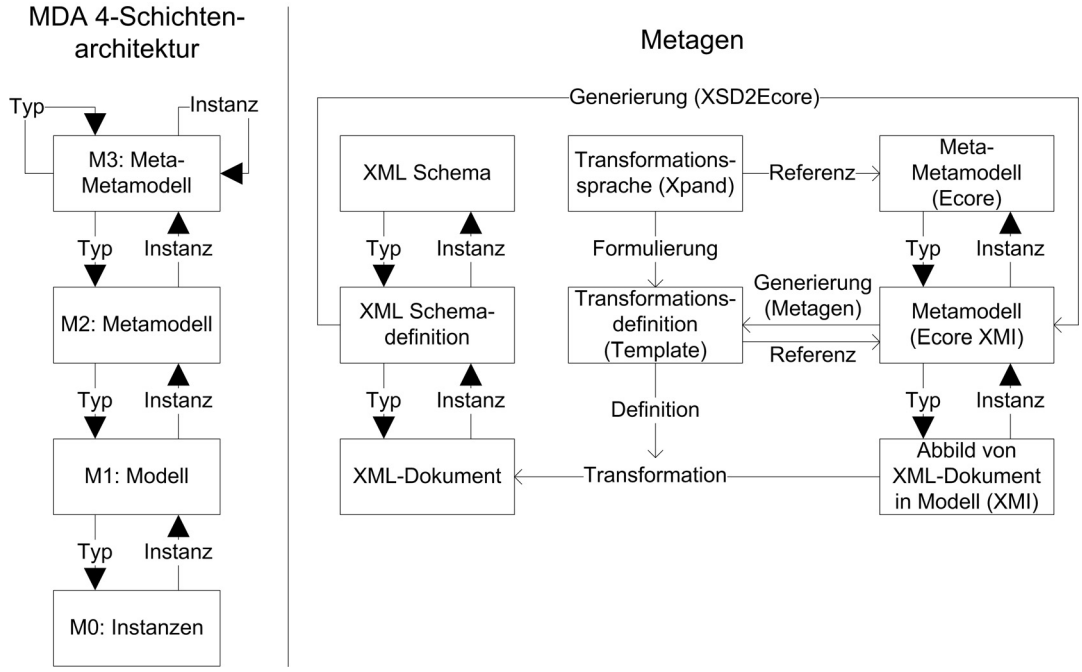

Abbildung 57: Einordnung von *Metagen* in die Vier-Schichtenarchitektur der MDA (vgl. Abbildung 11)

Metagen ist in der Vier-Schichtenarchitektur der OMG der Schicht M3 zuzuordnen und in Java implementiert. Analog sind das durch EMF generierte Metamodell und das Template als Generat von *Metagen* der Schicht M2 zuzuordnen.

5.4 Struktur der generierten Templates

Die grundsätzliche Struktur jedes generierten Templates setzt sich zusammen aus einem Header zur optionalen Einbindung von Xtend-Dateien in einem geschützten Bereich, sowie einem Rumpf aus DEFINE-Blöcken mit dem Namen *Element* für die verschiedenen EClasses des Metamodells. Die DEFINE-Blöcke sind kategorisiert in solche für speziell behandelte EClasses, in solche, die manuell für *RefBased*-EClasses in einem geschützten Bereich dem Metamodell hinzugefügt sind und in solche für die restlichen EClasses, die konventionell aus dem XML-Schema abgeleitet sind.

So enthält beispielsweise das generierte Template *Hibernate30Config_XSD_Adapted.xpt* in Entsprechung zu dem gleichnamigen Metamodell (vgl. Kapitel 4.6.3.3) DEFINE-Blöcke für die speziellen EClasses *DocumentRoot* und *TextWrapper*, einen DEFINE-Block für die manuell dem Metamodell hinzugefügte EClass *MappingTypeRefBased* und diverse DEFI-

NE-Blöcke für die konventionell hinzugefügten EClasses wie z. B. *HibernateConfiguration-Type* (vgl. Listing 80). Innerhalb der DEFINE-Blöcke werden Xpand-Ausdrücke generiert, die der Erzeugung von XML-Attributen und XML-Subelementen dienen, wie z. B. im Fall der EReference *sessionFactory*.

```
1  //PROTECTED_REGION_START:EXT
2  //PROTECTED_REGION_END:EXT
3
4  «DEFINE Element FOR hbconf30::DocumentRoot»
5    <?xml version='1.0' encoding='UTF-8'?>
6    <!DOCTYPE hibernate-configuration PUBLIC ...>
7    ...
8    «IF this.hibernateConfiguration != null»
9      «EXPAND Element("hibernate-configuration") FOR this.hibernateConfiguration»
10   «ENDIF»
11 «ENDDEFINE»
12
13 «DEFINE Element(String xmlTag) FOR hbconf30::TextWrapper»«this.text»«ENDDEFINE»
14
15 //PROTECTED_REGION_START:DEFINES
16 «DEFINE Element(String xmlTag) FOR hbconf30::MappingTypeRefBased»...«ENDDEFINE»
17 //PROTECTED_REGION_END:DEFINES
18
19 «DEFINE Element(String xmlTag) FOR hbconf30::HibernateConfigurationType»
20   <«xmlTag»>
21     «IF this.sessionFactory != null»
22       «EXPAND Element("session-factory") FOR this.sessionFactory»
23     «ENDIF»
24     ...
25   </«xmlTag»>
26 «ENDDEFINE»
27 ...
```

Listing 80: Ausschnitt aus dem generierten Template *Hibernate30Config_XSD_Adapted.xpt* (M2)

Die Struktur der Templates spiegelt sich in der zentralen Generatormethode *run* von *Metagen* wider, mit der die Abfolge des Headers und der DEFINE-Blöcke gesteuert wird (vgl. Listing 81). Die Methode generiert initial durch Aufruf der Methode *writeXpand-Header* den Header inkl. des geschützten Bereichs. Anschließend werden für die speziellen EClasses *DocumentRoot* und *TextWrapper* im Metamodell durch Aufruf der Methoden *writeXpandDocumentRootDefineBlock* und *writeXpandTextWrapperDefineBlock* die entsprechenden DEFINE-Blöcke erzeugt. Die DEFINE-Blöcke für *RefBased*-EClasses werden mit der Methode *getXpandRefBasedDefineBlock* aus denjenigen EClasses des Metamodells erzeugt, deren Name das Suffix *RefBased* aufweist, und in einen geschützten Bereich im Template mit dem Schlüssel *REFBASED* geschrieben. Abschließend werden aus den restlichen EClasses des Metamodells mit der Methode *writeXpandDefaultDefineBlock* die konventionellen DEFINE-Blöcke generiert.

5.5 Generierung von DEFINE-Blöcken für XML-Elemente

Die Generatormethoden für konventionelle und *RefBased*-EClasses rufen die Methode *getXpandDefaultDefineBlock* auf. Diese generiert für eine EClass einen DEFINE-Block,

```
1  public class XpandTemplateGenerator extends metagen.TransformationDefinitionGenerator{
2    public synchronized void run(
3      // Header
4      writeXpandHeader();
5
6      // Special EClasses
7      EClass documentRoot = EClassHelper.getDocumentRootEClass(eClasses);
8      writeXpandDocumentRootDefineBlock(documentRoot);
9
10     EClass textWrapper = EClassHelper.getTextWrapperEClass(eClasses);
11     writeXpandTextWrapperDefineBlock(textWrapper);
12
13     // RefBased EClasses
14     StringBuilder refBasedDefineBlocks = new StringBuilder();
15     Collection<EClass> refBasedEClasses = EClassHelper.getRefBasedEClasses(eClasses);
16     for (EClass eClass : refBasedEClasses)
17       refBasedDefineBlocks.append(getXpandRefBasedDefineBlock(eClass));
18     this.writeProtectedRegion("REFBASED", refBasedDefineBlocks.toString());
19
20     // Standard EClasses
21     for (EClass eClass : EClassHelper.getStandardEClasses(eClasses))
22       writeXpandDefaultDefineBlock(eClass);
23     ...
24   }
25   ...
26 }
```

Listing 81: Orchestrierung des *XpandTemplateGenerator* (M3)

der zur Laufzeit des Codegenerators unter dem String-Parameter *xmlTag* einen XML-Tag entgegennimmt und diesen anschließend in das XML-Dokument schreibt (vgl. Listings 82 und 80, Zeile 19 ff.). Durch die Methodenaufrufe *getXmlAttributes* und *getExpandBlocksForEClass* werden zudem in jeden DEFINE-Block Xpand-Ausdrücke eingefügt, mit denen während der Codegenerierung die Attribute und Kindelemente des XML-Elements in das XML-Dokument geschrieben werden.

```
1  protected String getXpandDefaultDefineBlock(EClass eClass) throws IOException {
2    StringBuilder sb = new StringBuilder();
3    sb.append("«DEFINE␣Element(String␣xmlTag)␣FOR␣" + getXpandQualifiedName(eClass) + "»");
4    sb.append("<«xmlTag»");
5    sb.append(getXmlAttributes(xmlTag, eClass));
6    sb.append(">");
7    sb.append(getExpandBlocksForEClass(eClass));
8    sb.append("</«xmlTag»>");
9    sb.append("«ENDDEFINE»");
10   return sb.toString();
11 }
```

Listing 82: Vereinfachte Darstellung der Methode *getXpandDefaultDefineBlock* (M3)

Der XML-Tag des XML-Elements eines DEFINE-Blocks wird also nicht statisch durch den DEFINE-Block festgelegt, sondern zur Laufzeit dynamisch über den Parameter *xmlTag* vom aufrufenden EXPAND-Block übergeben, wie z. B. im Fall des DEFINE-Blocks für die EClass *HibernateConfigurationType* aus dem Metamodell *Hibernate30Config_XSD_Adapted* (vgl. Listing 83).

Dieses dynamische Verfahren zur Festlegung des XML-Tags ist notwendig, da der XML-Tag des XML-Elements nicht dem Namen der EClass entspricht, sondern dem der ERe-

```
1 «DEFINE Element(String xmlTag) FOR hibernateconfiguration30::HibernateConfigurationType»
2 <«xmlTag»>
3   «IF this.sessionFactory != null»
4     «EXPAND Element("session-factory") FOR this.sessionFactory»
5   «ENDIF»
6   ...
7 </«xmlTag»>
8 «ENDDEFINE»
```

Listing 83: Generierter DEFINE-Block für die EClass *HibernateConfigurationType* (M2)

ference zu der EClass. Da ein DEFINE-Block aber einer EClass zugeordnet ist, und aus Sicht des DEFINE-Blocks nicht bekannt ist, über welche der vielen möglichen EReferences er zur Laufzeit des Codegenerators expandiert wird, muss diese Information deshalb per Parameter *xmlTag* übergeben werden. Die Information wird mit dem Parameter als *String*, und nicht etwa als Referenz übergeben, weil die Übergabe einer Referenz als Ausdruck von Meta-Programmierung in Xpand nicht möglich ist.

Ursächlich für die Verortung der XML-Tags im Metamodell in Form von Namen der EReferences, und nicht etwa als Namen von EClasses ist, dass erstens in der XML-Schemadefinition jeder XML-Tag als Name einer XML-Element-Deklaration erfasst wird, und zweitens jede XML-Element-Deklaration, die einen ComplexType besitzt, in eine EReference transfomiert wird [SBPM08, S. 231-245]). Dabei wird der ComplexType in eine EClass überführt [SBPM08, S. 211-222] und als Ziel der EReference festgelegt.

Die EXPAND-Blöcke innerhalb eines DEFINE-Blocks (vgl. Listing 83) werden mittels der Methode *getExpandBlocksForEClass* für sämtliche EReferences einer EClass angelegt (vgl. Listing 84). Abhängig von der Multiplizität der EReference wird der EXPAND-Block entweder als einzelner Aufruf auf einer einwertigen EReference erzeugt oder mittels *FOREACH* als wiederholter Aufruf auf einer mehrwertigen EReference. Im einwertigen Fall muss durch einen umgebenenden IF-Block zudem geprüft werden, ob die EReference mit einem Wert belegt ist. Bei beiden Formen des Aufrufs wird der vorab beschriebene obligatorische Parameter *xmlTag* des aufzurufenden DEFINE-Blocks mit dem XML-Tag belegt, der mit der Methode *toXmlTag* aus dem Namen der EReference abgeleitet wird. Das Resultat der Methode *getExpandBlocksForEClass* ist eine Sequenz von EXPAND-Blöcken für die verschiedenen EReferences der EClass, mit denen bei der Codegenerierung im XML-Dokument in derselben Reihenfolge die XML-Elemente aus dem Modell übertragen werden.

Das sukzessive Expandieren von DEFINE-Blöcken für die verschiedenen EReferences einer EClass ist eine geeignete Lösung, falls in der zugrunde liegenden XML-Schemadefinition in den XML-Elementdeklarationen vorgegeben ist, dass z. B. per Elementgruppe (vgl. [W3C04b, Kapitel 3.8.2]) die Kindelemente jedes XML-Elements in einer bestimmten Reihenfolge vorliegen müssen, um die Reihenfolge der EXPAND-Blöcke in derselben Reihenfolge im DEFINE-Block anzulegen.

```
protected String getExpandBlocksForEClass(EClass eClass, int namingMethod) {
  StringBuilder sb = new StringBuilder();
  for (EReference reference : eClass.getEAllContainments()) {
    if (!reference.isMany()) {
      //«IF this.sessionFactory != null»
      sb.append("«IF␣this." + reference.getName() + "␣!=␣null»");
      //«EXPAND Element("session-factory") FOR this.sessionFactory»
      sb.append("«EXPAND␣Element(\"" + toXmlTag(reference) + "\")␣FOR␣this." + reference.
              getName() + "»");
      //«ENDIF»
      sb.append("«ENDIF»");
    } else {
      sb.append("«EXPAND␣Element(\"" + toXmlTag(reference) + "\")␣FOREACH␣this." + reference
              .getName()  + "»");
    }
  }
  return sb.toString();
}
```

Listing 84: Vereinfachte Darstellug der Methode *getExpandBlocksForEClass* (M3)

Die festgelegte Reihenfolge liegt im Rahmen von XML-Konfigurationsdateien üblicherweise vor, ist aber z. B. im Fall der XML-Schemadefinition für XHTML nicht erfüllt. Bei Letzterer weisen einige der ComplexTypes das Attribut *mixed* (vgl. [W3C04b, Kapitel 3.4.2]) mit dem Wert *true* auf, um zur Beschreibung von Webseiten eine Mischung von Text und XML-Elementen in beliebiger Reihenfolge zu dessen Auszeichnung zu ermöglichen. Ein ComplexType dieser Art wird zwar von EMF im Ecore-basierten Metamodell in eine EClass überführt, die ein EAttribute *mixed* vom Typ FeatureMap besitzt, in der die verschiedenen XML-Elemente in Form von EFeatureMapEntries geordnet enthalten sind [SBPM08, S. 215-217]. Problematisch ist aber, dass in Xpand auf die EFeatureMapEntries bzw. die darin enthaltenen XML-Elemente lediglich über den Xpand-Ausdruck

```
«EXPAND Element FOREACH this.eContents»
```

zugegriffen werden kann, der zwar die Reihenfolge erhält, aber keinen expliziten Zugriff auf den XML-Tag zulässt. Z. B. im Fall der XML-Schemadefinition von XHTML ist dies aber kein Hindernis, da die XML-Tags alternativ zu den XML-Elementdeklarationen bzw. EReferences auch aus den ComplexTypes bzw. EClasses aufgrund einer passenden Benennung abgeleitet werden können.

```
«FOREACH this.mixed AS entry»
  <«entry.getEStructuralFeature().getName()»>
    «EXPAND Element FOR entry.getValue()»
  </«entry.getEStructuralFeature().getName()»>
«ENDFOREACH»...
```

Listing 85: FOREACH-Schleife über eine FeatureMap

Die optimale Lösung wäre gleichwohl der direkte Zugriff auf die FeatureMap, um in Form eines FOREACH-Blocks über die EFeatureMapEntries zu iterieren und die EXPAND-Aufrufe im Kontext des korrekten Tags auszuführen (vgl. Listing 85). Da dies aber

z. B. von Xpand nicht unterstützt wird, ist abhängig von den Fähigkeiten der zu verwendenden Templatesprache aus den drei Alternativen die passende auszuwählen. Im Kontext des WASL-Generatorframeworks werden z. B. sämtliche der mit *Metagen* erzeugten Templates über den ersten, EReference-basierten Weg erstellt, bis auf das Template für das Metamodell *XHTML1-Strict_XSD_Adapted.ecore*, das über den zweiten, EClass-basierten Weg angelegt wird.

5.6 Generierung von DEFINE-Blöcken für XML-Attribute

Die Attribute von XML-Elementen werden durch das Xpand-Template in das XML-Dokument geschrieben, indem konform zum XML-Standard beim Anlegen eines Start-Tags die Attribute mit ihren Werten dem XML-Element hinzugefügt werden. Im Abbild des XML-Dokuments im Modell werden die XML-Attribute in den meisten Fällen als EAttributes der EClasses repräsentiert [SBPM08, S. 222-231] und vom DEFINE-Block für die EClass sukzessive in das XML-Dokument übertragen. So wird beispielsweise in dem generierten Template *Hibernate30Config_XSD_Adapted.xpt* durch den DEFINE-Block *Element* für die EClass *SessionFactoryType* das Attribut *name* im Fall einer Belegung mit einem Wert im XML-Dokument angelegt und ausgeprägt (vgl. Listing 86).

```
1 «DEFINE Element(String xmlTag) FOR hibernateconfiguration30::SessionFactoryType»
2   <«xmlTag»«IF this.name != null» name="«this.name»"«ENDIF»>...</«xmlTag»>
3 «ENDDEFINE»
```

Listing 86: DEFINE-Block für die EClass *SessionFactoryType* (M2)

Die entsprechenden Xpand-Anweisungen werden in den DEFINE-Blöcken gesamtheitlich durch Aufruf der Methode *getXmlAttributes* angelegt (vgl. Listing 82), die für jedes EAttribute die Anweisung durch Aufruf der Methode *getXmlAttribute* erhält. Letztere erzeugt die Xpand-Anweisung durch Einfügung des EAttribute-Namens in das zugrunde liegende Xpand-Muster (vgl. Listing 87).

```
1 protected String getXmlAttribute(String xmlTag, EAttributeImpl eAttribute) {
2   if (eAttribute.getEType() != null && eAttribute.getEType().getName() != null) {
3     //«IF this.name != null» name="«this.name»"«ENDIF»
4     return "«IF this." + eAttribute.getName() + " != null» "
5       + eAttribute.getName() + "=\"«this." + eAttribute.getName() + "»\""
6       + "«ENDIF»";
7   }
8   return "";
9 }
```

Listing 87: Vereinfachte Darstellung der Methode *getXmlAttribute* (M3)

6 TaskUI-Generatorframework

6.1 Einführung

Neben der reinen Datenhaltung ist die Abbildung und Unterstützung betrieblicher Geschäftsprozesse ein weiteres Aufgabengebiet moderner betrieblicher Anwendungssysteme [HN08]. Entsprechend existieren für die Modellierung von Geschäftsprozessen diverse Modellierungstechniken, zu denen als prominenteste das ARIS-Konzept [Sch01] auf Grundlage ereignisgesteuerter Prozessketten (EPK) [KNS92], die Business Process Execution Language (BPEL) [OAS07] und die Business Process Model and Notation (BPMN) [Obj11a] zählen. Für den Verwendungszweck der Prozessautomatisierung können Prozessmodelle zunehmend softwaregestützt für die Ausführung der spezifizierten Prozesse operativ eingesetzt werden. Dies betrifft sowohl insgesamt die Steuerung des Prozessflusses, als auch die Ausführung einzelner automatisierbarer Prozesselemente. Auf diese Weise bietet sich eine Möglichkeit, die Lücke zwischen den fachlichen Prozessbeschreibungen und deren technischen Implementierungen durch Anwendungssysteme zu schließen.

Eine Kernanforderung für die softwaregestützte Ausführung von Prozessmodellen ist, dass diese eine formale Ausführungssemantik aufweisen, welche durch die Ausführungsumgebung interpretiert und verarbeitet werden kann. Die BPMN 2.0 erfüllt mittels ihrer standardisierten Ausführungssemantik diese Anforderung und ermöglicht prinzipiell, sowohl fachliche als auch technische Prozessmodelle zu erstellen und diese mit Hilfe standardkonformer BPMN-Engines zu instanziieren (vgl. [FRH10]). Zu diesem Zweck existiert eine Vielzahl an BPMN-kompatiblen Plattformen[50], von denen einige bereits vor der Existenz von BPMN 2.0 entwickelt wurden und aufgrund ihres historischen Ursprungs weitere Prozessmodellierungstechniken wie z. B. *jPDL* oder *BPEL* unterstützen.

Die BPMN 2.0 sieht für die Mitwirkung von Menschen im Prozessfluss die beiden Prozesselementtypen *ManualTask* und *UserTask* vor [Obj11a, S. 165]. Der Typ *ManualTask* spezifiziert aus Sicht der BPMN-Ausführungsumgebung keine Ausführungssemantik, und wird entsprechend nicht durch diese verwaltet. Im Gegensatz dazu repräsentiert der Typ *UserTask* eine softwaregestützte Aktivität eines menschlichen Akteurs, der mit dem Prozess bzw. der Ausführungsumgebung üblicherweise über eine graphische Benutzerschnittstelle interagiert. Dazu deckt die Kernfunktionalität der Benutzerschnittstelle eine Darstellung von Arbeitsaufträgen in Listenform (*Tasklist*) sowie Detailansichten zu den einzelnen Aufträgen (*Task GUIs*) ab. Zu dem Zweck der Verwaltung durch die BPMN-Ausführungsumgebung verfügt der Typ *UserTask* über Attribute zur Modellierung fachlicher Aspekte, z. B. zur Zuordnung von Arbeitsaufträgen zu Rollen oder deren Priorisierung, sowie zur

[50] Vgl. `http://www.bpmn.org/`

Spezifikation technischer Details, beispielsweise zur Anbindung von Webservices [Obj11a, S. 167].

Die Implementierung eines *UserTask* ist hingegen nicht durch den BPMN-Standard spezifiziert und unterliegt den Plattformspezifika der konkreten BPMN-Plattform. So sind im BPMN-Modell die Attribute *implementation* zur Wahl einer Technik für die Implementierung z. B. per *WS-HumanTask*, sowie *renderings* zur Spezifikation von Attributen für die Darstellung der Benutzerschnittstelle in Abhängigkeit von der Plattform bzw. deren Fähigkeiten auszuprägen [Obj11a, S. 166 f.]. In der praktischen Umsetzung führt die Offenheit des BPMN-Standards in diesem Punkt zu einer heterogenen Vielzahl von Lösungsansätzen, sowohl in Bezug auf die diversen Plattformen als auch auf deren Versionen. So findet beispielsweise im Fall der *Activiti BPM Platform* für das Rendering HTML i.V.m. der *Java Unified Expression Language* (JUEL) Verwendung[51], für *jBPM* die Template-Engine *Freemarker*[52] und für die *Oracle SOA Suite* die Web-Technologie *JavaServer Pages* (JSP)[53]. Die mangelnde Standardisierung des Renderings führt zu enginespezifischen BPMN-Prozessdefinitionen und Implementierungen, aus denen Abhängigkeiten von den Herstellern der BPMN-Engines (Vendor-Lock-in-Effekt) und erhöhte Kosten eines Plattformwechsels resultieren.

Aufgrund der klar definierten BPMN-Zielplattformen und der textuellen Natur der Artefakte bietet sich die modellgetriebene Softwareentwicklung als Option, um Benutzerschnittstellen plattformunabhängig zu spezifizieren und anschließend in plattformspezifische textuelle Repräsentationen zu übersetzen. Da die *UserTasks* in den BPMN-Modellen potentiell Schnittstelleninformationen aufweisen, bietet sich zudem die Möglichkeit, auch die plattformunabhängigen Modelle der Benutzerschnittstellen teilautomatisiert zu erstellen.

Die meisten Ansätze zur Generierung von Benutzerschnittstellen für Geschäftsprozesse sind der modellgetriebenen Softwareentwicklung und im Speziellen partiell dem Forschungsfeld des MDWE zuzuordnen.

So existiert im Kontext von WebML für das Softwareentwicklungswerkzeug *WebRatio* ein Generator, der aus BPMN-Modellen bzw. deren *UserTasks* und *ServiceTasks* intermediäre WebML-Modelle ableitet, und aus diesen eigenständige, generatorspezifische Java EE-Applikationen ohne Bezug zu etablierten BPMN-Engines generiert [BB10, Web11, BCFM06].

Für OO-H und UWE existieren Ansätze zur Erstellung konzeptioneller Prozessmodelle in Form von UML-Aktivitätsdiagrammen, die im Fall von OO-H durch Navigationsmo-

[51] `http://www.activiti.org/userguide/index.html#forms`

[52] `http://docs.jboss.org/jbpm/v5.2/userguide/ch11.html#d0e3537`

[53] `http://docs.oracle.com/cd/E25054_01/dev.1111/e10224/bp_designtf.htm#CACHGEFD`

delle (NADs) und im Fall von UWE durch Navigations- und verfeinerte Prozessmodelle ergänzt werden, um sie als Teil von Webapplikationen mit Benutzerschnittstellen zu versehen [KKCM04, KKCM03]. Auf eine Einbindung der BPMN oder anderer fachlicher Prozessmodellierungssprachen sowie die Erstellung von PSMs wird bei der Modellierung aber verzichtet.

Eine Erweiterung für OOWS sieht die Transformation von BPMN-Modellen in plattformunabhängige OOWS-Navigationsmodelle vor, aus denen eigenständige Webapplikationen zur Interaktion mit BPEL-Workflowspezifikationen generiert werden, welche die BPMN-Prozesse in ausführbarer Form repräsentieren [TP06].

Ein Entwicklungswerkzeug von Reichwald et al. [RDB+08] generiert aus angereicherten BPEL-Workflowspezifikationen plattformspezifische JSP-Templates für die Darstellung von Benutzerschnittstellen, die zur Laufzeit mit dem ausgeführten BPEL-Workflow interagieren. Ein weiterer Ansatz [FBNG07] konfiguriert eine obligatorische Prozessausführungsumgebung mittels angereicherter XPDL-Prozessmodelle und stellt in dieser Benutzerdialoge mit XHTML, CSS und XForms dar.

Den Ansätzen ist gemein, dass eine minimal-invasive Injektion generierter Benutzerschnittstellen in die plattformspezifischen Rendering-Umgebungen der diversen BPMN-Engines nicht angestrebt wird. Stattdessen wird entweder auf eine vergleichsweise komplexe Generierung von Webapplikationen als Laufzeitumgebungen für die Benutzerschnittstellen abgezielt, oder die BPMN-Engine bzw. deren Auswahl als Teil des gestaltbaren Lösungsraums aufgefasst. Diese Herangehensweise ist aber in der Praxis wenig pragmatisch, da bei der Entscheidung für eine BPMN-Plattform i.V.m. mit einem Generator die präferierte BPMN-Plattform tendenziell die Auswahl an kompatiblen Generatoren determiniert, nicht aber die Plattformabdeckung eines präferierten Generators die Auswahl an BPMN-Plattformen. Insofern ist es sinnvoll, die BPMN-Entwicklungs- und -Ausführungsumgebung nach Möglichkeit vollständig agnostisch bzgl. des Generators zu belassen, um die Abhängigkeit vom Generator zu minimieren. Eine weitere Restriktion der genannten Ansätze ist, dass eine plattformspezifische Repräsentation von Benutzerschnittstellen in Form strukturähnlicher DSLs wie z. B. mit *WASL PSM* nicht unterstützt wird, sodass die Einflussnahme auf plattformspezifische Details des Generats Modifikationen am Codegenerator bedingt.

Im Folgenden wird deshalb mit dem TaskUI-Generatorframework ein prototypisches modellgetriebenes Softwareentwicklungswerkzeug vorgestellt, das die teilautomatisierte Generierung prozessbasierter Webapplikationen in Form von formularbasierten Web-Benutzerschnittstellen für *UserTasks* aus BPMN-Modellen unterstützt. Es verwendet Metamodelle aus *WASL PSM* sowie den Codegenerator des WASL-Generatorframeworks für *Java EE* wieder, und ergänzt die Metamodelle um die domänenspezifische und plattformunab-

hängige Modellierungssprache *TaskUI Generic*. Entsprechend basiert das TaskUI-Generatorframework ebenfalls auf Techniken, Standards und Prinzipien der MDA und illustriert die Flexibilität des plattformspezifischen Teils des WASL-Generatorframeworks.

Im Folgenden werden zu diesem Zweck initial aufgabenrelevante Konzepte der BPMN erläutert, anschließend die Architektur des TaskUI-Generatorframeworks beschrieben, das Metamodell der Modellierungssprache *TaskUI Generic* vorgestellt und abschließend Transformationsdefinitionen zur automatisierten Generierung der Web-Benutzerschnittstellen vorgestellt.

6.2 Business Process Model and Notation

Die Business Process Model and Notation (BPMN) ist eine Modellierungssprache zur fachlichen Modellierung von Geschäftsprozessen, die erstmals im Jahr 2004 veröffentlicht wurde [Whi04] und seit dem Jahr 2006 im Rahmen der OMG gepflegt wird [Obj06a]. Ein BPMN-Prozessmodell hat die Form eines gerichteten Graphen, dessen Knoten im Kern Flussobjekte (*Flow Nodes*) sind, die durch Kanten in Form von Sequenzflüssen (*Sequence Flows*) verbunden werden (vgl. [FRH10, S. 21], [Obj11a, S. 28 ff.]). Die *Flow Nodes* sind untergliedert in Aktivitäten (*Activities*) für Tätigkeiten, Ereignisse (*Events*) und *Gateways* zur Steuerung des Kontrollflusses. Sie können in *Pools* oder deren *Lanes* enthalten sein, mit denen Prozessteilnehmer oder deren Rollen repräsentiert werden.

Abhängig vom Modellierungszweck können BPMN-Modelle mit fachlichem oder technischem Fokus spezifiziert werden (vgl. [FRH10, S. 14-17]). Letzteres ist nötig, falls die Ausführung der Prozessspezifikation mit einer BPMN-Engine oder die Generierung eigenständiger Implementierungen aus der Prozessspezifikation intendiert ist. Die Verfeinerung betrifft beispielsweise die Definition von Datenstrukturen und deren Typisierung, die Anbindung von Webservices und die Formulierung von Ausdrücken für Konditionen von Gateways (vgl. [FRH10, S. 199-254]).

Für die Generierung von Benutzerschnittstellen für BPMN-Prozesse sind insbesondere die Attribute des Metamodellelements *UserTask* relevant, sowie dessen sprachlich vordefinierten Verbindungen zu anderen Modellelementen inkl. solchen für Datenstrukturen. Das Metamodellelement *UserTask* besitzt das Attribut *implementation* zur Kennzeichnung der Implementierungstechnik wie z. B. *BPEL4People* und das Attribut *renderings* zur Referenzierung herstellerspezifischer Rendering-Informationen [Obj11a, S. 166 f.]. Das Rendering der Benutzerschnittstelle bzw. deren Spezifikation ist also nicht durch BPMN standardisiert. Über die Spezialisierungsbeziehung zu dem Metamodellelement *Task*, das seinerseits eine Spezialisierung von *Activity* ist, enthält es zudem optional unter dem Attribut *ioSpecification* eine *InputOutputSpecification* zur Beschreibung der ein- und ausgehenden Daten

[Obj11a, S. 212 f.] (vgl. Abbildung 58). Dazu enthält diese *DataInputs* und *DataOutputs*, die in Form von *InputSets* und *OutputSets* mehrfach gebündelt werden können.

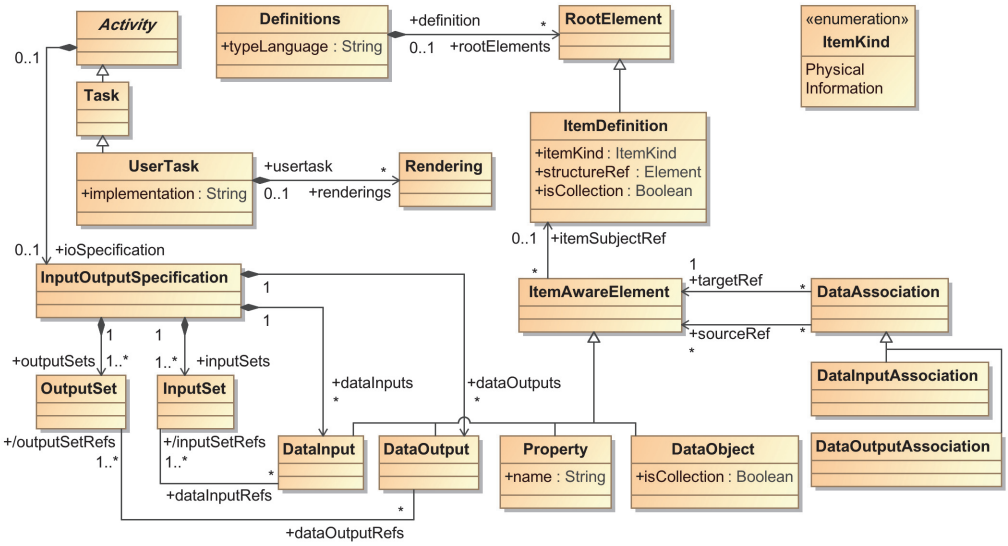

Abbildung 58: BPMN-Datenmodell

Die Speicherung von Daten, die im Laufe der Prozessausführung erzeugt und modifiziert werden, wird in *ItemAwareElements* als Pendants zu Variablen vorgenommen [Obj11a, S. 203 f.]. Wesentliche Spezialisierungen des Metamodellelements *ItemAwareElement* sind *Property* für nicht-visualisierte Prozessvariablen [Obj11a, S. 210 f.], *DataObject* für visualisierte Prozessvariablen [Obj11a, S. 205-208], und *DataInput* sowie *DataOutput* für Schnittstellenvariablen. Jedes *ItemAwareElement* wird optional über das Attribut *itemSubjectRef* durch eine *ItemDefinition* typisiert [Obj11a, S. 204].

Eine *ItemDefinition* repräsentiert einen fachlichen Datentyp wie z. B. einen Vornamen, der durch das Attribut *structureRef* mit einem technischen Datentypen bzw. einer Struktur wie z. B. *String* fundiert wird [Obj11a, S. 91]. Durch das Attribut *itemKind* zu der gleichnamigen Enumeration kann festgelegt werden, ob ein *Item* als ein Ding aus der Realwelt zu interpretieren ist, oder als dessen Repräsentation in Form eines Datensatzes. Durch das Attribut *isCollection* kann zudem die Multiplizität festgelegt werden, die im Standardfall als einwertig angenommen wird. Die BPMN ist nicht auf bestimmte Typsysteme bzw. Strukturen zur Referenzierung durch das Attribut *structureRef* festgelegt, sondern ermöglicht es, über das Attribut *typeLanguage* des Metamodellelements *Definitions* ein Typsystem wie z. B. das von Java oder als Standardfall XML Schema zu referenzieren [Obj11a, S. 53, 91].

Der Transfer von Daten zwischen *ItemAwareElements* wie z. B. zwischen einem *Da-*

taOutput und einem *DataObject* zur Speicherung der Datenausgabe eines *UserTask* wird mit dem Metamodellelement *DataAssociation* spezifiziert [Obj11a, S. 221-224], das mit den Attributen *sourceRef* und *targetRef* die *ItemAwareElements* referenziert. Falls entsprechend der mehrwertigen Kardinalität des Attributs *sourceRef* multiple *ItemAwareElements* als Quellen spezifiziert werden, ist aus diesen durch einen Ausdruck in einer beliebigen Expression Language ein einzelner Wert zu berechnen.

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions ...>
  <itemDefinition id="reportItem" structureRef="String" />

  <process id="weeklyReport" name="Weekly␣report␣process">
    <property id="report" itemSubjectRef="reportItem"/>

    <startEvent id="processStart" />
    <sequenceFlow id="flow1" sourceRef="processStart" targetRef="writeReportTask" />

    <userTask id="writeReportTask" name="Write␣report">
      <ioSpecification>
        <dataOutput id="reportOutput" name="Report" itemSubjectRef="reportItem" />
        <outputSet>
          <dataOutputRefs>reportOutput</dataOutputRefs>
        </outputSet>
      </ioSpecification>
      <dataOutputAssociation>
        <sourceRef>reportOutput</sourceRef>
        <targetRef>report</targetRef>
      </dataOutputAssociation>
    </userTask>

    <sequenceFlow id="flow2" sourceRef="writeReportTask" targetRef="processEnd" />
    <endEvent id="processEnd" />
  </process>
</definitions>
```

Listing 88: Serialisiertes BPMN-Beispielmodell

So lässt sich beispielsweise ein BPMN-Modell konstruieren (vgl. Listing 88), das eine *ItemDefinition* mit der ID *reportItem* vom Basisdatentyp *String* als Datenstruktur für Berichte enthält. Die *ItemDefinition* wird in einem Prozess von einem Property *report* als Typ referenziert, das durch die *DataOutputAssociation* eines *UserTask* mit einem Wert belegt wird.

6.3 Architektur

Das TaskUI-Generatorframework unterstützt die automatisierte Generierung prozessbasierter Webapplikationen mit Fokus auf Benutzerschnittstellen für BPMN-Prozesse. Im Unterschied zu den datenbankorientierten und programmiersprachlich heterogenen Plattformen der datenbasierten Webapplikationen dienen bei den prozessbasierten Pendants BPMN-Ausführungsumgebungen verschiedener Hersteller als Plattformen, deren Schwerpunkt auf Java als Programmiersprache liegt.

Konkret adressiert das Generatorframework exemplarisch die zwei praxisrelevanten BPMN-Plattformen *Activiti BPM Platform* (Activiti) und *jBPM*. Die eingesetzten Ge-

nerator-Technologien sind mit EMF, QVTO und XSL Transformation (XSLT) ähnlich zu denen des WASL-Generatorframeworks, sodass das TaskUI-Generatorframework einen vergleichbar hohen Standardisierungsgrad aufweist, welcher die Adaptionsfähigkeit für projektspezifische Anforderungen sicherstellt. Zudem sind die Codegeneratoren der beiden Generatorframeworks identisch.

Das TaskUI-Generatorframework ist ebenfalls als Schichtenarchitektur mit den Modelltypen CIM, PIM und PSM strukturiert, die durch Transformationsdefinitionen miteinander verbunden sind (vgl. Abbildung 59). Für die CIMs wird als Modellierungssprache aufgrund ihrer fachlichen Ausdrucksmittel BPMN verwendet. Die PIMs der untergeordneten Schicht werden mit der Modellierungssprache *TaskUI Generic* formuliert, die sprachliche Ausdrucksmittel zur Spezifikation von Datenstrukturen und Benutzerschnittstellen für BPMN-Prozesse umfasst (vgl. Kapitel 6.4). Für die Modellierung der plattformspezifischen Implementierungen in Form von PSMs dienen die Modellierungssprachen *TaskUI Activiti* und *TaskUI jBPM*. Die PSMs werden jeweils in den Quelltext der Implementierung übersetzt und können i.V.m. automatisiert angepassten Versionen der BPMN-Modelle ausgeführt werden.

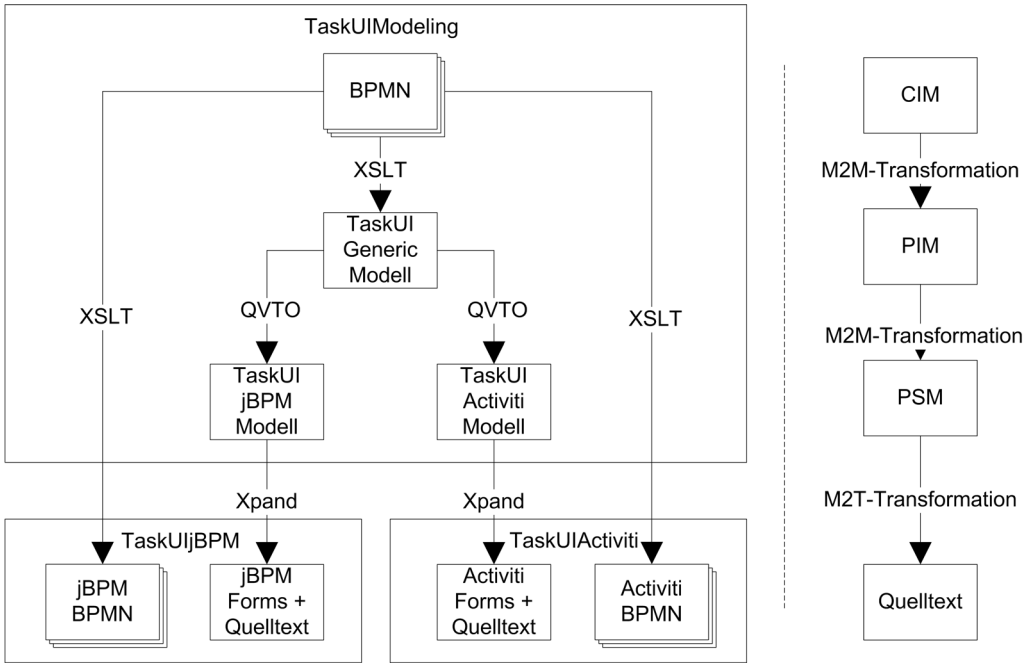

Abbildung 59: Architektur des TaskUI-Generatorframeworks

Für die Modellierungssprache *TaskUI Generic* existiert ein gleichnamiges Metamodell, das mit Ecore als Metasprache spezifiziert ist. Die plattformspezifischen Sprachen resultieren dagegen aus Kombinationen der plattformspezifischen Metamodelle des WASL-Gene-

ratorframeworks und illustrieren somit deren Wiederverwendbarkeit.

Die Transformationsdefinitionen des Generatorframeworks sind in zwei Strängen konzipiert. Einerseits wird aus den CIMs in Form von BPMN-Modellen ein PIM, aus diesem ein PSM, und aus diesem schließlich der Quelltext abgeleitet. Andererseits werden die BPMN-Modelle direkt in plattformspezifische Pendants transformiert. Letzteres ist damit motiviert, dass die BPMN-Ausführungsumgebungen abweichend vom BPMN-Standard kompatibilitätsbedingte Anpassungen an den BPMN-Modellen erfordern. Die BPMN-Modelle sind somit in ihren verschiedenen Versionen sowohl ein fachlicher Startpunkt der Transformation als auch Teil der finalen Implementierung.

Die M2M-Transformationsdefinitionen für BPMN-Eingabemodelle sind mit XSLT als Transformationssprache spezifiziert, da XSLT der technischen Sphäre von XML zuzuordnen ist, und sowohl BPMN als auch XMI als Serialisierungsformat von EMF auf XML basieren. Die Transformationsdefinitionen zwischen dem Metamodell *TaskUIGeneric* und den Metamodellen *TaskUIActiviti* sowie *TaskUIjBPM* sind dagegen in der Transformationssprache QVTO formuliert, da sämtliche der Metamodelle der technischen Sphäre von MOF/EMF entstammen. Der Codegenerator ist identisch zu dem des WASL-Generatorframeworks, da in beiden Frameworks die Metamodelle von WASL PSM Verwendung finden und als Schnittstelle zum Codegenerator dienen (vgl. Abbildung 60).

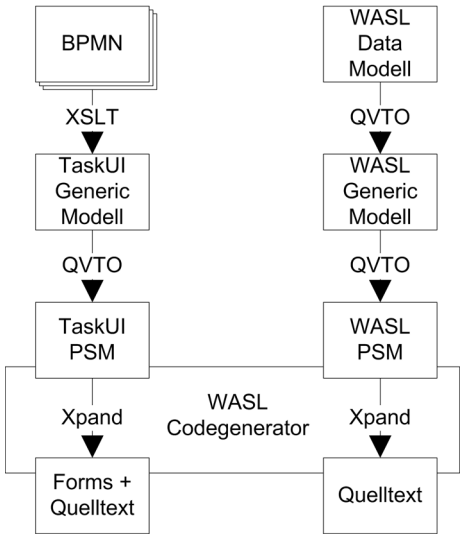

Abbildung 60: TaskUI- vs. WASL-Generatorframework

Bei der Wahl der technischen Sphäre bestünde prinzipiell die Alternative, die XML-basierte Serialisierung eines BPMN-Prozessmodells vorab mittels eines im Kontext des BPMN-Standards definierten XSLT-Stylesheets[54] in eine intermediäre XMI-basierte Form

[54] `http://www.omg.org/spec/BPMN/2.0/`

zu überführen, um die Transformation zu *TaskUI Generic* in der technischen Sphäre von MOF/EMF durchzuführen (vgl. Abbildung 61). Dem Vorteil einer Vereinheitlichung der technischen Sphäre stehen als Nachteile gegenüber, dass erstens herstellerspezifische Erweiterungen des BPMN-Standards wie z. B. die Markierung von Prozesselementen mit dem Attribut *activiti:formKey* im Fall der *Activiti BPM Platform* nicht transformiert werden, zweitens durch das intermediäre Modell die Komplexität der Gesamttransformation erhöht wird und drittens die Auswahl für die MOF-basierte Transformationssprache zur Transformation von BPMN zu *TaskUI Generic* eingeschränkt ist, da sie mehrere BPMN-Eingabemodelle in ein gemeinsames Zielmodell zu überführen hat.

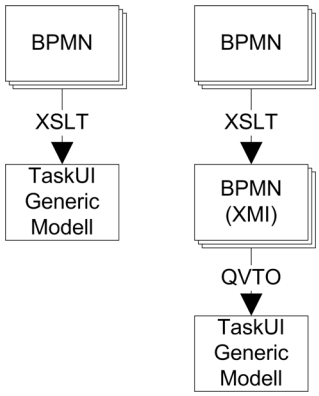

Abbildung 61: Alternativen für die Transformation von BPMN zu *TaskUI Generic*

Die Überführung multipler BPMN-Eingabemodelle per Transformation in ein gemeinsames integrierendes *TaskUI Generic*-Modell ist nötig, um eine prozessübergreifende Sicht auf die Benutzerschnittstellen verschiedener BPMN-Prozesse in einem gemeinsamen *TaskUI Generic*-Modell zu erhalten. Durch die integrierte Darstellung wird die Erkennung von Inkonsistenzen im Modell wie z. B. Namenskollisionen ermöglicht, die Pfadkollisionen im Zuge der Generierung einer Verzeichnisstruktur im PSM induzieren können. Zudem wird die Steuerung des Build-Prozesses vereinfacht, da der Generator nicht erneut für jedes BPMN-Eingabemodell auszuführen ist und entsprechend die potentielle Generierung von Steuerungs- und Konfigurationsdateien für den Build- bzw. Generatorprozess weniger komplex ausfällt.

Ebenfalls sprechen Gründe für die Zusammenführung der BPMN-Eingabemodelle bereits im ersten M2M-Transformationsschritt, anstatt die Integration nachgelagert z. B. im Zuge der Transformation von *TaskUI Generic* zu den plattformspezifischen Modellen vorzunehmen. Diese sind erstens, dass die nachgelagerten Transformationsdefinitionen auf diese Weise kompakt gehalten werden können, da sie nicht multiple Eingabemodelle verarbeiten müssen, und zweitens XSLT eine Zusammenführung dieser Art optimal unterstützt.

Die Methode zur Entwicklung des TaskUI-Generatorframeworks weist einige Detailunterschiede zu derjenigen für die Entwicklung des WASL-Generatorframeworks auf. Erstens entfallen die Projektabschnitte zur Konstruktion der plattformspezifischen Metamodelle und M2T-Transformationsdefinitionen, da diese aus dem WASL-Generatorframework übernommen werden. Zweitens ist die BPMN als Sprache für das CIM vorgegeben, sodass auch die Entwicklung des CIM-Metamodells ausbleibt. Da somit lediglich das Metamodell von *TaskUI Generic* für die PIM-Schicht sowie die verbindenden M2M-Transformationsdefinitionen zu konstruieren und parallel zu evaluieren sind, entfällt drittens das strikte Bottom-Up-Vorgehen.

6.4 Metamodell von *TaskUI Generic*

TaskUI Generic ist eine semiformale und domänenspezifische Modellierungssprache zur plattformunabhängigen Modellierung von Benutzerschnittstellen für BPMN-Geschäftsprozesse. Sie ermöglicht die Modellierung von Daten sowie von Formularen zu deren Modifikation, die im Laufe des Prozesses gegenüber dem Benutzer angezeigt werden.

Das Metamodell *TaskUIGeneric* enthält die EClass *TaskUIGenericModel*, die das Wurzelelement der Containment-Hierarchie eines Modells typisiert (vgl. Abbildung 62). Die EClass referenziert beliebig viele *BPMNFiles*, die BPMN-Dateien zur Serialisierung von Prozessmodellen repräsentieren. Im Vergleich zu dem plattformspezifischen Metamodell *Filesystem* wird auf die Modellierung von Dateisystemhierarchien verzichtet, da diese für die Modellierung der Formulare irrelevant sind.

Da bei der BPMN jedes Wurzelelement *Definitions* (vgl. Listing 88) über die Referenz *rootElements* beliebig viele *Processes* als indirekte Spezialisierung von *RootElement* enthalten kann [Obj11a, S. 52][Obj11a, S. 146], also in einem BPMN-Dokument mehrere Prozesse spezifiziert werden können, umfasst in *TaskUIGeneric* jedes *BPMNFile* beliebig viele *Processes*. Jeder *Process* kann über die EReference *itemAwareElements* zu der EClass *ItemAwareElement* aus dem EPackage *data* Informationen zu Daten, und über die EReference *flowNodes* zu der EClass *FlowNode* aus dem EPackage *presentation* Informationen zu Prozesselementen i.V.m. deren Formularen beinhalten.

Die EClass *ItemAwareElement* typisiert ähnlich zu Variablen und korrespondierend zu dem gleichnamigen Elementtyp der BPMN (vgl. [Obj11a, S. 203]) Prozessdatenelemente, die in Form primitiv und strukturiert typisierter Daten im Prozessablauf modifiziert werden bzw. diesen beeinflussen. So kann beispielsweise in einem BPMN-Modell ein *DataEntity*, z. B. mit dem Namen *approved* und vom Typ *Boolean*, den Prozessfluss an einem exklusiven Gateway (vgl. [Obj11a, S. 290]) steuern. Da die BPMN keine Sprachelemente für die Spezifikation von Datenstrukturen vorgibt, sondern lediglich Erweiterungspunkte

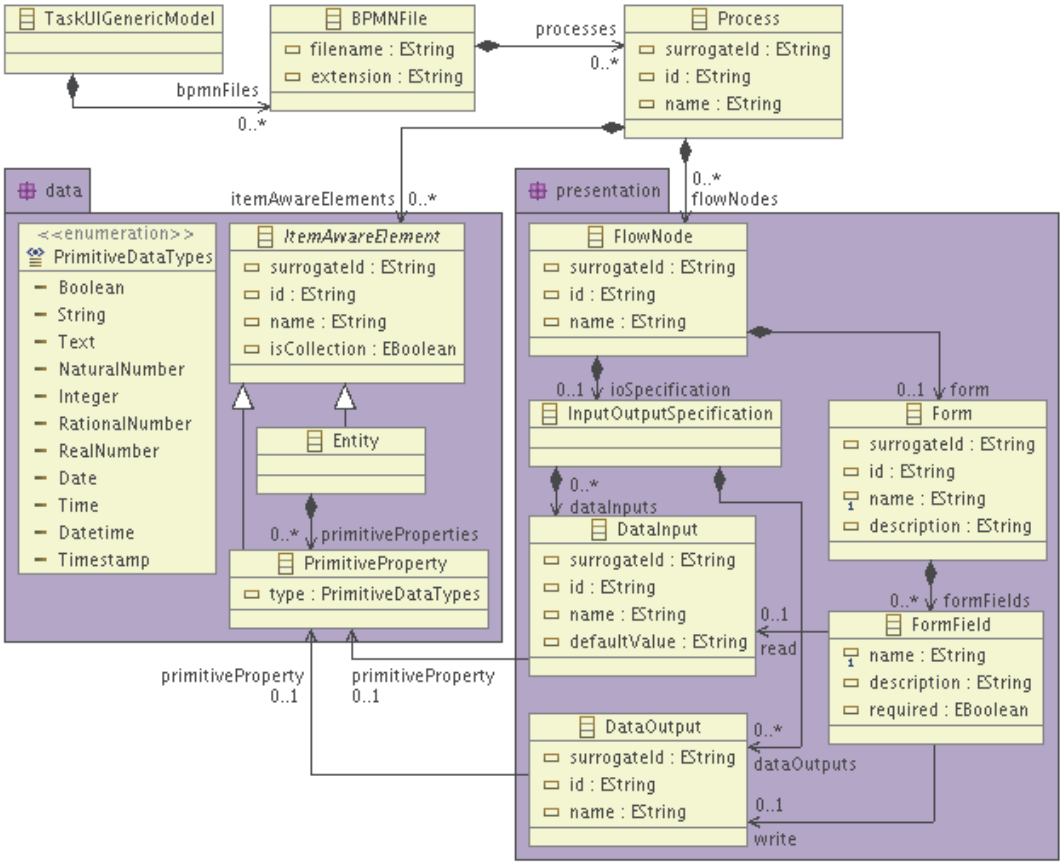

Abbildung 62: Metamodell von *TaskUI Generic*

bietet (vgl. [Obj11a, S. 203]), werden diese in *TaskUIGeneric* in einfacher Form ausgeprägt. Die EClass *PrimitiveProperty* ist eine Spezialisierung von *ItemAwareElement*, und dient der Modellierung von Variablen mit Basisdatentyp. Jedes *PrimitiveProperty* wird durch das EAttribute *type* mit einem der primitiven Datentypen aus der EEnum *PrimitiveDataTypes* typisiert, welche der gleichnamigen EEnum aus dem Metamodell von *WASL Data* entspricht. Ein *PrimitiveProperty* kann entweder in einem *Process* enthalten sein, oder Teil eines *Entitiy* sein, das ebenfalls als Spezialisierung von *ItemAwareElement* der Modellierung einfacher Datenstrukturen im Kontext von Prozessen dient.

Die EClass *FlowNode* wird im Modell für jedes Prozesselement aus einem BPMN-Prozessmodell instanziiert, das Benutzereingaben erwartet. Im Standardfall sind dies *UserTasks*, abhängig von plattformspezifischen Adaptionen der BPMN für bestimmte Ausführungsumgebungen können aber auch Ereignisse wie z. B. *StartEvents* Benutzereingaben fordern und somit in das Modell einfließen. Jeder *FlowNode* kann ein *Form* enthalten, das der Modellierung von Formularen für Dateneingaben dient und dessen Eingabefelder mittels der referenzierten EClass *FormField* modelliert werden. Mit dem EAttribute *required* kann ein *FormField* zudem als obligatorisch für die Eingabe deklariert werden. Jedes *FormField* verweist optional über die EReferences *read* und *write* auf ein *DataInput* bzw. ein *DataOutput*. Diese entstammen der Schnittstellendefinition, die im BPMN-Prozessmodell einem Prozesselement bzw. primär einem *UserTask* zugeordnet ist. Die Schnittstellendefinition beschreibt die Daten, die bei der Prozessausführung in den *UserTask* eingehen bzw. von diesem ausgegeben werden (vgl. [Obj11a, S. 211 f.]). Analog zu der BPMN sind die Details der Schnittstelle im *TaskUIGeneric*-Metamodell in einem Element vom Typ *InputOutputSpecification* enthalten, das in demjenigen *FlowNode* enthalten ist, dessen Schnittstelle beschrieben wird.

Die EClasses *Process*, *ItemAwareElement*, *FlowNode*, *DataInput*, *DataOutput*, *Form* und *FormField* besitzen jeweils die EAttributes *surrogatedId* und *id* vom Typ *EString*. Bei der Transformation werden die Id-Werte der verschiedenen BPMN-Modellelemente gleichwertig in das EAttribute *id* des jeweiligen Modellelements im *TaskUI Generic*-Modell kopiert. Da mehrere BPMN-Modelle in die Erstellung eines *TaskUI Generic*-Modells eingehen können, ist die Eindeutigkeit der Id-Werte aber nicht sichergestellt. Um im XSLT-Stylesheet Referenzen zu Modellelementen in Form direkter Referenzen zu intrinsischen Ids (vgl. [SBPM08, S. 485]) anstelle von pfadähnlichen URI Fragmenten (vgl. [SBPM08, S. 483 f.]) wie z. B. *//@bpmnFiles.0/@processes.0* verwenden zu können, wird deshalb jedem der Modellelemente unter dem EAttribute *surrogatedId* ein Surrogatschlüssel hinzugefügt. Dieses Vorgehen vereinfacht auch die Transformationsdefinition. Das EAttribute wird durch das Meta-EAttribute *ID* als Schlüssel gekennzeichnet, sodass mit dem EAttribute bzw.

dessen Wert die enthaltende EClass innerhalb des Modells direkt referenziert werden kann. Da die Schlüssel im BPMN-Prozessmodell vom Typ *string* sind [Obj11a, S. 56], ist beiden EAttributes ebenfalls der Typ EString zugeordnet, und nicht etwa der Typ EInteger.

6.5 Modell-zu-Modell-Transformationsdefinitionen

Im Folgenden wird auf die M2M-Transformationsdefinitionen *Bpmn2TaskUIGeneric* und exemplarisch *TaskUIGeneric2TaskUIActiviti* des TaskUI-Generatorframeworks eingegangen. Sie sind unidirektional mit den Transformationssprachen XSLT und QVTO implementiert und referenzieren Elemente aus dem Metamodell *TaskUIGeneric* sowie solche aus den Metamodellen von *WASL JavaEE*.

6.5.1 Transformationsdefinition *Bpmn2TaskUIGeneric*

Die M2M-Transformationsdefinition *Bpmn2TaskUIGeneric* leitet *TaskUI Generic*-Modelle aus BPMN-Prozessmodellen ab. Sie ist als XSLT-Stylesheet formuliert, da BPMN in der technischen Sphäre von XML verortet ist, *TaskUI Generic* dagegen in derjenigen von MOF/EMF, dessen Serialisierungsformat mit XMI ebenfalls auf XML basiert. Das XSLT-Stylesheet enthält als Einstiegspunkt ein namenloses Template, welches durch das Attribut *match="/"* bei der Ausführung automatisch auf das Wurzelelement des XML-Eingabedokuments angewendet wird und zu dessen Verarbeitung das Template *Index2TaskUIGenericModel* aufruft (vgl. Listing 89).

```
1 <xsl:stylesheet version="2.0" ...
2   xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
3   xmlns:bpmn2taskuigeneric="http://bpmn2taskuigeneric" >
4
5   <xsl:template match="/">
6     <xsl:call-template name="Index2TaskUIGenericModel" />
7   </xsl:template>
8   ...
9 </xsl:stylesheet>
```

Listing 89: Stylesheeet *Bpmn2TaskUIGeneric*

Das Eingabedokument für die Transformation spezifiziert einen Index über die Dateien der BPMN-Eingabemodelle in Form von XML-Elementen, die den XML-Tag *bpmnFile* tragen und jeweils auf die BPMN-Modelldatei über den Verzeichnispfad und den Dateinamen verweisen (vgl. Listing 90).

Auf dem Index der XML-basierten BPMN-Eingabedokumente wird das Template *Index2TaskUIGenericModel* aufgerufen, welches als Ausgabe in der XMI-Datei des *TaskUI Generic*-Modells ein XML-Wurzelelement in Form des XML-Elements *TaskUIGenericModel* erzeugt (vgl. Listing 91, Zeile 2). Für jedes *bpmnFile*-Element aus dem Index (vgl. Zeile 4) wird mit der XSLT-Funktion *document* das korrespondierende BPMN-Eingabemodell

```
1  <?xml version="1.0"?>
2  <bpmn2taskuigeneric:index xmlns:bpmn2taskuigeneric="http://bpmn2taskuigeneric">
3     <bpmnFile dir="./models/" filename="WeeklyReportProcess.xml"/>
4     <bpmnFile dir="./models/" filename="VacationRequestProcess.xml"/>
5     ...
6  </bpmn2taskuigeneric:index>
```

Listing 90: Index über die BPMN-Eingabemodelle

geladen (vgl. Zeile 13), und für dessen Wurzelelement mit dem XML-Tag *definitions* dem *TaskUI Generic*-Modell unter der EReference *bpmnFiles* ein Modellelement hinzugefügt (vgl. Zeilen 13-20). Die Typisierung dieses Elements mit der EClass *BPMNFile* aus dem Metamodell *TaskUIGeneric* geschieht implizit über den Bezug des XMI-Dokuments zu der Datei des Metamodells (vgl. Zeile 3). Die EAttributes und EReferences jedes *BPMNFile* werden durch das Template *Definition2BPMNFile* angelegt, das im Kontext der *bpmnFiles*-Elemente aufgerufen wird (vgl. Zeile 15).

```
1  <xsl:template name="Index2TaskUIGenericModel">
2    <taskuigeneric1.0:TaskUIGenericModel xmlns:xmi="http://www.omg.org/XMI"
3      xsi:schemaLocation="taskuigeneric1.0␣metamodels/TaskUIGeneric1.0.ecore" ...>
4      <xsl:for-each select="bpmn2taskuigeneric:index/bpmnFile">
5        <xsl:variable name="filename" select="@filename" />
6        <xsl:variable name="extension">
7          <xsl:call-template name="get-file-extension">
8            <xsl:with-param name="path" select="@filename" />
9          </xsl:call-template>
10       </xsl:variable>
11       <xsl:variable name="dir" select="@dir" />
12
13       <xsl:for-each select="document(concat($dir,␣$filename))/bpmn:definitions">
14         <bpmnFiles>
15           <xsl:call-template name="Definition2BPMNFile">
16             <xsl:with-param name="filename" select="$filename" />
17             <xsl:with-param name="extension" select="$extension" />
18           </xsl:call-template>
19         </bpmnFiles>
20       </xsl:for-each>
21     </xsl:for-each>
22   </taskuigeneric1.0:TaskUIGenericModel>
23 </xsl:template>
```

Listing 91: Template *Index2TaskUIGenericModel*

Das Template *Definition2BPMNFile* legt für jeden der BPMN-Prozesse aus dem Eingabemodell einen *Process* im Zielmodell an, indem es für jedes der *process*-Elemente aus dem BPMN-Modell im Zielmodell dem *BPMNFile* unter der EReference *processes* ein Modellelement vom *TaskUIGeneric*-Typ *Process* hinzufügt (vgl. Listing 92). Jeder erstellte *Process* wird jeweils durch Aufruf des Templates *Process2Process* attributiert, das die Parameter *definition* und *filename* entgegennimmt, die vorab für das Element vom Typ *BPMNFile* berechnet werden. Bei einer stringenten Auslegung des BPMN-Standards wäre die Menge der eingehenden Prozesse eigentlich auf solche zu beschränkten, die mit dem Attribut *isExecutable* [Obj11a, S. 148] als ausführbar definiert sind (vgl. Zeile 10). In der praktischen Umsetzung scheitert dies aber an der Unterspezifikation vieler BPMN-Prozess-

modelle.

```
1  <xsl:template name="Definition2BPMNFile">
2    <xsl:param name="filename"></xsl:param>
3    <xsl:param name="extension"></xsl:param>
4
5    <xsl:attribute name="filename"><xsl:value-of select="$filename" /></xsl:attribute>
6    <xsl:attribute name="extension"><xsl:value-of select="$extension" /></xsl:attribute>
7
8    <xsl:variable name="definition" select="." />
9    <xsl:for-each select="./bpmn:process">
10   <!-- alternativ: ./bpmn:process[@isExecutable = 'true'] -->
11     <processes>
12       <xsl:call-template name="Process2Process">
13         <xsl:with-param name="definition" select="$definition" />
14         <xsl:with-param name="filename" select="$filename" />
15       </xsl:call-template>
16     </processes>
17   </xsl:for-each>
18 </xsl:template>
```

Listing 92: Template *Definition2BPMNFile*

Das Template *Process2Process* nimmt mit dem Parameter *filename* den Dateinamen des BPMN-Modells sowie mit dem Parameter *definition* das darin enthaltene XML-Wurzelelement entgegen und legt die EAttributes und EReferences eines *Process*-Modellelements an (vgl. Listing 93). Da sich diese Parameterkonstellation in sämtlichen weiteren Templates des XSLT-Stylesheets wiederholt, unterbleibt im Folgenden deren Darstellung in den Listings.

Das Template schreibt initial Werte in die EAttributes *id*, *surrogateId* und *name* (vgl. Zeilen 5-8). Der Surrogatschlüssel wird nach dem Schema *<filename>_PROCESS_<id>* erstellt (vgl. Zeile 7) und ist modellweit eindeutig, da sowohl der Dateiname, als auch die ID per Prämisse eindeutig sind. Eine Bedingung für den Surrogatschlüssel als intrinsische ID im Sinne von EMF [SBPM08, S. 485] ist dabei, dass der Surrogatschlüssel ein NCName[55] im Sinne von XML Schema sein muss [SBPM08, S. 495], also keinen Doppelpunkt enthalten darf.

Die EReference *itemAwareElements* des *Process*-Elements wird mit *PrimitiveProperties* belegt, indem im Zielmodell XML-Elemente mit dem Tag *itemAwareElements* für jedes *Property* und *DataObject* (vgl. Zeile 10) sowie sämtliche *DataInputs* (vgl. Zeile 16) und *DataOutputs* (vgl. Zeile 27) von *UserTasks* ohne Bezug zu einem *ItemAwareElement* (vgl. Zeile 20) angelegt werden. Die *PrimitiveProperties* werden als solche durch Aufruf des Templates *ItemAwareElement2PrimitiveProperty* bzw. *DataInOutput2PrimitiveProperty* gekennzeichnet (vgl. Zeilen 12 und 22).

Die Generierung von *PrimitiveProperties* aus *DataInputs* und *DataOutputs* der *UserTasks* des BPMN-Modells ist damit motiviert, dass im *TaskUIGenericModel* optimalerweise jedem *FormField* mindestens ein *PrimitiveProperty* indirekt über den *DataInput* bzw.

[55] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#NCName

```
1  <xsl:template name="Process2Process">
2    <xsl:param name="definition" />
3    <xsl:param name="filename" />
4
5    <xsl:attribute name="id"><xsl:value-of select="@id" /></xsl:attribute>
6    <xsl:attribute name="surrogateId"><xsl:value-of
7      select="concat($filename,␣'_PROCESS_',␣@id)" /></xsl:attribute>
8    <xsl:attribute name="name"><xsl:value-of select="@name" /></xsl:attribute>
9
10   <xsl:for-each select=".//bpmn:property|.//bpmn:dataObject">
11     <itemAwareElements>
12       <xsl:call-template name="ItemAwareElement2PrimitiveProperty" />
13     </itemAwareElements>
14   </xsl:for-each>
15
16   <xsl:for-each select="./bpmn:userTask/bpmn:ioSpecification/bpmn:dataInput">
17     <xsl:variable name="id" select="@id" />
18     <xsl:variable name="DIA"
19       select="../../bpmn:dataInputAssociation[bpmn:targetRef/text()=$id]" />
20     <xsl:if test="string-length($DIA/bpmn:sourceRef/text())␣=␣0">
21       <itemAwareElements>
22         <xsl:call-template name="DataInOutput2PrimitiveProperty" />
23       </itemAwareElements>
24     </xsl:if>
25   </xsl:for-each>
26
27   <xsl:for-each select="./bpmn:userTask/bpmn:ioSpecification/bpmn:dataOutput">
28     ...
29   </xsl:for-each>
30
31   <xsl:for-each select=".//bpmn:userTask">
32     <flowNodes>
33       <xsl:call-template name="FlowNode2FlowNode" />
34     </flowNodes>
35   </xsl:for-each>
36
37   <xsl:call-template name="Process2ActivitiFlowElements" />
38 </xsl:template>
```

Listing 93: Vereinfachte Darstellung des Templates *Process2Process*

DataOutput zugeordnet ist, dessen EAttribute *type* das *FormField* typisiert. Die Generierung eines *PrimitiveProperty* aus einem *DataInput* bzw. *DataOutput* des BPMN-Modells ist aber nur dann sinnvoll, falls der *DataInput* bzw. *DataOutput* im BPMN-Modell nicht auf ein *Property* oder *DataObject* verweist, das wie beschrieben bereits eigenständig in das *TaskUIGenericModel* überführt wird. Diese Kondition wird überprüft, indem z. B. für die *DataInputs* aus dem BPMN-Modell jeweils initial unter der Variable *DIA* die *DataInputAssociation* ermittelt wird (vgl. Zeile 18 f.), die auf das betreffende *DataInput* verweist, und nur im Fall einer Nichtbelegung des Referenzattributs *sourceRef* (vgl. Zeile 20), also bei einem fehlenden Verweis auf ein *Property* oder *DataObject*, das *PrimitiveProperty* unter der EReference *itemAwareElements* angelegt wird (vgl. Zeilen 21-23). Falls die Referenz *sourceRef* nicht nur unbelegt ist, sondern die *DataInputAssociation* generell dem *UserTask* fehlt, evaluiert die Kondition ebenfalls zu *true* und führt somit ebenfalls zu der Generierung eines *PrimitiveProperty*. Für die Erstellung von *PrimitiveProperties* aus *DataOutputs* wird die Überprüfung analog mit Fokus auf das Referenzattribut *targetRef* durchgeführt.

Abhängig von dem Fehlen einer *DataInputAssociation* bzw. *DataOutputAssociation* resultiert ein *DataInput* bzw. *DataOutput* des BPMN-Modells also in der Generierung eines

PrimitiveProperty im *TaskUIGenericModel*. Für eine Schnittstellendefinition mit maximal einem *DataInput* und ebenso maximal einem *DataOutput* ergeben sich aus dem Zusammenspiel der Selektionen und Konditionen im Template *Process2Process* acht mögliche Fälle (vgl. Tabelle 2), die aber auch bei umfangreicheren Schnittstellendefinitionen mit multiplen *DataInputs* und *DataOutputs* gültig sind.

So bildet z. B. der Fall *C1* eine Konstellation ab, bei der ein *PrimitiveProperty* aus einem *DataInput* erstellt wird, da im BPMN-Modell von dem *UserTask* bzw. dessen Schnittstellendefinition keine Prozessvariable referenziert wird (vgl. Listing 94). Dagegen spiegelt Fall *C2* eine Konstellation wider, bei der ein *PrimitiveProperty* aus einem *ItemAwareElement* erstellt wird, das durch ein *DataInput* referenziert wird (vgl. Listing 95). In diesem Fall verweist im BPMN-Modell die Schnittstellendefinition des *UserTask* also auf eine Prozessvariable, deren Repräsentation im *TaskUIGenericModel* referenziert werden kann, sodass für den *DataInput* aus der Schnittstellendefinition keine eigene Prozessvariable anzulegen ist. Analog sind die Fälle *C3* und *C4* für ein *DataOutput* ohne bzw. mit Referenz auf eine Prozessvariable möglich. In den Fällen *C5*, *C6* und *C7* sind sowohl ein *DataInput* als auch ein *DataOutput* Teil der Schnittstellendefinition, die sich aber nicht explizit durch *DataAssociations* auf dasselbe *ItemAwareProperty* beziehen. Da eine Verbindung zwischen dem *DataInput* und dem *DataOutput* in diesen Fällen nur heuristisch z. B. über Namensvergleiche möglich ist, resultieren diese Fälle im Zielmodell in zwei separaten *PrimitiveProperties*. Eine klare Beziehung zwischen dem *DataInput* und dem *DataOutput* lässt sich dagegen in Fall *C8* feststellen, bei dem beide jeweils über eine *DataAssociation* dasselbe *ItemAwareProperty* referenzieren (vgl. Listing 96), aus dem lediglich ein einzelnes *PrimitiveProperty* resultiert.

```
1  <definitions ...>
2    <process id="weeklyReport" name="Weekly␣report␣process">
3      <userTask id="writeReportTask" name="Write␣report">
4        <ioSpecification>
5          <dataInput id="reportInput" name="Report" itemSubjectRef="reportItem" />
6          <inputSet>
7            <dataInputRefs>reportInput</dataInputRefs>
8          </inputSet>
9        </ioSpecification>
10     </userTask>
11   </process>
12 </definitions>
```

Listing 94: BPMN-Beispielmodell für Fall *C1*

Das Template *Process2Process* transformiert abschließend die *UserTasks*, indem es für jeden *UserTask* aus dem BPMN-Modell einen *FlowNode* im *TaskUIGenericModel* anlegt (vgl. Listing 93, Zeilen 31-35). Der nachfolgende Aufruf des Templates *Process2Activiti-FlowElements* (vgl. Zeile 37) überführt zudem *StartEvents* aus BPMN-Modellen in *FlowNodes*, falls diese abweichend vom Standard durch herstellerspezifische Erweiterungen der

Fall	Data-Input	DataInput-Association	Data-Output	DataOutput-Association	Implikation
C1	ja				1 *PrimitiveProperty* aus *DataInput*
C2	ja	ja			1 *PrimitiveProperty* aus *ItemAware-Element*
C3			ja		1 *PrimitiveProperty* aus *DataOut-put*
C4			ja	ja	1 *PrimitiveProperty* aus *ItemAware-Element*
C5	ja		ja		2 *PrimitiveProperties* aus *DataInput* und *DataOutput*
C6	ja	ja	ja		2 *PrimitiveProperties* aus *ItemAwa-reElement* und *DataOutput*
C7	ja		ja	ja	2 *PrimitiveProperties* aus *DataInput* und *ItemAwareElement*
C8	ja	ja	ja	ja	1 *PrimitiveProperty* aus *ItemAware-Element*

Tabelle 2: Konstellationen der Schnittstellendefinitionen von *Activities* und deren Implikationen für die Generierung

Activiti BPM Platform ebenfalls Formulare anzeigen. Vor diesem Hintergrund ist auch die generische Benennung der EReference als *flowNodes* motiviert, und nicht etwa als *user-Tasks*.

Das Template *ItemAwareElement2PrimitiveProperty* (vgl. Listing 97) wird durch das Template *Process2Process* aufgerufen (vgl. Listing 93, Zeile 12) und dient der Attributierung von *PrimitiveProperties*, die aus *ItemAwareElements* abgeleitet sind. Analog wird das Template *DataInOutput2PrimitiveProperty* zur Attributierung von *PrimitiveProperties* aufgerufen, die aus *DataInputs* und *DataOutputs* abgeleitet sind (vgl. Listing 93, Zeile 22). Das Template ist weitestgehend ähnlich konzipiert wie das Template *ItemAwareElement2PrimitiveProperty*, da *DataInput* und *DataOutput* Spezialisierungen von *ItemAwareElement* sind [Obj11a, S. 204]. Initial wird durch beide Templates im Zielmodell jeder Instanz der EClass *PrimitiveProperty* explizit der Typ *data:PrimitiveProperty* zugeordnet (vgl. Listing 97, Zeile 2). Dies ist notwendig, da die Containment-EReference *itemAwareElements* der EClass *Process* auf die EClass *ItemAwareElement* verweist bzw. diese als Typ referenziert, und somit aus dem Metamodell nicht hervorgeht, ob die erzeugten Modellelemente spezialisierend den Typ *PrimitiveProperty* oder den Typ *Entity* aufweisen.

```
<definitions ...>
  <itemDefinition id="reportItem" structureRef="String" />
  <process id="weeklyReport" name="Weekly␣report␣process">
    <dataObject id="report" itemSubjectRef="reportItem"/>
    <userTask id="writeReportTask" name="Write␣report">
      <ioSpecification>
        <dataInput id="reportInput" name="Report" itemSubjectRef="reportItem" />
        <inputSet>
          <dataInputRefs>reportInput</dataInputRefs>
        </inputSet>
      </ioSpecification>
      <dataInputAssociation>
        <sourceRef>report</sourceRef>
        <targetRef>reportInput</targetRef>
      </dataInputAssociation>
    </userTask>
  </process>
</definitions>
```

Listing 95: BPMN-Beispielmodell für Fall *C2*

```
<definitions ...>
  <itemDefinition id="reportItem" structureRef="String" />
  <process id="weeklyReport" name="Weekly␣report␣process">
    <property id="report" itemSubjectRef="reportItem"/>
    <userTask id="writeReportTask" name="Write␣report">
      <ioSpecification>
        <dataInput id="reportInput" name="Report" itemSubjectRef="reportItem" />
        <dataOutput id="reportOutput" name="Report" itemSubjectRef="reportItem" />
        <inputSet>
          <dataInputRefs>reportInput</dataInputRefs>
        </inputSet>
        <outputSet>
          <dataOutputRefs>reportOutput</dataOutputRefs>
        </outputSet>
      </ioSpecification>
      <dataInputAssociation>
        <sourceRef>report</sourceRef>
        <targetRef>reportInput</targetRef>
      </dataInputAssociation>
      <dataOutputAssociation>
        <sourceRef>reportOutput</sourceRef>
        <targetRef>report</targetRef>
      </dataOutputAssociation>
    </userTask>
  </process>
</definitions>
```

Listing 96: BPMN-Beispielmodell für Fall *C8*

Im Anschluss an die Belegung der EAttributes *id*, *surrogateId* und *name* mit Werten (vgl. Zeilen 3-6) wird durch Aufruf des Templates *ItemDefinition2PrimitivePropertyType* das EAttribute *type* mit einem Wert aus der EEnum *PrimitiveTypes* des Metamodells *TaskUIGeneric* belegt (vgl. Zeilen 7-10). Dazu wird als Parameter die ID der *ItemDefinition* aus dem Attribut *itemSubjectRef* übergeben, anhand derer der zugrunde liegende Basisdatentyp ermittelt wird.

Das Template *ItemDefinition2PrimitivePropertyType* legt den Basisdatentyp eines *PrimitiveProperty* im Zielmodell fest. Es ermittelt den Basisdatentyp der *ItemDefinition* im BPMN-Modell, indem initial aus der *Definition* die *ItemDefinition* mit der passenden ID selektiert wird (vgl. Listing 98, Zeile 6), und deren Attribut *structureRef* als textueller Verweis auf den Basisdatentyp unter der Variable *type* gespeichert wird (vgl. Zeile 5). Der

```
<xsl:template name="ItemAwareElement2PrimitiveProperty">
  <xsl:attribute name="xsi:type">data:PrimitiveProperty</xsl:attribute>
  <xsl:attribute name="id"><xsl:value-of select="@id" /></xsl:attribute>
  <xsl:attribute name="surrogateId"><xsl:value-of
    select="concat($filename,␣'_ITEMAWAREELEMENT_',␣@id)" /></xsl:attribute>
  <xsl:attribute name="name"><xsl:value-of select="@name" /></xsl:attribute>
  <xsl:call-template name="ItemDefinition2PrimitivePropertyType">
    <xsl:with-param name="definition" select="$definition" />
    <xsl:with-param name="itemDefinitionId" select="./@itemSubjectRef" />
  </xsl:call-template>
</xsl:template>
```

Listing 97: Vereinfachte Darstellung des Templates *ItemAwareElement2PrimitiveProperty*

Basisdatentyp muss im Kontext seines Typsystems wie z. B. dem von Java oder XML interpretiert werden, das in Form einer URI über das Attribute *typeLanguage* der *Definition* [Obj11a, S. 53] festgelegt wird (vgl. Zeile 8). Das Template umfasst für jedes Typsystem Zuordnungen, die den Typ in Kleinschreibweise auf einen Basisdatentyp aus dem Metamodell *TaskUIGeneric* abbilden (vgl. Zeilen 9-34), und schreibt diesen abschließend in das EAttribute *type* der EClass *PrimitiveProperty*.

```
<xsl:template name="ItemDefinition2PrimitivePropertyType">
  <xsl:param name="definition" />
  <xsl:param name="itemDefinitionId" />

  <xsl:variable name="type"
    select="$definition//bpmn:itemDefinition[@id=$itemDefinitionId]/@structureRef" />
  <xsl:if test="string-length($type)␣>␣0">
    <xsl:variable name="typeLanguage" select="$definition/@typeLanguage" />
    <xsl:choose>
      <!-- Java Types: http://www.java.com/javaTypes -->
      <xsl:when test="$typeLanguage␣=␣'http://www.java.com/javaTypes'">
        <xsl:choose>
          <xsl:when test="translate($type,␣$uppercase,␣$smallcase)='boolean'">
            <xsl:attribute name="type">Boolean</xsl:attribute>
          </xsl:when>
          <xsl:when test="translate($type,␣$uppercase,␣$smallcase)='string'">
            <xsl:attribute name="type">String</xsl:attribute>
          </xsl:when>
          <xsl:when test="translate($type,␣$uppercase,␣$smallcase)='int'">
            <xsl:attribute name="type">Integer</xsl:attribute>
          </xsl:when>
          ...
        </xsl:choose>
      </xsl:when>
      <!-- XML Types: http://www.w3.org/2001/XMLSchema -->
      <xsl:otherwise>
        <xsl:choose>
          <xsl:when test="translate($type,␣$uppercase,␣$smallcase)='boolean'">
            <xsl:attribute name="type">Boolean</xsl:attribute>
          </xsl:when>
          ...
        </xsl:choose>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:if>
</xsl:template>
```

Listing 98: Vereinfachte Darstellung des Templates *ItemDefinition2PrimitivePropertyType*

Im Anschluss an die Generierung von *PrimitiveProperties* ruft das Template *Process2Process* wie beschrieben das Template *FlowNode2FlowNode* auf (vgl. Listing 93, Zeile 33), das die *FlowNodes* attributiert, die aus *UserTasks* und *StartEvents* abgeleitet sind.

Das Template *FlowNode2FlowNode* erzeugt im Anschluss an die EAttributes *id*, *surrogateId* und *name* obligatorisch ein *Form* und optional eine *InputOutputSpecification*, falls der *FlowNode* im BPMN-Modell eine solche besitzt (vgl. Listing 99).

```
<xsl:template name="FlowNode2FlowNode">
  ...
  <xsl:for-each select="./bpmn:ioSpecification">
    <ioSpecification>
      <xsl:call-template name="InputOutputSpecification2InputOutputSpecification" />
    </ioSpecification>
  </xsl:for-each>
  <form>
    <xsl:call-template name="FlowNode2Form" />
  </form>
</xsl:template>
```

Listing 99: Vereinfachte Darstellung des Templates *FlowNode2FlowNode*

Die *InputOutputSpecification* eines *FlowElement* aus dem BPMN-Modell wird mittels des Templates *InputOutputSpecification2InputOutputSpecification* inkl. seiner *DataInputs* und *DataOutputs* in das *TaskUIGeneric*-Modell überführt (vgl. Listing 100).

```
<xsl:template name="InputOutputSpecification2InputOutputSpecification">
  <xsl:for-each select="./bpmn:dataInput">
    <dataInputs>
      <xsl:call-template name="DataInput2DataInput" />
    </dataInputs>
  </xsl:for-each>
  ...
</xsl:template>
```

Listing 100: Vereinfachte Darstellung des Templates *InputOutputSpecification2InputOutputSpecification*

Im *TaskUIGeneric*-Modell referenziert jeder *DataInput* und *DataOutput* optional ein *PrimitiveProperty*, das auf diesem Weg dem Formular des *FlowElement* bzw. dessen Formularfeldern zugänglich gemacht wird. Das Template *DataInput2DataInput* (vgl. Listing 101) bzw. *DataOutput2DataOutput* erstellt deshalb neben den üblichen EAttributes *id*, *surrogateId* und *name* eine Referenz zu dem zugrunde liegenden *PrimitiveProperty* durch Aufruf des Templates *DataInOutput2ItemAwareElementId*.

```
<xsl:template name="DataInput2DataInput">
  ...
  <xsl:call-template name="DataInOutput2ItemAwareElementId" />
</xsl:template>
```

Listing 101: Vereinfachte Darstellung des Templates *DataInput2DataInput*

Das Template *DataInOutput2ItemAwareElementId* generiert im *TaskUIGeneric*-Modell für ein *DataInput* bzw. ein *DataOutput* die EReference *primitiveProperty* inkl. deren Wert, der von der Schnittstellendefinition des *FlowElement* im BPMN-Modell bzw. den Fällen *C1-C8* abhängt (vgl. Listing 102). Falls eine Referenz zu einem *PrimitiveProperty* zu

erstellen ist, das seinen Ursprung in einem *ItemAwareElement* des BPMN-Modells hat, und über eine *DataInputAssociation* in Beziehung zu dem *DataInput* steht (Fälle *C2*, *C6*, *C8*), wird dessen Surrogatschlüssel rekonstruiert und in das Attribute *primitiveProperty* geschrieben. Selbiges geschieht für ein *DataOutput* i.V.m. einer *DataOutputAssociation* (Fälle *C4*, *C7*, *C8*). Falls dagegen keine *DataInputAssociation* für den *DataInput* bzw. keine *DataOutputAssociation* für den *DataOutput* existiert, und somit durch das Template *Process2Process* neue *PrimitiveProperties* generiert wurden (Fälle *C1*, *C5*, *C7* bzw. *C3*, *C5*, *C6*), werden deren Surrogatschlüssel rekonstruiert und ebenfalls dem Attribut hinzugefügt.

```
1  <xsl:template name="DataInOutput2ItemAwareElementId">
2    <xsl:variable name="id" select="@id" />
3    <xsl:variable name="DIA"
4      select="../../bpmn:dataInputAssociation[bpmn:targetRef/text()=$id]" />
5    <xsl:variable name="DOA"
6      select="../../bpmn:dataOutputAssociation[bpmn:sourceRef/text()=$id]" />
7    ...
8    <xsl:choose>
9      <!-- cases 2, 6, 8 -->
10     <xsl:when
11       test="name()␣=␣'dataInput'␣and␣string-length($DIA/bpmn:sourceRef/text())␣>␣0">
12       <xsl:attribute name="primitiveProperty"><xsl:value-of
13         select="concat($filename,␣'_ITEMAWAREELEMENT_',␣$DIA/bpmn:sourceRef/text())" /></
                 xsl:attribute>
14     </xsl:when>
15     <!-- cases 4, 7, 8 -->
16     <xsl:when
17       test="name()␣=␣'dataOutput'␣and␣string-length($DOA/bpmn:targetRef/text())␣>␣0">
18       <xsl:attribute name="primitiveProperty">...</xsl:attribute>
19     </xsl:when>
20     <!-- cases 1, 5, 7 -->
21     <xsl:when
22       test="name()␣=␣'dataInput'␣and␣string-length($DIA/bpmn:sourceRef/text())␣=␣0">
23       <xsl:attribute name="primitiveProperty">...</xsl:attribute>
24     </xsl:when>
25     <!-- cases 3, 5, 6 -->
26     <xsl:when
27       test="name()␣=␣'dataOutput'␣and␣string-length($DOA/bpmn:targetRef/text())␣=␣0">
28       <xsl:attribute name="primitiveProperty">...</xsl:attribute>
29     </xsl:when>
30     <xsl:otherwise></xsl:otherwise>
31   </xsl:choose>
32 </xsl:template>
```

Listing 102: Vereinfachte Darstellung des Templates *DataInOutput2ItemAwareElementId*

Jeder *FlowNode* besitzt über die *InputOutputSpecification* hinaus zudem optional ein Formular bzw. *Form*, dessen Attribute mit dem Template *FlowElement2Form* erstellt werden (vgl. Listing 103). Im Wesentlichen werden *FormFields* unter der EReference *formFields* erzeugt, die aus den *DataInputs* und *DataOutputs* des BPMN-Modells abgeleitet werden. Generell werden *FormFields* sowohl für *DataInputs*, als auch für *DataOutputs* angelegt, falls diese nicht über *DataAssociations* zu demselben *ItemAwareElement* in Verbindung zueinander stehen (Fall *C8*), um als Vorlage für manuelle Verfeinerungen des *TaskUIGeneric*-Modells zu dienen. Daraus folgt, dass in den Fällen *C5*, *C6* und *C7* korrespondierend zu den beiden *PrimitiveProperties* auch zwei *FormFields* erzeugt werden.

Im Detail ermittelt das Template für jeden *DataInput* des *FlowElement* die potentiell

eingehende *DataInputAssociation* (vgl. Zeile 5 f.) sowie die potentielle *DataOutputAssociation* (vgl. Zeile 7 ff.), die dasselbe *ItemAwareElement* referenziert, und legt das *FormField* durch Aufruf des Templates *DataInOutput2FormField* mit dem *DataInput* und dem *DataOutput* als Parametern an (vgl. Zeilen 10-15).

Für jeden *DataOutput* geschieht selbiges, falls Fall *C8* nicht vorliegt (vgl. Zeile 18 f.). Zu dessen Überprüfung wird die potentiell ausgehende *DataOutputAssociation* selektiert (vgl. Zeile 22), und erstens festgestellt, ob diese mit dem Attribut *targetRef* auf ein *ItemAwareElement* verweist (vgl. Zeile 24), sowie zweitens geprüft, ob in der Schnittstellendefinition eine *DataInputAssociation* existiert, welche mit dem Attribut *sourceRef* ebenfalls dieses *ItemAwareElement* referenziert (vgl. Zeile 25). Falls dies nicht der Fall ist, der *DataOutput* also nicht auf ein *ItemAwareElement* verweist, das bereits durch eine *DataInputAssociation* aus der Schnittstellendefinition referenziert wird, folgt die Erzeugung eines eigenen *FormField* für den *DataOutput* (vgl. Zeilen 26-33).

```
1  <xsl:template name="FlowNode2Form">
2    ...
3    <xsl:for-each select="./bpmn:ioSpecification/bpmn:dataInput">
4      <xsl:variable name="id" select="@id" />
5      <xsl:variable name="DIA"
6        select="../../bpmn:dataInputAssociation[bpmn:targetRef/text()=$id]" />
7      <xsl:variable name="DOA"
8        select="../../bpmn:dataOutputAssociation[bpmn:targetRef/text()=$DIA/bpmn:sourceRef/
                 text()]" />
9
10     <formFields>
11       <xsl:call-template name="DataInOutput2FormField">
12         <xsl:with-param name="dataInputId" select="@id" />
13         <xsl:with-param name="dataOutputId" select="$DOA/bpmn:sourceRef/text()" />
14       </xsl:call-template>
15     </formFields>
16   </xsl:for-each>
17
18   <xsl:for-each select="./bpmn:ioSpecification/bpmn:dataOutput">
19     <!-- check case 8 -->
20     <xsl:variable name="id" select="@id" />
21     <xsl:variable name="DOA"
22       select="../../bpmn:dataOutputAssociation[bpmn:sourceRef/text()=$id]" />
23     <xsl:variable name="sameItemAwareElement"
24       select="string-length($DOA/bpmn:targetRef/text())␣>␣0␣and
25 ␣␣␣␣␣␣../../bpmn:dataInputAssociation/bpmn:sourceRef/text()=$DOA/bpmn:targetRef/text()" />
26     <xsl:if test="not($sameItemAwareElement)">
27       <formFields>
28         <xsl:call-template name="DataInOutput2FormField">
29           <xsl:with-param name="dataInputId" select="" />
30           <xsl:with-param name="dataOutputId" select="@id" />
31         </xsl:call-template>
32       </formFields>
33     </xsl:if>
34   </xsl:for-each>
35 </xsl:template>
```

Listing 103: Vereinfachte Darstellung des Templates *FlowNode2Form*

Final werden die *FormFields* im Zielmodell mit dem Template *DataInOutput2FormField* attributiert. Dies betrifft jeweils primär den Namen und die EReferences *read* zum *DataInput* bzw. *write* zum *DataOutput* (vgl. Listing 104). Dazu werden die Surrogatschlüssel rekonstruiert und unter den gleichnamigen Attributen der XML-Elemente gespeichert.

```
<xsl:template name="DataInOutput2FormField">
  <xsl:param name="dataInputId" />
  <xsl:param name="dataOutputId" />
  ...
  <xsl:if test="string-length($dataInputId) > 0">
    <xsl:attribute name="read"><xsl:value-of
      select="concat($filename, '_DATAINPUT_', $dataInputId)" /></xsl:attribute>
  </xsl:if>

  <xsl:if test="string-length($dataOutputId) > 0">
    <xsl:attribute name="write"><xsl:value-of
      select="concat($filename, '_DATAOUTPUT_', $dataOutputId)" /></xsl:attribute>
  </xsl:if>
</xsl:template>
```

Listing 104: Vereinfachte Darstellung des Templates *DataInOutput2FormField*

6.5.2 Transformationsdefinition *TaskUIGeneric2TaskUIActiviti*

Die Transformationsdefinition *TaskUIGeneric2TaskUIActiviti* bildet in Form einer QVTO-Transformation Elemente des Metamodells von *TaskUI Generic* auf solche der Metamodelle von *WASL JavaEE* ab und überbrückt die Abstraktionslücke zwischen *TaskUI Generic*-Modellen als PIMs und *WASL JavaEE*-Modellen als PSMs für die *Activiti BPM Platform*. Die Wahl der Metamodelle von *WASL JavaEE* ist damit motiviert, dass die *Activiti BPM Platform* in Java implementiert ist und entsprechend Programmierschnittstellen in Java bietet. Da die plattformspezifischen Metamodelle aus denjenigen von *WASL JavaEE* stammen, weisen die Mappings der QVTO-Transformation Ähnlichkeiten zu denen der plattformspezifischen QVTO-Transformationen des WASL-Generatorframeworks auf. Um die Vorteile einer geringen Abstraktionslücke zwischen den plattformspezifischen Metamodellen von *WASL JavaEE* und der Plattform in Form der *Activiti BPM Platform* zu illustrieren, liegt in den folgenden Ausführungen der Fokus darauf, wie die Metamodelle von *WASL JavaEE* für die Zwecke der Transformation wiederverwendet werden.

Für die QVTO-Transformation werden das Metamodell von *TaskUI Generic* unter dem Modelltyp *GENERIC* sowie eine Auswahl an Metamodellen von *WASL JavaEE* eingebunden (vgl. Listing 105). Über die Metamodelle *Filesystem*, *WaslJavaEE*, *Java6*, *XHTML1-Strict_XSD_Adapted* und *XHTML1File* hinaus werden keine weiteren plattformspezifischen Metamodelle benötigt, sodass auch der Codegenerator aus dem WASL-Generatorframework ohne Modifikationen wiederverwendet werden kann. Die Transformation erwartet ein GENERIC-Modell und gibt ein Modell vom Typ WASLJAVAEE zurück. Entsprechend ruft die Main-Operation der Transformation für jedes Wurzel-Modellelement vom Typ *TaskUIGenericModel* das Mapping *TaskUIGenericModel::toWaslJavaEEModel* auf, das ein Modellelement vom Typ *WaslJavaEEModel* liefert.

Das Mapping *TaskUIGenericModel::toWaslJavaEEModel* legt ein kompaktes Dateisystemmodell an (vgl. Listing 106), das den Konventionen von Eclipse-Projekten für Activiti folgt. Für jedes der *BPMNFiles* aus dem *TaskUIGenericModel* wird im *Directory forms* ein

```
1  import TaskUIGeneric;
2  modeltype GENERIC uses 'taskuigeneric';
3  modeltype FILESYSTEM uses 'filesystem';
4  modeltype WASLJAVAEE uses 'wasljavaee';
5  modeltype JAVA6 uses 'java6';
6  modeltype XHTML1 uses 'http://www.w3.org/1999/xhtml';
7  modeltype XHTML1FILE uses 'xhtml1file';
8
9  transformation WaslGeneric2WaslJavaEE(in waslgeneric : GENERIC, out WASLJAVAEE);
10
11 main() {
12   waslgeneric.rootObjects()[GENERIC::TaskUIGenericModel]->map toWaslJavaEEModel();
13 }
```

Listing 105: Transformation *WaslGeneric2WaslJavaEE*

eigenes *Directory* durch Aufruf des Mappings *BPMNFile::toBPMNFileDirectory* angelegt (vgl. Zeile 14). Damit wird ähnlich zu dem *TaskUIGenericModel* auch im *WaslJavaEEModel* eine integrierte Repräsentiation mehrerer BPMN-Modelle bzw. deren Prozesse, Flow-Elemente und Formulare durch ein einziges Modell erzielt. *Entities* aus dem *TaskUIGenericModel* werden durch Aufruf des Mappings *Entity::toClassifierFile* in POJOs überführt (vgl. Zeile 8), die als Datenhaltungsklassen dienen.

```
1  mapping GENERIC::TaskUIGenericModel::toWaslJavaEEModel() : WASLJAVAEE::WaslJavaEEModel{
2    directoryElements := object FILESYSTEM::Directory{
3      name := 'src';
4      directoryElements := object FILESYSTEM::Directory{
5        name := 'main';
6        directoryElements := object FILESYSTEM::Directory{
7          name := 'java';
8          directoryElements := self.getEntities().map toClassifierFile(self);
9        };
10       directoryElements += object FILESYSTEM::Directory{
11         name := 'process';
12         directoryElements := object FILESYSTEM::Directory{
13           name := 'forms';
14           directoryElements := self.bpmnFiles.map toBPMNFileDirectory(self);
15         };
16       };
17     };
18   };
19 }
```

Listing 106: Mapping *TaskUIGenericModel::toWaslJavaEEModel*

Ein *BPMNFile*-spezifisches *Directory* erhält für jeden *Process* jeweils ein Sub-*Directory* (vgl. Listing 107). In diesem wird für jeden *FlowNode* eines *Process* bzw. für jedes seiner *Forms* durch Aufruf des Mappings *Form::toFormFile* ein *XHTMLFile* angelegt, mit dem das jeweilige Formular implementiert wird (vgl. Listing 108).

```
1  mapping GENERIC::BPMNFile::toBPMNFileDirectory(in model : GENERIC::TaskUIGenericModel) :
           FILESYSTEM::Directory{
2    name := self.filename.replace(self.extension, 'bpmn20.xml');
3    directoryElements := self.processes.map toProcessDirectory(model);
4  }
```

Listing 107: Mapping *BPMNFile::toBPMNFileDirectory*

Jedes *XHTMLFile* erhält ein *DocumentRoot*-Element, dessen Inhalte entsprechend den

```
1  mapping GENERIC::Process::toProcessDirectory(in model : GENERIC::TaskUIGenericModel) :
           FILESYSTEM::Directory{
2    name := self.name.toDirectoryElementName();
3    directoryElements := self.flowNodes.form.map toFormFile(model);
4  }
```

Listing 108: Mapping *Process::toProcessDirectory*

Konventionen von Activiti erstellt werden (vgl. Listing 109). Die Eingabefelder der Formulare werden aus den *FormFields* des *TaskUI Generic*-Modells in Abhängigkeit von dem jeweils zugrunde liegenden *PrimitiveProperty* abgeleitet.

```
1  mapping GENERIC::presentation::Form::toFormFile(in model : GENERIC::TaskUIGenericModel) :
           XHTML1FILE::XHTMLFile{
2    name := self.name;
3    extensionName := 'form';
4    documentRoot := object XHTML1::DocumentRoot{
5      div := object XHTML1::DivType{
6        h1 := object XHTML1::H1Type{
7          text := object XHTML1::TextWrapper{text := self.name;}
8        };
9        ...
10       p += self.formFields.getInputWrapper(model);
11     };
12   };
13 }
```

Listing 109: Ausschnitt aus dem Mapping *Form::toFormFile*

Für *FormFields*, deren EReferences *write* oder *read* beispielsweise auf ein *DataInput* bzw. *DataOutput* mit einem *PrimitiveProperty* vom Typ *String* verweisen, werden mit dem Mapping *FormField::toInputWrapper_InputType* Formularfelder zur Texteingabe erstellt (vgl. Listing 110). Im Detail werden diese entsprechend den Konventionen von Activiti angelegt, z. B. mit Variablennamen in Camelcase-Notation und Typauszeichnungen durch versteckte XHTML-Inputelemente. Der Name der Zielvariable eines XHTML-Input-Elements wird über die *write*-EReference zu dem *DataOutput* des *FormFields* ermittelt, der Name der Quellvariable zur Vorbelegung des Input-Elements mit einem Wert dagegen über die EReference *read* zum *DataInput*. Falls eine der EReferences nicht belegt sein sollte, wird automatisch durch die Queries *getWriteOrRead* bzw. *getReadOrWrite* die jeweils konträre EReference gewählt.

```
1  mapping GENERIC::presentation::FormField::toInputWrapper_InputType(in model : GENERIC::
           TaskUIGenericModel) : XHTML1::PType{
2    text := object XHTML1::TextWrapper{text := self.name;};
3    input := object XHTML1::InputType1{
4      name := toCamelCase(getWriteOrRead(self).name);
5      type := XHTML1::InputType::text;
6      value := '${' + toCamelCase(getReadOrWrite(self).getExpression()) + '}';
7    };
8    input += object XHTML1::InputType1{
9      name := toCamelCase(getWriteOrRead(self).name) + '_type';
10     type := XHTML1::InputType::hidden;
11     value := primitive2String(getReadOrWrite(self).type);
12   };
13   ...
14 }
```

Listing 110: Ausschnitt aus dem Mapping *FormField::toInputWrapper_InputType*

7 Evaluation

Die beiden Generatorframeworks und der *Metagen*-Templategenerator sind Artefakte im Sinne der gestaltungsorientierten Wissenschaft, die im Anschluss an ihre Konstruktion einer Evaluation zu unterziehen sind. Die Evaluation befasst sich im Unterschied zu den kleinteiligen und repetitiven Abfolgen von Konstruktions- und Evaluationsphasen zur Entwicklung der Metamodelle und Transformationsdefinitionen (vgl. Kapitel 4.4) mit der Zweckmäßigkeit bzw. dem Nutzen, der insgesamt aus dem Zusammenspiel der Einzelkomponenten für verschiedene Aufgabenstellungen zu erzielen ist. Sie erfolgt in einem ersten Schritt als analytische Bewertung der Eigenschaften und Limitationen der Artefakte anhand qualitativer Kriterien (vgl. [HMPR04]). In einem zweiten Schritt wird die Praktikabilität der Generatorframeworks für den Einsatz in der modellgetriebenen Softwareentwicklung im Vergleich zu der klassischen quelltextorientierten Softwareentwicklung erörtert.

7.1 Eigenschaften und Limitationen der Generatorframeworks

Die Generatorframeworks entsprechen den vorab aufgestellten Gestaltungszielen (vgl. Kapitel 1 und 3.8), und implementieren vollständig die intendierte Funktionalität zur Adressierung der genannten Applikationstypen und Webplattformen. So werden parallel mehrere relevante Zielplattformen durch plattformspezifische und -unabhängige Modellierungssprachen adressiert, deren Metamodelle über Transformationsdefinitionen miteinander verbunden sind. Aufgrund des Bottom-Up-Vorgehens bei der Entwicklung des WASL-Generatorframeworks wirken sich sämtliche der vorgestellten Metamodellelemente der CIM-, PIM- und PSM-Sprachen über die Transformationsdefinitionen auf das Generat aus. Bezogen auf die einzelnen Zielplattformen ist jedes der Templatepakete bzw. jeder der Codegeneratoren i.V.m. der jeweiligen PSM-Modellierungssprache eigenständig ausführbar, sodass solche Generatorkomponenten abgespalten werden können, die projektspezifisch nicht benötigt werden. Die strukturähnlichen PSM-Sprachen dienen als Schnittstellen für die Codegeneratoren und sind zudem ein Fundament für die Entwicklung bzw. Anbindung weiterer PIM-Sprachen. Die Generatorframeworks können in vordefinierte Build-Umgebungen integriert werden, mit denen Build-Prozesse zur automatisierten Erstellung und Auslieferung von Applikationen gesteuert werden. Die Transformationen lassen sich dazu als Zwischenschritte in die Build-Prozesse einbinden, die abhängig von der Technologieauswahl weitere Schritte aufweisen, wie z. B. zur Interaktion mit Projektarchiven für Zwecke der Versionsverwaltung, zur Kompilierung, zu Tests und zur Auslieferung (Deployment). Die Ausführungsdauer der Transformationen hängt von den Eingabemodellen, Transformationsdefinitionen und Ausführungsumgebungen ab, hebt sich aber im Vergleich zu den

restlichen Schritten der Build-Prozesse nicht signifikant ab.

Das Generat wird ohne Bezüge zum Generator erzeugt, sodass es unabhängig vom Generator ausgeführt und weiterentwickelt werden kann. Im Build-Prozess äußert sich dies darin, dass Build-Schritte, die auf die Transformationen folgen, bzgl. dieser vollständig agnostisch sind. So müssen keine Laufzeitkomponenten des Generators in die Laufzeitumgebung der Applikation ausgeliefert werden. Auch kann das Generat nach einer Entfernung des Generators aus dem Build-Prozess für manuelle Weiterentwicklungen verwendet werden, da es in einer lesbaren Form erzeugt wird.

Die Generatorframeworks sind jedoch keine universell einsetzbaren Softwareenwicklungswerkzeuge, da ihr Funktionsumfang nicht die Generierung von Quelltext für beliebige Applikations- und Plattformtypen abdeckt. Die Limitationen resultieren aus der Zweckbindung der Generatorframeworks für die Erzeugung daten- und prozessbasierter Webapplikationen i.V.m. dem Verzicht auf eine feingranulare Modellierung von Programmlogik. So ist die feingranulare Modellierung algorithmisch komplexer Programmlogik oder allgemein beliebiger Codefragmente nicht möglich, da die Methodenrümpfe und einige der Templates der Präsentationsschicht nicht detailliert ausmodelliert werden. Stattdessen werden sie jeweils aggregiert als Modellelemente repräsentiert und durch vordefinierte Metamodellelemente typisiert. Folglich sind die darstellbaren Applikationstypen durch die Menge der verfügbaren Typen restringiert.

Bezogen auf die PSMs bedingt die Zweckbindung im Einzelfall, dass die Codegeneratoren projektspezifisch zu verfeinern sind, obwohl die PSMs strukturähnlich zu den Implementierungen und somit unabhängig von bestimmten Applikationstypen sind. Im Wesentlichen sind plattformspezifische Metamodelle anzulegen, deren Elemente das Metamodellelement *Method* aus dem Metamodell der verwendeten Programmiersprache spezialisieren und die applikationsspezifischen Methoden im Modell typisieren. Korrespondierend sind die M2T-Transformationsdefinitionen für die Generierung der Methodenrümpfe der zusätzlichen Metamodellelemente zu erweitern.

Bezogen auf die PIMs sind die Modellierungssprachen *WASL Generic* und *TaskUI Generic* auf sprachliche Ausdrucksmittel im intendierten Fokus der DSLs beschränkt. Die Fokussierung auf eine bestimmte Domäne bzw. auf als relevant erachtete Konzepte aus dieser limitiert zwangsläufig die im Modell abbildbare Semantik. Insofern ist eine modellgetriebene Einheitslösung zur plattformunabhängigen Modellierung sämtlicher funktionaler Details von Applikationen kaum sinnvoll realisierbar. Als alternativer Lösungsansatz wird aber die Spezifikation maßgeschneiderter applikationsspezifischer Erweiterungen der PIM-Modellierungssprachen um zusätzliche Metamodellelemente ermöglicht. Die zusätzlichen Metamodellelemente dienen der Modellierung der applikationsspezifischen Funktionalität

und können entweder Spezialisierungen bestehender Metamodellelemente wie z. B. *Logic-Tuples* oder eigenständige Metamodellelemente ohne Spezialisierungsbeziehungen sein. Der Umfang der Erweiterungen ist dabei prinzipiell und technisch unbegrenzt.

Ergänzend zu den Erweiterungen der verschiedenen Metamodelle und Transformationsdefinitionen lassen sich dem Generat zudem kollisionsfrei manuell erstellte Codefragmente hinzufügen. Zum einen werden manuelle Verfeinerungen von Methodenrümpfen innerhalb generierter Quelltext-Dateien durch geschützte Regionen ermöglicht. Zum anderen können vollständig manuell entwickelte Dateien in das Dateisystem des Generats eingebettet werden. Überschneidungen zwischen den generierten und den manuell erstellten Dateien werden dabei vermieden, da das Dateisystemmodell die Verortung des Generats im Dateisystem vor der Generierung eindeutig festlegt.

Eine Komplikation beim Einsatz der Codegeneratoren ist, dass die korrekte Instanziierung und Parametrisierung der PSM-Metamodellelemente Wissen über die nachgelagerten M2T-Transformationsdefinitionen voraussetzt. Dies betrifft sowohl die Programmierung von M2M-Transformationsdefinitionen, als auch die manuelle Erstellung von PSMs durch Modellierer, wenn die Codegeneratoren ohne vorgelagerte M2M-Transformationsdefinitionen verwendet werden. Beispielsweise müssen bei der Instanziierung der EClass *ListFacelet* aus dem Metamodell *JavaServerFaces1* diverse EAttributes und EReferences ausgeprägt werden (vgl. Kapitel 4.6.3.3). Die Parameter werden durch das Xpand-Template in den Kontext des restlichen Quelltexts eingefügt, sodass das Template am präzisesten die Auswirkungen der Parameter auf das Generat dokumentieren und dementsprechend durch den Modellierer verstanden werden muss.

7.2 Vergleich zu klassischer Softwareentwicklung

Im Vergleich zu der quelltextorientierten Softwareentwicklung weist der Einsatz der Generatorframeworks für die modellgetriebene Softwareentwicklung diverse Vor- und Nachteile bzgl. der Resultate und der Auswirkung auf die Softwareentwicklungsmethode auf. Wesentlich für das Generat ist, dass Methodenrümpfe gleicher Art im Quelltext grundsätzlich gleichförmig strukturiert sind. Dies sorgt für eine Verbesserung der Übersichtlichkeit, die aber partiell durch den schablonenhaften Charakter einiger der Methodenrümpfe konterkariert wird. So werden z. B. einige lokale Variablen der Methoden mit generischen Namen deklariert, da bei der Generierung der Methodenrümpfe dem Generator die fachliche Semantik unbekannt ist. Auch ist der generierte Quelltext weniger kompakt, da jede Klasse im generierten Quelltext grundsätzlich mittels ihres vollqualifizierten Namens referenziert wird, um zur Begrenzung der Komplexität der M2T-Transformationsdefinitionen auf einen Import der Namensräume von Klassen verzichten zu können (vgl. Kapitel 4.5.4). Abge-

sehen davon fällt die Lesbarkeit des generierten Quelltexts im Vergleich zu dem manuell entwickelten Pendant nicht schlechter aus, da ein konsistenter Programmierstil und im Speziellen Einrückungsstil eingehalten wird.

Die Art des Einsatzes der Generatorframeworks beeinflusst sowohl die Methode zur Entwicklung der Applikation, als auch auf einer übergeordneten Ebene die Methode zur Entwicklung des Generators. Die Adaption bzw. Erweiterung der Generatorframeworks ist dabei nicht vollends von der Entwicklung der eigentlichen Applikation zu trennen. So sollten Änderungswünsche am Generat zwar präferenziell in Modifikationen des Modells resultieren, der Bedarf für Anpassungen der Metamodelle und Transformationsdefinitionen ist aber abhängig vom jeweiligen Änderungswunsch nicht auszuschließen. Entsprechend verläuft die Entwicklung des Generators nicht strikt sequentiell vor der Entwicklung der Applikation, sondern zumindest teilweise parallel. Im Vergleich zu der rein quelltextorientierten Softwareentwicklung verlagert sich somit der Schwerpunkt der Entwicklung vom reinen Quelltext auf Modelle, Metamodelle, Transformationsdefinitionen und manuell gepflegten Quelltext. Damit ist eine stetige Abwägung während der Softwareentwicklung über den optimalen Angriffspunkt zur Realisierung der Änderungswünsche am Quelltext verbunden, die sich komplexitätssteigernd auswirkt.

Die Nachteile können jedoch abhängig von der Art der Applikation durch Qualitätssteigerungen und Zeitersparnis kompensiert werden. Sinnvolle Einsatzszenarien zur Entwicklung betrieblicher Anwendungssysteme mit den Generatorframeworks sind vorhanden, falls die Plattformauswahl kompatibel zu der des eingesetzten Generatorframeworks ist und die Applikationen voraussichtlich repetitive Codefragmente aufweisen, die vom Generatorframework abgedeckt werden oder diesem zumindest mit vertretbarem Aufwand hinzuzufügen sind. Das Auftreten ähnlicher Codefragmente kann sich dabei sowohl über einzelne Applikationen, als auch über mehrere Applikationen mit ähnlicher Kernfunktionalität erstrecken. Falls dies der Fall ist, resultiert die Zeitersparnis dann bei einer initialen Erstellung einer Applikation aus der beschleunigten Instanziierung der vordefinierten Codefragmente, und im Rahmen der nachfolgenden Wartung aus der automatisierten Sicherstellung der Konsistenz ähnlicher Codefragmente.

8 Zusammenfassung und Ausblick

Den Ausgangspunkt der vorliegenden Arbeit bildet der Bedarf für ein MDA-konformes Generatorframework, das die Erzeugung betrieblicher Webapplikationen für unterschiedliche Webplattformen unterstützt, und mittels der Modelltypen der MDA die diversen Zielplattformen individuell adressiert sowie in einer plattformunabhängigen Modellsicht integriert. Dabei ist eine wesentliche Anforderung an die Codegeneratoren, dass diese jeweils selbstständig und unabhängig von den restlichen Komponenten des Generatorframeworks zur Erzeugung betrieblicher Webapplikationen für ihre jeweilige Webplattform einsetzbar sind. Die Autonomie soll die Attraktivität für den Einsatz in plattformspezifischen Softwareentwicklungsprojekten erhöhen, wie sie für das Software Engineering üblich sind.

Die Arbeit führte initial in die Grundlagen der modellgetriebenen Softwareentwicklung sowie der MDA ein. Anschließend wurde auf verwandte Ansätze zur Modellierung und Generierung von Webapplikationen eingegangen, und der Bedarf an strukturähnlichen plattformspezifischen Modellen bzw. deren Modellierungssprachen herausgestellt. Als Beiträge wurden konform zu der Zielsetzung der Arbeit das WASL-Generatorframework für datenbasierte Webapplikationen, der Templategenerator *Metagen*, und das TaskUI-Generatorframework für prozessbasierter Webapplikationen vorgestellt, sowie einer Evaluation unterzogen.

Das WASL-Generatorframework deckt den Bedarf an strukturähnlichen Modellen mit seinen plattformspezifischen Metamodellen und korrespondierenden M2T-Transformationsdefinitionen zur Adressierung verschiedener Programmiersprachen und Frameworks ab. Die plattformspezifischen Metamodelle werden über ein gemeinsames Metamodell zur Repräsentation des Dateisystems integriert, sodass sie projektspezifisch beliebig und kollisionsfrei kombiniert werden können. Auch ist es möglich, nachträglich zusätzliche Generatorkomponenten für weitere Frameworks hinzuzufügen.

Methodisch sind die plattformspezifischen Metamodelle der Ausgangspunkt für die Entwicklung der vorgestellten plattformunabhängigen Modellierungssprachen *WASL Generic* und *TaskUI Generic*. Diese ermöglichen die plattformunabhängige Modellierung datenbasierter- und prozessbasierter Webapplikationen für Zwecke der Datenhaltung bzw. der Interaktion mit Geschäftsprozessen. Die Modellierungssprachen sind Bestandteile des WASL- bzw. TaskUI-Generatorframeworks und über M2M-Transformationsdefinitionen mit den Codegeneratoren verbunden. Mit diesen ist im Fall von *WASL Generic* die Generierung von Webapplikationen für die Plattformen Java EE, Python und PHP, sowie im Fall von *TaskUI Generic* die Generierung für die BPMN-Prozessausführungsumgebungen *Activiti BPM Platform* und *jBPM* möglich. Der Aufwand für die Erstellung der *WASL Generic*-Modelle wird minimiert, indem diese aus *WASL Data*-Datenmodellen erzeugt werden

können, die ihrerseits aus UML- und ER-Modellen ableitbar sind. Analog existieren Transformationsdefinitionen, um BPMN-Prozessmodelle in *TaskUI*-Modelle zu überführen.

Die Konzeption und Implementierung der Generatorframeworks spiegelt eine Anwendung der Prinzipien und Techniken der MDA wider. Konformität zu den Standards wird hergestellt, indem Ecore für die Spezifikation der Metamodelle verwendet wird, QVTO für die Formulierung der M2M-Transformationsdefinitionen und XMI für die Serialisierung der Modelle. Die Anwendung der Prinzipien drückt sich in diesem Verzicht auf proprietäre Techniken und hartcodierte Generatorkomponenten, sowie in der Systemspezifikation von Webapplikationen auf verschiedenen Abstraktionsniveaus durch die Modelltypen CIM, PIM und PSM aus.

Ein weiteres Resultat ist der Templategenerator *Metagen*, der als Nebenprodukt bzw. Werkzeug zur Lösung der ursprünglichen Aufgabenstellung die automatisierte Erzeugung von M2T-Transformationsdefinitionen für Codegeneratoren unterstützt. Er wird für Metamodelle benötigt, die aus XML-Schemadefinitionen abgeleitet sind, da diese einen verhältnismäßig großen Umfang besitzen und entsprechend umfangreiche Transformationsdefinitionen induzieren. Die Generierung von Xpand-Templates weist exemplarisch die generelle Machbarkeit nach und illustriert den Umgang mit Einschränkungen der Transformationssprache.

Als Ergebnis ist festzuhalten, dass durch die modellgetriebene Softwareentwicklung die Entwicklung von Prototypen oder Anwendungsrümpfen beschleunigt werden kann, falls durch die fachliche Modellierungssprache die benötigte Semantik abgedeckt wird. Davon kann aufgrund der Projektspezifika nur begrenzt ausgegangen werden, sodass abhängig von den semantischen Abweichungen Anpassungen am Generator regelmäßig obligatorisch sind. Im Vergleich zu der quelltextorientierten Softwareentwicklung wirkt sich die Entwicklung und Modifikation eines Generators initial in Form eines verhältnismäßig großen Aufwands und einer verlangsamten Entwicklungsgeschwindigkeit aus. Diese Defizite werden jedoch durch die strukturgleichen plattformspezifischen Metamodelle als Schnittstelle zwischen den Codegeneratoren und der plattformunabhängigen Spezifikationsschicht reduziert. So wirken die plattformspezifischen Metamodelle als konstanter Bezugsrahmen innerhalb der Modellwelt, der aufgrund der breiten Abdeckung von Plattformspezifika nur in begrenztem Umfang projektspezifisch zu adaptieren ist und von den Codegeneratoren abstrahiert. Die Anpassungen werden auf diese Weise weg von den M2T-Transformationsdefinitionen mit einseitiger Typsicherheit hin zu den M2M-Transformationsdefinitionen mit beidseitiger Typsicherheit verlagert, was sich komplexitätsreduzierend auswirkt.

Weiterer Forschungsbedarf resultiert aus den Limitationen der Modellierungssprachen und Transformationsdefinitionen, sowie aus offenen Fragestellungen:

Expansion der Codegeneratoren: Die plattformspezifischen Metamodelle des WASL-Generatorframeworks decken zwar einen breiten Funktionsumfang der Programmiersprachen und Software-Frameworks ab, sind aber im Wesentlichen auf solche Funktionalität beschränkt, die für die Implementierung der prototypischen betrieblichen Webapplikationen benötigt wird. Auch wird nur ein Ausschnitt der Software-Frameworks adressiert. Eine komplette Abdeckung sämtlicher Frameworkfähigkeiten ist zwar nicht erforderlich, um die Vorteilhaftigkeit strukturähnlicher Metamodelle exemplarisch nachzuweisen, für den praktischen Einsatz jedoch wünschenswert. Im Optimalfall sollte für jedes Software-Framework ein MDA-konformer Codegenerator in Form eines Metamodells und einer M2T-Transformationsdefinition existieren, das den friktionslosen Zugriff in der Modellwelt auf die Frameworkfunktionalität zulässt. Da eine solche Verfügbarkeit vorraussichtlich nicht durch die Hersteller der Software-Frameworks hergestellt wird, besteht Bedarf für Eigenentwicklungen aus dem Umfeld der MDA bzw. institutionell der OMG. Das Vorgehen zur Realisierung des Vorhabens kann aus Sicht der fachlichen Domänen optimiert werden, indem eine Auswahl an Software-Frameworks getroffen wird, die bei den wesentlichen Applikationstypen der Domäne verstärkt Verwendung finden und daher von gesteigerter Relevanz sind. Dies schließt die Sondierung klassischer Desktop-Anwendungen ein, die im Rahmen der Arbeit zwar nicht betrachtet wurden, deren Abdeckung durch Codegeneratoren aber prinzipiell möglich ist.

Integration weiterer plattformunabhängiger Modellierungssprachen: Die Zweckmäßigkeit der plattformspezifischen Metamodelle für unterschiedliche Applikationstypen und Plattformen kann durch die Anbindung weiterer plattformunabhängiger Modellierungssprachen über M2M-Transformationsdefinitionen evaluiert werden. Aus diesem Grund verspricht beispielsweise eine Kopplung der verschiedenen Modellierungssprachen der MDWE-Ansätze wie z. B. UWE oder WebML an die Codegeneratoren des WASL-Generatorframeworks Erkenntnisse bzgl. fehlender plattformspezifischer Metamodellelemente.

Entwicklung graphischer Notationsformen: Die Modellierungswerkzeuge der vorgestellten Generatorframeworks liegen als graphische Editoren vor, die automatisiert durch EMF generiert sind und Modifikationen an den Containment-Hierarchien der Modelle ermöglichen. Da jeder Editor die syntaktischen Restriktionen der Metamodelle einbezieht und kontextsensitiv valide Wertebereiche für die Belegung von EAttributes und EReferences vorgibt, wird während der Modellierung die Konsistenz der Modelle zu ihren Metamodellen erzielt. Als alternative konkrete Syntax bietet

sich vor allem für die Modellierungssprachen *WASL Data*, *WASL Generic* und *TaskUI Generic* die Entwicklung konkreter Syntaxen bzw. graphischer Notationsformen an, um die Modellierung z. B. anhand von Diagrammen zu ermöglichen. Diese sind zwar aufgrund der beschriebenen automatisiert erzeugten Editoren nicht zwingend notwendig, vereinfachen aber unter Umständen die Modellierung, indem sie die perspektivische Betonung von Modellausschnitten ermöglichen. Für die PSMs ist eine Darstellung in Diagrammform aufgrund der Vielzahl der Modellelemente und des hohen Vernetzungsgrads zwischen diesen zwar nur begrenzt sinnvoll, die Modellierung überschaubarer Ausschnitte wie z. B. zur Spezifikation von Datenbankschemata kann hingegen zweckdienlich sein. Für die Implementierung der graphischen Editoren existiert im Kontext von EMF das Graphical Modeling Framework (GMF).

Ersetzung von Xpand durch MOFM2T: Die MOF Model to Text Transformation Language (MOFM2T) ist als Templatesprache der OMG eine Alternative zu Xpand für die Formulierung von M2T-Transformationsdefinitionen. MOFM2T wird durch einen vergleichsweise neuen MDA-Standard aus dem Jahr 2008 spezifiziert [Obj08b] und im Rahmen des Eclipse-Projekts *Model to Text (M2T)* seit dem Jahr 2010 durch das Werkzeug *Acceleo 3* implementiert. Da die Strukturähnlichkeit der Metamodelle zu den Zielplattformen M2T-Transformationsdefinitionen mit geringer Komplexität induziert, sind die Anforderungen an die Transformationssprache ebenfalls begrenzt. Insofern ist eine Substitution von Xpand durch MOFM2T voraussichtlich zwar zeitintensiv, aber wenig komplex.

Ausbau von Metagen: Die Funktionalität des Templategenerators Metagen deckt die automatisierte Erzeugung von Xpand-Templates zur Generierung von XML-Dokumenten ab. Weiterentwicklungen versprechen Mehrwert sowohl bzgl. der unterstützten Transformationssprachen, als auch hinsichtlich der Dokumententypen. Die Unterstützung von MOFM2T als zusätzlicher Transformationssprache verspricht Potential für eine beschleunigte Substitution von Xpand. Als weitere Dokumententypen sind solche zu erwägen, deren korrespondierenden Metamodelle eine zu XML vergleichbar strukturierte konkrete Syntax aufweisen. Kandidaten dafür sind beispielsweise Schemata für YAML-, sowie JSON- und INI-Dokumente.

Unterstützung von Reverse Engineering: Die Strukturähnlichkeit der PSMs zu den Implementierungen vereinfacht die Entwicklung von T2M-Transformationsdefinitionen, um invers zu den Codegeneratoren aus Implementierungen PSMs abzuleiten. Die in die Analyse eingehenden Dateien der Implementierung können initial anhand ihrer Dateiendungen klassifiziert und entsprechend im Modell des Dateisystems typisiert

werden. Abhängig vom Typ der Datei gestaltet sich die Klassifizierung der Inhalte unterschiedlich komplex. So sind beispielsweise bei XML-Dokumenten die XML-Elemente über ihre XML-Tags eindeutig typisiert und gleichartig im Modell abzubilden. Konträr dazu sind z. B. Java-Klassen und -methoden zwar als solche im Quelltext gekennzeichnet, jedoch beispielsweise im Fall von Datenzugriffsklassen unzureichend für die präzise Ermittlung der passenden Metamodellelemente typisiert, sodass weitere Merkmale wie Annotationen, Namen und charakteristische Muster auszuwerten sind.

Literatur

[04.04a] United States Patent and Trademark Office: *US Trademark No. 78289148*. http://tdr.uspto.gov/search.action?sn=78289148, 2004

[04.04b] Eclipse Foundation: *XML Schema to Ecore Mapping*. http://www.eclipse.org/modeling/emf/docs/overviews/XMLSchemaToEcoreMapping.pdf, Juni 28, 2004

[06.06] Bray, Tim (Hrsg.) ; Hollander, Dave (Hrsg.) ; Layman, Andrew (Hrsg.) ; Tobin, Richard (Hrsg.)W3C: *Namespaces in XML 1.1 (Second Edition)*. http://www.w3.org/TR/2006/REC-xml-names11-20060816/, August 16, 2006

[09.09] *openArchitectureWare.org - Why openArchitectureWare moved to Eclipse.org*. http://www.openarchitectureware.org/staticpages/index.php/oaw_eclipse_letter_of_intent, September 20, 2009

[10.10a] PricewaterhouseCoopers LLP: Information Security Breaches Survey 2010. Infosecurity Europe, 2010. – Forschungsbericht

[10.10b] OWASP: OWASP Top 10 - 2010, The Ten Most Critical Web Application Security Risks. OWASP, April 19, 2010. – Forschungsbericht

[10.10c] Generative Software GmbH: Umfrage zu Verbreitung und Einsatz modellgetriebener Softwareentwicklung. Generative Software GmbH, Juni 18, 2010. – Forschungsbericht

[11.11a] Christey, Steve (Hrsg.): 2011 CWE/SANS Top 25 Most Dangerous Software Errors. The MITRE Corporation, September 13, 2011. – Forschungsbericht

[11.11b] IBM Corporation: *org.eclipse.emf.ecore (EMF Javadoc)*. http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/package-summary.html, Juni 21, 2011

[ABBB07] Acerbis, Roberto ; Bongio, Aldo ; Brambilla, Marco ; Butti, Stefano: WebRatio 5: An Eclipse-Based CASE Tool for Engineering Web Applications. In: Baresi, Luciano (Hrsg.) ; Fraternali, Piero (Hrsg.) ; Houben, Geert-Jan (Hrsg.): *Web Engineering* Bd. 4607. Berlin, Heidelberg : Springer, 2007, S. 501–505

[ADMR05] Aumueller, David ; Do, Hong-Hai ; Massmann, Sabine ; Rahm, Erhard: Schema and ontology matching with COMA++. In: *Proc. of SIGMOD 2005*. New York, NY, USA : ACM, 2005 (SIGMOD '05), S. 906–908

[BB10] Brambilla, Marco ; Butti, Stefano: Multi-faceted BPM. In: Fischer, Layna (Hrsg.): *BPMN 2.0 Handbook*. Future Strategies Inc., Dezember 2, 2010. – ISBN 978–0981987033, Kapitel 5, S. 73–84

[BCFM00] Bongio, Aldo ; Ceri, Stefano ; Fraternali, Piero ; Maurino, Andrea: Modeling Data Entry and Operations in WebML. In: Suciu, Dan (Hrsg.) ; Vossen, Gottfried (Hrsg.): *WebDB (Selected Papers)* Bd. 1997, Springer, 2000 (Lecture Notes in Computer Science). – ISBN 3–540–41826–1, S. 201–214

[BCFM06] Brambilla, Marco ; Ceri, Stefano ; Fraternali, Piero ; Manolescu, Ioana: Process Modeling in Web Applications. In: *ACM Transactions on Software Engineering and Methodology* 15 (2006), Nr. 4, S. 360–409

[BCFM07] *Kapitel* 9. In: Brambilla, Marco ; Comai, Sara ; Fraternali, Piero ; Matera, Maristella: *Designing Web Applications with WebML and WebRatio*. Springer, 2007 (Human-Computer Interaction Series). – ISBN 978–1846289224, S. 221–261

[BCFT06] Bozzon, Alessandro ; Comai, Sara ; Fraternali, Piero ; Toffetti Carughi, Giovanni: Conceptual Modeling and Code Generation for Rich Internet Applications. In: *ICWE '06: Proceedings of the 6th international conference on Web engineering*. New York, NY, USA : ACM, 2006, S. 353–360

[BDK04] Becker, Jörg ; Delfmann, Patrick ; Knackstedt, Ralf: Konstruktion von Referenzmodellierungssprachen - Ein Ordnungsrahmen zur Spezifikation von Adaptionsmechanismen für Informationsmodelle. In: *WIRTSCHAFTSINFORMATIK* 46 (2004), Nr. 4, S. 251–264

[BFT08] Brambilla, Marco ; Fraternali, Piero ; Tisi, Massimo: A Metamodel Transformation Framework for the Migration of WebML models to MDA. In: Koch, Nora (Hrsg.) ; Houben, Geert-Jan (Hrsg.) ; Vallecillo, Antonio (Hrsg.): *Proc. of the 4th Int. Workshop on Model-Driven Web Engineering (MDWE 2008)* Bd. 389, 2008, S. 91–105

[BK09] Busch, Marianne ; Koch, Nora: MagicUWE - A CASE Tool Plugin for Modeling Web Applications. In: *Proc. 9th Int. Conf. Web Engineering (ICWE'09)* Bd. 5648. Berlin : Springer, 2009, S. 505–508

[BKM99] Baumeister, Hubert ; Koch, Nora ; Mandel, Luis: Towards a UML Extension for Hypermedia Design. In: *UML'99 Proceedings of the 2nd international conference on The unified modeling language: beyond the standard* Bd. 1723. Berlin, Heidelberg : Springer-Verlag, 1999 (Lecture Notes in Computer Science), S. 614–629

[Boe06] Boehm, Barry: A view of 20th and 21st century software engineering. In: *International Conference on Software Engineering*, 2006, S. 12–29

[Boo93] Booch, Grady: *Object-oriented Analysis and Design with Applications*. 2. Addison-Wesley Professional, 1993. – ISBN 978–0805353402

[BR70] Buxton, J. N. (Hrsg.) ; Randell, Brian (Hrsg.): *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. NATO, Scientific Affairs Division, Brussels, Mai 1970

[BR95] Booch, Grady ; Rumbaugh, James ; Rational Software Corporation (Hrsg.): *Unified Method for Object-Oriented Development, Documentation Set, Version 0.8*. Santa Clara: Rational Software Corporation, 1995

[BRJ96] Booch, Grady ; Rumbaugh, James ; Jacobson, Ivar ; Rational Software Corporation (Hrsg.): *The Unified Modeling Language for Object-Oriented Development, Documentation Set, Version 0.9a Addendum*. Santa Clara: Rational Software Corporation, Juni 6, 1996

[Bro03] Brocke, Jan vom: *Referenzmodellierung: Gestaltung und Verteilung von Konstruktionsprozessen*. Berlin/Heidelberg, Westfälische Wilhelms-Universität Münster, Diss., 2003

[BRS95] Becker, Jörg ; Rosemann, Michael ; Schütte, Reinhard: Grundsätze ordnungsmäßiger Modellierung. In: *WIRTSCHAFTSINFORMATIK* 37 (1995), Nr. 5, S. 435–445

[Bub07] *Kapitel* 1. In: Bubenko, Janis A.: *From Information Algebra to Enterprise Modelling and Ontologies - a Historical Perspective on Modelling for Information Systems*. Berlin : Springer, 2007. – ISBN 978–3540726760

[CCVM06] Cáceres, Paloma ; Castro, Valeria de ; Vara, Juan M. ; Marcos, Esperanza: Model transformation for Hypertext Modeling on Web Information System. In: *Proc. of the 2006 ACM symposium on Applied Computing (SAC'06)*. New York, NY, USA : ACM, 2006, S. 1232–1239

[CFB00] Ceri, Stefano ; Fraternali, Piero ; Bongio, Aldo: Web Modeling Language (WebML): a modeling language for designing Web sites. In: *Computer Networks* 33 (2000), Nr. 1-6, S. 137–157

[CFB+02] Ceri, Stefano ; Fraternali, Piero ; Bongio, Aldo ; Brambilla, Marco ; Comai, Sara ; Matera, Maristella: *Designing Data-Intensive Web Applications*. San Francisco : Morgan Kaufmann, 2002. – ISBN 978–1558608436

[CH03] Czarnecki, Krzysztof ; Helsen, Simon: Classification of Model Transformation Approaches. In: *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003

[CH06] Czarnecki, Krzysztof ; Helsen, Simon: Feature-based survey of model transformation approaches. In: *IBM Systems Journal* 45 (2006), Juli 1,, Nr. 3, S. 621–645

[Che76] Chen, Peter Pin-Shan: The Entity-Relationship Model - Toward a Unified View of Data. In: *ACM Transactions on Database Systems* 1 (1976), Nr. 1, S. 9–36. ISBN 0362–5915

[CMV03] Cáceres, Paloma ; Marcos, Esperanza ; Vela, Belén: A MDA-Based Approach for Web Information System Development. In: *Proc. of Workshop in Software Model Engineering (WiSME'03)*, 2003

[Con99a] Conallen, Jim: Modeling Web Application Architectures with UML. Rational Software Corporation, Juni 1999. – Forschungsbericht

[Con99b] Conallen, Jim: Modeling Web Application Architectures with UML. In: *Communications of the ACM* 42 (1999), Oktober, Nr. 10, S. 63–70

[Con02] Conallen, Jim: *Building Web Applications With Uml*. 2. Reading, USA : Addison-Wesley, 2002. – ISBN 978–0201615777

[CP05] Chivers, Howard ; Paige, Richard: XRound: Bidirectional Transformations and Unifications Via a Reversible Template Language. In: Hartman, Alan (Hrsg.) ; Kreische, David (Hrsg.): *Model Driven Architecture - Foundations and Applications* Bd. 3748. Berlin, Heidelberg : Springer, 2005. – ISBN 978–3642026737, S. 205–219

[CP09] Chivers, Howard ; Paige, Richard F.: XRound: A reversible template language and its application in model-based security analysis. In: *Information and Software Technology* 51 (2009), Nr. 5, S. 876–893

[CW85] Cardelli, Luca ; Wegner, Peter: On Understanding Types, Data Abstraction, and Polymorphism. In: *ACM Computing Surveys* 17 (1985), S. 471–522

[Del11a] Delta Software Technology GmbH (Hrsg.): *HyperSenses - Modellgetriebene Software-Entwicklung*. Delta Software Technology GmbH, März 2011

[Del11b] Delta Software Technology GmbH (Hrsg.): *HyperSenses - Tutorial: Getting Started*. Delta Software Technology GmbH, Dezember 2011

[Del11c] Delta Software Technology GmbH (Hrsg.): *White Paper - HyperSenses*. Delta Software Technology GmbH, November 2011

[DeM79] DeMarco, Tom: *Structured analysis and system specification*. New Jersey : Prentice Hall, 1979. – ISBN 978–0138543808

[Dij72] Dijkstra, Edsger W.: The Humble Programmer. In: *Communications of the ACM* 15 (1972), Nr. 10, S. 859–866. ISBN 0001–0782

[Dij82] Dijkstra, Edsger W.: On the role of scientific thought. In: Dijkstra, Edsger W. (Hrsg.): *Selected writings on Computing: A Personal Perspective*. New York : Springer, 1982. – ISBN 0–387–90652–5, S. 60–66

[DV07] Del Fabro, Marcos D. ; Valduriez, Patrick: Semi-automatic model integration using matching transformations and weaving models. In: *Proceedings of the 2007 ACM symposium on Applied computing*. New York, NY, USA : ACM, 2007 (SAC '07), S. 963–970

[DW96] Dale, Nell ; Walker, Henry M.: *Abstract data types: specifications, implementations, and applications*. Lexington, MA, USA : D. C. Heath and Company, 1996. – ISBN 978–0669400007

[EFH+08] Efftinge, Sven ; Friese, Peter ; Haase, Arno ; Hübner, Dennis ; Kadura, Clemens ; Kolb, Bernd ; Köhnlein, Jan ; Moroff, Dieter ; Thoms, Karsten ; Völter, Markus ; Schönbach, Patrick ; Eysholdt, Moritz: *openArchitectureWare 4.3 Reference*, April 11, 2008

[EK06] Escalona, Maria J. ; Koch, Nora: Metamodeling the Requirements of Web Systems. In: Filipe, Joaquim (Hrsg.) ; Cordeiro, José (Hrsg.) ; Pedrosa, Vitor (Hrsg.) ; Aalst, Wil van d. (Hrsg.) ; Mylopoulos, John (Hrsg.) ; Rosemann, Michael (Hrsg.) ; Shaw, Michael J. (Hrsg.) ; Szyperski, Clemens (Hrsg.): *Web Information Systems and Technologies*. Berlin, Heidelberg

: Springer, 2006 (Lecture Notes in Business Information Processing). – ISBN 978–3540740629, S. 267–280

[EK08] Emam, Khaled E. ; Koru, A. G.: A Replicated Survey of IT Software Project Failures. In: *IEEE Software* 25 (2008), September, Nr. 5, S. 84–90

[EMM10] Einig, Daniel ; Müller, Klaus ; Meixner, Gerrit: Evaluation von Sprachen zur Spezifikation von Transformationen in modellbasierten Entwicklungsprozessen von Benutzungsschnittstellen. In: *Softwaretechnik-Trends* 30 (2010), Nr. 4

[EN10] Elmasri, Ramez ; Navathe, Shamkant: *Fundamentals of database systems*. 6. Addison-Wesley, 2010. – ISBN 978–0136086208

[FBNG07] Freudenstein, Patrick ; Buck, Jan ; Nussbaumer, Martin ; Gaedke, Martin: Model-driven Construction of Workflow-based Web Applications with Domain-specific Languages. In: *Proc. of 3rd International Workshop on Model-Driven Web Engineering (MDWE 2007)*, 2007, S. 215–229

[FHB06] Frasincar, Flavius ; Houben, Geert-Jan ; Barna, Peter: HPG: the Hera Presentation Generator. In: *Journal of Web Engineering* 5 (2006), Nr. 2, S. 175–200

[FL03] Fettke, Peter ; Loos, Peter: Model Driven Architecture (MDA). In: *WIRTSCHAFTSINFORMATIK* 45 (2003), Nr. 5, S. 555–559

[Fow02] Fowler, Martin: *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. – ISBN 978–0321127426

[Fow03] Fowler, Martin: *UML Distilled: A Brief Guide to the Standard Object Modeling Languange*. 3. Amsterdam : Addison-Wesley Longman, 2003. – ISBN 978–0321193681

[Fow10] Fowler, Martin: *Domain Specific Languages*. Addison-Wesley Longman Publishing Co., Inc., 2010. – ISBN 978–0321712943

[FPAP03] Fons, Joan ; Pelechano, Vicente ; Albert, Manoli ; Pastor, Oscar: Development of Web Applications from Web Enhanced Conceptual Schemas. In: *Workshop on Conceptual Modeling and the Web, ER'03* Bd. 2813. Chicago, USA : Springer, 2003 (LNCS), S. 232–245

[FPM10] Favre, Liliana M. ; Pereira, Claudia T. ; Martinez, Liliana I.: Foundations for MDA Case Tools. In: Favre, Liliana M. (Hrsg.): *Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution*. Engineering Science Reference, Februar 5, 2010. – ISBN 978–1615206490, Kapitel 13, S. 242–252

[FR07] France, Robert ; Rumpe, Bernhard: Model-driven Development of Complex Software: A Research Roadmap. In: *Proc. of 2007 Future of Software Engineering (FOSE '07)*. Washington : IEEE Computer Society, 2007, S. 37–54

[Fra03] Frankel, David S.: *Model Driven Architecture. Applying MDA to Enterprise Computing*. Indianapolis : John Wiley and Sons, 2003. – ISBN 978–0471319207

[FRH10] Freund, Jakob ; Rücker, Bernd ; Henninger, Thomans: *Praxishandbuch BPMN*. Hanser Fachbuch, 2010. – ISBN 978–3446417687

[FS08] Ferstl, Otto K. ; Sinz, Elmar. J.: *Grundlagen der Wirtschaftsinformatik*. 6. Oldenbourg Wissenschaftsverlag, 2008. – ISBN 978–3486587555

[FT10] Fraternali, Piero ; Tisi, Massimo: Multi-level tests for model driven web applications. In: *Proc. of the 10th Int. Conf. on Web Engineering (ICWE'10)*. Berlin, Heidelberg : Springer-Verlag, 2010, S. 158–172

[Fug93] Fuggetta, Alfonso: A classification of CASE technology. In: *Computer* 26 (1993), Dezember, Nr. 12, S. 25 – 38

[FVR+03] Fons, Joan ; Valderas, Pedro ; Ruiz, Marta ; Rojas, Gonzalo ; Pastor, Oscar: Oows: A method to develop web applications from web-oriented conceptual models. In: *International Workshop on Web Oriented Software Technology (IWWOST)*, 2003, S. 65–70

[Gar05] Garrett, Jesse J.: *Ajax: A New Approach to Web Applications*. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, Februar 18, 2005

[GC02] Gómez, Jaime ; Cachero, Cristina: OO-H Method: Extending UML to Model Web Interfaces. In: Bommel, Patrick V. (Hrsg.): *Information modeling for internet applications*. Idea Group Pub, Oktober 31, 2002. – ISBN 1–591–40050–3, Kapitel 8, S. 144 – 173

[GCP00] Gómez, Jaime ; Cachero, Cristina ; Pastor, Oscar: Extending a Conceptual Modelling Approach to Web Application Design. In: *Advanced Information Systems Engineering* Bd. 1789, 2000 (LNCS), S. 79–93

[GCP01] Gómez, Jaime ; Cachero, Cristina ; Pastor, Oscar: On Conceptual Modeling of Device-Independent Web Applications: Towards a Web Engineering Approach. In: *IEEE MultiMedia* 8 (2001), Nr. 2, S. 26–39

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Amsterdam : Addison-Wesley Longman, 1994. – ISBN 978–0201633610

[GHV10] Groenewegen, Danny M. ; Hemel, Zef ; Visser, Eelco: Separation of Concerns and Linguistic Integration in WebDSL. In: *IEEE Software* 27 (2010), Nr. 5, S. 31–37

[Gia10] Giachetti, Ronald E.: *Design of Enterprise Systems: Theory, Architecture, and Methods*. Miami : Crc Pr Inc, 2010. – ISBN 978–1439818237

[Gin98] Ginige, Athula: Web Engineering: Methodologies for Developing Large and Maintainable Web-based Information Systems. In: *Proc. of IEEE Int. Conf. on Networking the India and the World (CNI'98)*. Ahmedabad, India, Dezember 1998

[Gin01] Ginige, Athula: Web Engineering in Action. In: *Lecture Notes in Computer Science* Bd. 1016. Springer, 2001, S. 24–32

[GM01] Ginige, Athula ; Murugesan, San: Web Engineering - An Introduction. In: *IEEE MultiMedia* 8 (2001), Nr. 1, S. 14–18

[GP11] Greefhorst, Danny ; Proper, Erik: *Architecture Principles: The Cornerstones of Enterprise Architecture*. Berlin : Springer, 2011. – ISBN 978–3642202780

[GPR06] Gruhn, Volker ; Pieper, Daniel ; Röttgers, Carsten: *MDA: Effektives Softwareengineering mit UML2 und Eclipse*. Berlin : Springer, 2006. – ISBN 978–3540287445

[GR03] Gerber, Anna ; Raymond, Kerry: MOF to EMF: There and Back Again. In: Burke, Michael G. (Hrsg.): *OOPSLA Workshop on Eclipse Technology eXchange (OOPSLA2003)*. Anaheim, California : ACM-Press, 2003, S. 60–64

[Gre04] Greiffenberg, Steffen: *Methodenentwicklung in Wirtschaft und Verwaltung.* Kovac, 2004. – ISBN 978–3830012061

[Góm04] Gómez, Jaime: Model-Driven Web Development with VisualWADE. In: Koch, Nora (Hrsg.) ; Fraternali, Piero (Hrsg.) ; Wirsing, Martin (Hrsg.): *Proc. of 4th Int. Conf. on Web Engineering (ICWE 2004)* Bd. 3140. Berlin, Heidelberg : Springer, 2004 (LNCS), S. 611–612

[HKGV10] Hemel, Zef ; Kats, Lennart C. L. ; Groenewegen, Danny M. ; Visser, Eelco: Code generation by model transformation: a case study in transformation modularity. In: *Software and Systems Modeling* 9 (2010), Juni, Nr. 3, S. 375–402

[HKL95] Hirschheim, Rudy ; Klein, Heinz K. ; Lyytinen, Kalle: *Information Systems Development and Data Modeling: Conceptual and Philosophical Foundations.* Cambridge University Press, 1995. – ISBN 978–0521373692

[HMPR04] Hevner, Alan R. ; March, Salvatore T. ; Park, Jinsoo ; Ram, Sudha: Design Science in Information Systems Research. In: *MIS Quarterly* 28 (2004), Nr. 1, S. 75–105

[HN08] Hansmann, Holger ; Neumann, Stefan: Prozessorientierte Einführung von ERP-Systemen. In: Becker, Jörg (Hrsg.) ; Kugeler, Martin (Hrsg.) ; Rosemann, Michael (Hrsg.): *Prozessmanagement: Ein Leitfaden zur prozessorientierten Organisationsgestaltung.* 6. Berlin, Heidelberg : Springer, 2008, S. 329–372

[Jac92] Jacobson, Ivar: *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Professional, 1992. – ISBN 978–0201544350

[KK02] Karagiannis, Dimitris ; Kühn, Harald: Metamodelling Platforms. In: Bauknecht, Kurt (Hrsg.) ; Tjoa, A (Hrsg.) ; Quirchmayr, Gerald (Hrsg.): *E-Commerce and Web Technologies* Bd. 2455. Berlin, Heidelberg : Springer-Verlag, 2002 (LNCS). – ISBN 978–3540441373, S. 451–464

[KK03] Koch, Nora ; Kraus, Andreas: Towards a Common Metamodel for the Development of Web Applications. In: Lovelle, Juan (Hrsg.) ; Rodriguez, Bernardo (Hrsg.) ; Gayo, Jose (Hrsg.) ; Puerto Paule Ruiz, Maria del (Hrsg.) ; Aguilar, Luis (Hrsg.): *Web Engineering* Bd. 2722. Berlin, Heidelberg : Springer, 2003 (LNCS), S. 497–506

[KK08] Kroiss, Christian ; Koch, Nora: UWE Metamodel and Profile: User Guide and Reference. Ludwig-Maximilians-Universität München, Februar 2008 (0802). – Forschungsbericht

[KK09] Kroiss, Christian ; Koch, Nora: UWE4JSF: A Model-Driven Generation Approach for Web Applications. In: *Proc. 9th Int. Conf. Web Engineering (ICWE'09)* Bd. 5648. San Sebastian, Spain : Springer, Juni 2009 (LNCS), S. 493–496

[KKCM03] Koch, Nora ; Kraus, Andreas ; Cachero, Cristina ; Melià, Santiago: Modeling Web Business Processes with OO-H and UWE. In: *Proc. of 3rd. Int. Workshop on Web Oriented Software Technology (IWWOST'03)*. Oviedo, Spain, 2003

[KKCM04] Koch, Nora ; Kraus, Andreas ; Cachero, Cristina ; Melià, Santiago: Integration of business processes in web application models. In: *Journal of Web Engineering* 3 (2004), Nr. 1, S. 22–47

[KKK+07a] Kappel, Gerti ; Kargl, Horst ; Kramler, Gerhard ; Schauerhuber, Andrea ; Seidl, Martina ; Strommer, Michael ; Wimmer, Manuel: Matching Metamodels with Semantic Systems - An Experience Report. In: *Proc. of BTW Workshops 2007*. Aachen : Verlagshaus Mainz, 2007, S. 38–52

[KKK07b] Kraus, Andreas ; Knapp, Alexander ; Koch, Nora: Model-Driven Generation of Web Applications in UWE. In: Koch, Nora (Hrsg.) ; Vallecillo, Antonio (Hrsg.) ; Houben, Geert-Jan (Hrsg.): *Proc. MDWE 2007 - 3rd Int. Workshop on Model-Driven Web Engineering (CEUR-WS)* Bd. 261, 2007

[KKZB08] Koch, Nora ; Knapp, Alexander ; Zhang, Gefei ; Baumeister, Hubert: UML-based Web Engineering. An Approach based on Standards. In: Rossi, Gustavo (Hrsg.) ; Pastor, Oscar (Hrsg.) ; Schwabe, Daniel (Hrsg.) ; Olsina, Luis (Hrsg.): *Web Engineering: Modelling and Implementing Web Applications* Bd. 12. Heidelberg : Springer, 2008, Kapitel 7, S. 157–191

[KM99] Koch, Nora ; Mandel, Luis: Using UML to Design Hypermedia Applications / Ludwig-Maximilians-Universität München. 1999. – Forschungsbericht

[KMP+05] Kappel, Gerti ; Michlmayr, Elke ; Pröll, Birgit ; Reich, Siegfried ; Retschitzegger, Werner: Web Engineering - Old Wine in New Bottles. In: *Proc. of the 5th Int. Conf. on Web Engineering (ICWE 2005)* Bd. 3255, Springer, 2005

[KNS92] Keller, Gerhard ; Nüttgens, Markus ; Scheer, August-Wilhelm ; Scheer, August-Wilhelm (Hrsg.): Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK). Iwi, Januar 1992. – Forschungsbericht

[Koc01] Koch, Nora: *Software Engineering for Adaptive Hypermedia Systems*, Ludwig-Maximilians-Universität München, Diss., 2001

[Koc06] Koch, Nora: Transformation Techniques in the Model-Driven Development Process of UWE. In: *Proc. of 2nd Workshop on Model-Driven Web Engineering (MDWE'06)*, 2006

[KP88] Krasner, Glenn E. ; Pope, Stephen T.: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System / ParcPlace Systems, Inc. 1988 (3). – Forschungsbericht. – 26–49 S.

[KPRR06a] Kappel, Gerti ; Pröll, Birgit ; Reich, Siegfried ; Retschitzegger, Werner: An Introduction to Web Engineering. In: Kappel, Gerti (Hrsg.) ; Pröll, Birgit (Hrsg.) ; Reich, Siegfried (Hrsg.): *Web Engineering - The Discipline of Systematic Development of Web Applications.* John Wiley and Sons, 2006. – ISBN 978–0470015544, Kapitel 1, S. 1–21

[KPRR06b] Kappel, Gerti (Hrsg.) ; Pröll, Birgit (Hrsg.) ; Reich, Siegried (Hrsg.) ; Retschitzegger, Werner (Hrsg.): *Web Engineering - The Discipline of Systematic Development of Web Applications.* John Wiley and Sons, 2006. – ISBN 978–0470015544

[KPZM09] Koch, Nora ; Pigerl, Matthias ; Zhang, Gefei ; Morozova, Tatiana: Patterns for the Model-Based Development of RIAs. In: *Proc. 9th Int. Conf. Web Engineering (ICWE'09)* Bd. 5648. San Sebastian, Spain : Springer, Juni 2009, S. 283–291

[Kro08] Kroiss, Christian: *Modellbasierte Generierung von Web-Anwendungen mit UWE*, Ludwig-Maximilians-Universität München, Diplomarbeit, Juni 23, 2008

[KWB03] Kleppe, Anneke ; Warmer, Jos B. ; Bast, Wim: *MDA Explained: The Model Driven Architecture: Practice and Promise.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 978–0321194428

[KZE06] Koch, Nora ; Zhang, Gefei ; Escalona, Maria J.: Model Transformations from Requirements to Web System Design. In: *Proc. of the 6th Int. Conf. on Web Engineering (ICWE '06)*. New York, NY, USA : ACM, 2006, S. 281–288

[Küh01] Kühn, Harald: Ein Vergleich von Modellierungssprachen für Web-Anwendungen / Universität Wien. 2001. – Forschungsbericht

[LF06] Loos, Peter ; Fettke, Peter: Architekturen und Prozesse - Zum Beitrag empirischer Forschung in der Informationsmodellierung. In: Loos, Peter (Hrsg.) ; Krcmar, Helmut (Hrsg.): *Architekturen und Prozesse - Strukturen und Dynamik in Forschung und Unternehmen*. Berlin : Springer-Verlag, 2006, S. 33–50

[LHA06] Lopes, Denivaldo ; Hammoudi, Slimane ; Abdelouahab, Zair: Schema Matching in the Context of Model Driven Engineering: From Theory to Practice. In: Sobh, Tarek (Hrsg.) ; Elleithy, Khaled (Hrsg.): *Advances in Systems, Computing Sciences and Software Engineering*, Springer Netherlands, 2006. – ISBN 978–1–4020–5263–7, S. 219–227

[LHBJ05] Lopes, Denivaldo ; Hammoudi, Slimane ; Bézivin, Jean ; Jouault, Frédéric: Generating Transformation Definition from Mapping Specification: Application to Web Service Platform. In: *Proceedings of CAiSE'2005*, 2005, S. 309–325

[MD99] Murugesan, San ; Deshpande, Yogesh: ICSE'99 Workshop on Web Engineering. In: *Proc of the 21st Int. Conf. on Software Engineering (ICSE'99)*. Los Angeles, USA, 1999, S. 693–694

[MDHG01] Murugesan, San ; Deshpande, Yogesh ; Hansen, Steve ; Ginige, Athula: Web Engineering: A New Discipline for Development of Web-Based Systems. In: *Proc. of 23rd Int. Conf. on Software Engineering (ICSE '01)*. Toronto, Canada, 2001

[MF05] Meservy, Thomas O. ; Fenstermacher, Kurt D.: Transforming Software Development: An MDA Road Map. In: *Computer* 38 (2005), Nr. 9, S. 52–58

[MFV06] Moreno, Nathalie ; Fraternali, Piero ; Vallecillo, Antonio: A UML 2.0 Profile for WebML Modeling. In: *Proc. of. Workshop on Model-Driven Web Engineering (MDWE2006)*. Palo Alto, USA, 2006

[MFV07] Moreno, Nathalie ; Fraternali, Piero ; Vallecillo, Antonio: WebML Modeling in UML. In: *IET Software* 1 (2007), Nr. 3, S. 67 – 80

[MG06a] Melià, Santiago ; Gómez, Jaime: The WebSA Approach: Applying Model Driven Engineering to Web Applications. In: *Journal of Web Engineering* 5 (2006), Juni 1,, Nr. 2, S. 121–149

[MG06b] Mens, Tom ; Gorp, Pieter V.: A Taxonomy of Model Transformation. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), März 27,, S. 125–142

[MGPD08] Melià, Santiago ; Gómez, Jaime ; Pérez, Sandy ; Diaz, Oscar: A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA. In: *Proc. of 8th Int. Conf. on Web Engineering (ICWE '08)*. 2008, S. 13–23

[MGS07] Melià, Santiago ; Gómez, Jaime ; Serrano, José L.: WebTE: MDA Transformation Engine for Web Applications. In: Baresi, Luciano (Hrsg.) ; Fraternali, Piero (Hrsg.) ; Houben, Geert-Jan (Hrsg.): *Web Engineering* Bd. 4607. Berlin, Heidelberg : Springer, 2007, S. 491–495

[MM87] Marca, David A. ; McGowan, Clement L.: *SADT: structured analysis and design technique*. New York : McGraw-Hill, Inc., 1987. – ISBN 978–0070402355

[MMÁ+06] Martinez, Francisco Javier L. ; Molina, Fernando M. ; Álvarez, Ambrosio T. ; De Castro, Maria V. ; Cáceres, Paloma ; Marcos, Esperanza: Precise WIS development. In: *Proc. of Int. Conf. on Web Engineering 2006 (ICWE'06)*. New York, NY, USA : ACM, 2006, S. 71–76

[MPM10] Maple, Carsten ; Phillips, Alan ; Morris, Benn: UK Security Breach Investigations Report - An Analysis of Data Compromise Cases 2010. 7Safe, Januar 2010. – Forschungsbericht

[MS95] March, Salvatore T. ; Smith, Gerald F.: Design and natural science research on information technology. In: *Decision Support Systems* 15 (1995), Nr. 4, S. 251–266

[MSUW04] Mellor, Stephen J. ; Scott, Kendall ; Uhl, Axel ; Weise, Dirk: *Mda Distilled - Principles Of Model-Driven Architecture*. Reading, USA : Addison-Wesley, 2004. – ISBN 978–0201788914

[NR69] Naur, Peter (Hrsg.) ; Randell, Brian (Hrsg.): *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*. Brüssel : Scientific Affairs Division, NATO, 1969

[OAS07] OASIS (Hrsg.): *Web Services Business Process Execution Language Version 2.0*. OASIS, April 11, 2007

[Obj01] Object Management Group (Hrsg.): *Model Driven Architecture (MDA)*. Object Management Group, Juli 9, 2001

[Obj02] Object Management Group (Hrsg.): *Meta Object Facility (MOF) Specification, Version 1.4*. Object Management Group, April 1, 2002

[Obj03] Object Management Group (Hrsg.): *MDA Guide*. Object Management Group, 2003

[Obj04] Object Management Group (Hrsg.): *Human-Usable Textual Notation (HUTN) Specification*. Object Management Group, August 1, 2004

[Obj06a] Object Management Group (Hrsg.): *Business Process Modeling Notation Specification*. Object Management Group, Februar 1, 2006

[Obj06b] Object Management Group (Hrsg.): *Meta Object Facility (MOF) Core Specification, Version 2.0*. Object Management Group, Januar 1, 2006

[Obj07a] Object Management Group (Hrsg.): *MOF 2.0/XMI Mapping, V2.1.1*. Object Management Group, Dezember 1, 2007

[Obj07b] Object Management Group (Hrsg.): *UML 2.1.2 Infrastructure*. Object Management Group, November 4, 2007

[Obj08a] Object Management Group (Hrsg.): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Management Group, April 3, 2008

[Obj08b] Object Management Group (Hrsg.): *MOF Model to Text Transformation Language (MOFM2T), 1.0*. Object Management Group, Januar 2008

[Obj10a] Object Management Group (Hrsg.): *Object Constraint Language, Version 2.2*. Object Management Group, Februar 1, 2010

[Obj10b] Object Management Group (Hrsg.): *OMG Unified Modeling Language (OMG UML), Infrastructure*. Object Management Group, Mai 3, 2010

[Obj10c] Object Management Group (Hrsg.): *OMG Unified Modeling Language (OMG UML), Superstructure*. Object Management Group, Mai 5, 2010

[Obj11a] Object Management Group (Hrsg.): *BPMN v2.0*. Object Management Group, Januar 3, 2011

[Obj11b] Object Management Group (Hrsg.): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*. Object Management Group, Januar 1, 2011

[Ora11] Oracle Corporation (Hrsg.): *JSR-000315 Java Servlet 3.0 Specification 3.0 Maintenance Release*. Oracle Corporation, März 9, 2011

[PAF01a] Pastor, Oscar ; Abrahão, Silvia ; Fons, Joan: Building E-commerce applications from object-oriented conceptual models. In: *SIGecom Exch.* 3 (2001), März, Nr. 3, S. 28–36

[PAF01b] Pastor, Oscar ; Abrahão, Silvia ; Fons, Joan: An Object-Oriented Approach to Automate Web Applications Development. In: Bauknecht, Kurt (Hrsg.) ; Madria, Sanjay (Hrsg.) ; Pernul, Günther (Hrsg.): *Electronic Commerce and Web Technologies* Bd. 2115. Berlin, Heidelberg : Springer, 2001, S. 16–28

[Par04] Parr, Terence J.: Enforcing strict model-view separation in template engines. In: *Proc. of the 13th Int. Conf. on World Wide Web*. New York, NY, USA : ACM, 2004 (WWW '04). – ISBN 1–58113–844–X, S. 224–233

[PDMG08] Pérez, Sandy ; Diaz, Oscar ; Melià, Santiago ; Gómez, Jaime: Facing Interaction-Rich RIAs: the Orchestration Model. In: *Proc. of 8th Int. Conf. on Web Engineering (ICWE '08)*, 2008, S. 24–37

[PGIP01] Pastor, Oscar ; Gómez, Jaime ; Insfrán, Emilio ; Pelechano, Vicente: The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. In: *Information Systems* 26 (2001), Nr. 7, S. 507–534

[PIP+97] Pastor, Oscar ; Insfrán, Emilio ; Pelechano, Vicente ; Romero, José ; Merseguer, José: OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods. In: *Proc. of the 9th Int. Conf. on Advanced Information Systems Engineering (CAISE 1997)* Bd. 1250. Berlin : Springer, 1997 (LNCS), S. 145–158

[PLCS07] Preciado, Juan C. ; Linaje, Marino L. ; Comai, Sara ; Sánchez-Figueroa, Fernando: Designing Rich Internet Applications with Web Engineering Methodologies. In: *WSE '07: Proceedings of the 2007 9th IEEE*

International Workshop on Web Site Evolution. Washington, DC, USA : IEEE Computer Society, 2007, S. 23–30

[PW07] Pfeiffer, Daniel ; Winkelmann, Axel: Ansätze zur Wiederverwendung von Software im Rahmen der Softwareindustrialisierung am Beispiel von Softwarekomponenten, serviceorientierten Architekturen und modellgetriebenen Architekturen. In: *WIRTSCHAFTSINFORMATIK* 49 (2007), Nr. 3, S. 208–216

[Rat97] Rational Software Corporation (Hrsg.): *Unified Modeling Language 1.1 Document Set*. Rational Software Corporation, August 11, 1997

[RBP+91] Rumbaugh, James ; Blaha, Michael ; Premerlani, William ; Eddy, Frederick ; Lorensen, William ; Rumbaugh, James (Hrsg.): *Object-Oriented Modeling and Design*. New Jersey : Prentice Hall, 1991. – ISBN 978–0136298410

[RDB+08] Reichwald, Julian ; Dörnemann, Tim ; Barth, Thomas ; Grauer, Manfred ; Freisleben, Bernd: Model-Driven Process Development Incorporating Human Tasks in Service-Oriented Grid Environments. In: Bichler, Martin (Hrsg.) ; Hess, Thomas (Hrsg.) ; Krcmar, Helmut (Hrsg.) ; Lechner, Ulrike (Hrsg.) ; Matthes, Florian (Hrsg.) ; Picot, Arnold (Hrsg.) ; Speitkamp, Benjamin (Hrsg.) ; Wolf, Petra (Hrsg.): *Multikonferenz Wirtschaftsinformatik 2008*, GITO-Verlag, 2008. – ISBN 978–3940019349, S. 79–90

[Ree79a] Reenskaug, Trygve: Models - Views - Controllers / Xerox Parc. 1979. – Forschungsbericht

[Ree79b] Reenskaug, Trygve: THING-MODEL-VIEW-EDITOR / Xerox PARC. 1979. – Forschungsbericht

[Ric06] Richardson, Chris: *POJOs in Action, Developing Enterprise Applications with Lightweight Frameworks*. 2006. – ISBN 978–1932394580

[Ric11] Richardson, Robert: 2010/2011 Computer Crime and Security Survey. Computer Security Institute, 2011. – Forschungsbericht

[RKR+06] Reiter, Thomas ; Kapsammer, Elisabeth ; Retschitzegger, Werner ; Schwinger, Wieland ; Stumptner, Markus: A Generator Framework for Domain-Specific Model Transformation Languages. In: *Proc. of the 8th Int. Conf. on Enterprise Information Systems (ICEIS 2006)*, 2006, S. 27–35

[RPT+06] Ricca, Filippo ; Penta, Massimiliano D. ; Torchiano, Marco ; Tonella, Paolo ; Ceccato, Mariano: An empirical study on the usefulness of Conallen's stereotypes in Web application comprehension. In: *Proc. of the 8th IEEE Int. Symposium on Web Site Evolution*. Washington, DC, USA : IEEE Computer Society, 2006, S. 58–68

[SBD+10] Schatten, Alexander ; Biffl, Stefan ; Demolsky, Markus ; Gostischa-Franta, Erik ; Östreicher, Thomas ; Winkler, Dietmar: *Best Practice Software-Engineering : eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten*. Heidelberg : Spektrum Akademischer Verlag, 2010. – ISBN 978–3827424860

[SBPM08] Steinberg, David ; Budinsky, Frank ; Paternostro, Marcelo ; Merks, Ed: *EMF: Eclipse Modeling Framework*. 2. Amsterdam : Addison-Wesley, 2008. – ISBN 978–0321331885

[Sch01] Scheer, August-Wilhelm: *ARIS - Modellierungsmethoden, Metamodelle, Anwendungen*. 4. Berlin : Springer, 2001. – ISBN 978–3540416012

[Sch06] Schmidt, Douglas C.: Guest Editor's Introduction: Model-Driven Engineering. In: *Computer* 39 (2006), Februar, Nr. 2, S. 25–31

[SGR07] Sauer, Chris ; Gemino, Andrew ; Reich, Blaize H.: The impact of size and volatility on IT project performance. In: *Communications of the ACM* 50 (2007), Nr. 11, S. 79–84

[SH04] Stahlknecht, Peter ; Hasenkamp, Ulrich: *Einführung in die Wirtschaftsinformatik*. 11. Springer, 2004. – ISBN 978–3540011835

[SK03] Sendall, Shane ; Kozaczynski, Wojtek: Model Transformation: The Heart and Soul of Model-Driven Software Development. In: *IEEE Software* 20 (2003), S. 42–45

[SK06] Schwinger, Wieland ; Koch, Nora: Modeling Web Applications. In: Kappel, Gerti (Hrsg.) ; Pröll, Birgit (Hrsg.) ; Reich, Siegried (Hrsg.) ; Retschitzegger, Werner (Hrsg.): *Web Engineering: The Discipline of Systematic Development of Web Applications*. John Wiley and Sons, 2006. – ISBN 978–0470015544, Kapitel 3, S. 39–64

[SLCA09] Sousa, José de ; Lopes, Denivaldo ; Claro, Daniela B. ; Abdelouahab, Zair: A Step Forward in Semi-automatic Metamodel Matching: Algorithms

and Tool. In: Filipe, Joaquim (Hrsg.) ; Cordeiro, José (Hrsg.): *Proc. of 11th Int. Conf. on Enterprise Information Systems (ICEIS 2009)* Bd. 24, Springer, 2009 (Lecture Notes in Business Information Processing), S. 137–148

[Sol00] Soley, Richard: Model Driven Architecture / Object Management Group. 2000. – Forschungsbericht

[SS77] Smith, John M. ; Smith, Diane C. P.: Database abstractions: aggregation and generalization. In: *ACM Transactions on Database Systems (TODS)* 2 (1977), Nr. 2, S. 105–133

[Sta05] Staud, Josef L.: *Datenmodellierung und Datenbankentwurf. Ein Vergleich aktueller Methoden.* Berlin : Springer, 2005. – ISBN 978–3540205777

[Ste08] Stevens, Perdita: A Landscape of Bidirectional Model Transformations. In: Lämmel, Ralf (Hrsg.) ; Visser, Joost (Hrsg.) ; Saraiva, João (Hrsg.): *Generative and Transformational Techniques in Software Engineering II* Bd. 5235. Berlin, Heidelberg : Springer, 2008, S. 408–424

[SVEH07] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management.* 2. Heidelberg : Dpunkt Verlag, 2007. – ISBN 978–3898644488

[SWK06] Schauerhuber, Andrea ; Wimmer, Manuel ; Kapsammer, Elisabeth: Bridging existing Web Modeling Languages to Model-Driven Engineering: A Metamodel for WebML. In: *Proc. of Workshop of 6th Int. Conf. on Web Engineering (ICWE '06).* New York, NY, USA : ACM, 2006

[SWK+07] Schauerhuber, Andrea ; Wimmer, Manuel ; Kapsammer, Elisabeth ; Schwinger, Wieland ; Retschitzegger, Werner: Bridging WebML to Model-Driven Engineering: From DTDs to MOF. In: *IET Software* 1 (2007), Nr. 3, S. 81–97

[Tof07] Toffetti Carughi, Giovanni: Modeling data-intensive Rich Internet Applications with server push support. In: *Proc. of Model-Driven Web Engineering Workshop (MDWE'07)*, 2007

[TP06] Torres, Victoria ; Pelechano, Vicente: Building Business Process Driven Web Applications. In: Dustdar, Schahram (Hrsg.) ; Fiadeiro, José L. (Hrsg.) ; Sheth, Amit P. (Hrsg.): *Proc. of 4th Int. Conf. on Business Process Management* Bd. 4102, Springer, 2006 (LNCS), S. 322–337

[TPB+07] Trompeter, Jens ; Pietrek, Georg ; Beltran, Juan Carlos F. ; Holzer, Boris ; Kamann, Thorsten ; Kloss, Michael ; Mork, Steffen A. ; Niehues, Benedikt ; Thoms, Karsten ; Pietrek, Georg (Hrsg.) ; Trompeter, Jens (Hrsg.): *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. Frankfurt am Main : Entwickler.Press, 2007. – ISBN 978–3939084112

[TYF86] Teorey, Toby J. ; Yang, Dongqing ; Fry, James P.: A logical design methodology for relational databases using the extended entity-relationship model. In: *ACM Computing Surveys* 18 (1986), Nr. 2, S. 197–222

[VFHB03] Vdovjak, Richard ; Frasincar, Flavius ; Houben, Geert-Jan ; Barna, Peter: Engineering Semantic Web Information Systems in Hera. In: *Journal of Web Engineering* 2 (2003), Nr. 1-2, S. 3–26

[VFP05] Valderas, Pedro ; Fons, Joan ; Pelechano, Vicente: From web requirements to navigational design : A transformational approach. In: *Proc. 5th Int. Conf. Web Engineering (ICWE'05)*. Berlin : Springer, 2005 (LNCS), S. 506–511

[Vis07] Visser, Eelco: WebDSL: A Case Study in Domain-Specific Language Engineering. In: Lämmel, Ralf (Hrsg.) ; Visser, Joost (Hrsg.) ; Saraiva, João (Hrsg.): *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007* Bd. 5235. Braga, Portugal : Springer, 2007 (LNCS). – ISBN 978–3540886426, S. 291–373

[VKC+07] Vallecillo, Antonio ; Koch, Nora ; Cachero, Cristina ; Comai, Sara ; Fraternali, Piero ; Garrigós, Irene ; Gómez, Jaime ; Kappel, Gerti ; Knapp, Alexander ; Matera, Maristella ; Melià, Santiago ; Moreno, Nathalie ; Pröll, Birgit ; Reiter, Thomas ; Retschitzegger, Werner ; Rivera, José E. ; Schauerhuber, Andrea ; Schwinger, Wieland ; Wimmer, Manuel ; Zhang, Gefei: MDWEnet: A Practical Approach to Achieving Interoperability of Model-Driven Web Engineering Methods. In: Koch, Nora (Hrsg.) ; Vallecillo, Antonio (Hrsg.) ; Houben, Geert-Jan (Hrsg.): *Proc. 7th Int. Conf. Web Engineering (ICWE'07)* Bd. 261, CEUR-WS.org, 2007 (CEUR Workshop Proceedings)

[VVFP07] Valverde, Francisco ; Valderas, Pedro ; Fons, Joan ; Pastor, Oscar: A MDA-based Environment for Web Applications Development: From Conceptual Models to Code. In: *6th Int. Workshop on Web-Oriented Software Technologies (IWWOST) (2007)*, 2007, S. 164–178

[W3C02] W3C (Hrsg.): *XHTML 1.0 in XML Schema*. W3C, August 2, 2002

[W3C04a] W3C (Hrsg.): *XML Schema Part 0: Primer Second Edition*. W3C, Oktober 28, 2004

[W3C04b] W3C (Hrsg.): *XML Schema Part 1: Structures Second Edition*. W3C, Oktober 28, 2004

[W3C08] W3C (Hrsg.): *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C, November 26, 2008

[Web11] Web Models s.r.l. (Hrsg.): *WebRatio: the other way to BPM*. Web Models s.r.l., Oktober 8, 2011

[Whi04] White, Stephen A.: *Business Process Modeling Notation (BPMN) Version 1.0*, Mai 3, 2004

[Wol10] Wolffgang, Ulrich: Ein Generator-Framework zur modellgetriebenen Entwicklung von Web-Applikationen. In: *WIRTSCHAFTSINFORMATIK & MANAGEMENT* 2010 (2010), Nr. 6, S. 26–34

[WSSK07] Wimmer, Manuel ; Schauerhuber, Andrea ; Schwinger, Wieland ; Kargl, Horst: On the Integration of Web Modeling Languages: Preliminary Results and Future Challenges. In: *Proc. of 3rd Workshop on Model-driven Web Engineering (MDWE '07)*, 2007

[YC79] Yourdon, Ed ; Constantine, Larry: *Structured design. Fundamentals of a discipline of computer program and systems design*. New York : Prentice Hall, 1979. – ISBN 978–0138544713

[YK58] Young, John W. ; Kent, Henry K.: An abstract formulation of data processing problems. In: *ACM '58: Preprints of papers presented at the 13th national meeting of the Association for Computing Machinery*, ACM, 1958, S. 1–4

[Zel98] Zelnick, Nate: Nifty Technology and Nonconformance: The Web in Crisis. In: *Computer* 31 (1998), Oktober, Nr. 10, S. 115 – 116, 119

[Zim09] Zimmermann, Frank: Nutzung und Erfolg modellgetriebener Softwareentwicklung. 2009. (Arbeitspapiere der Nordakademie). – Forschungsbericht

[Zuc07] Zucker, Daniel F.: What Does Ajax Mean for You? In: *interactions* 14 (2007), Nr. 5, S. 10–12