Thema: Vom SOM-Geschäftsprozessmodell zum Softwareartefakt – modellgetriebene Systementwicklung mit dem Eclipse Modeling Framework

Masterarbeit

im Studiengang Wirtschaftsinformatik
der Fakultät Wirtschaftsinformatik
und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Verfasser: Felix Härer

Gutachter: Prof. Dr. Elmar J. Sinz

URN: urn:nbn:de:bvb:473-opus4-508089
DOI: https://doi.org/10.20378/irbo-50808

Inhalt

Abbildungsverzeichnis ... **III**

Tabellenverzeichnis ... **V**

Abkürzungsverzeichnis ... **VI**

1 Einleitung ... **1**

1.1 Problemstellung und Untersuchungsziel ... 1

1.2 Aufbau der Arbeit ... 3

2 Konzeption eines Ansatzes zur modellbasierten Transformation mit dem Eclipse Modeling Framework ... **5**

2.1 Methoden und Konzepte der modellgetriebenen Systementwicklung ... 5

2.1.1 Der Modellbegriff ... 5

2.1.2 Das modellgetriebene Vorgehensmodell der SOM-Methodik ... 6

2.1.3 Der generische Architekturrahmen ... 8

2.1.4 Die Model Driven Architecture der OMG ... 8

2.1.5 Der Begriff der modellgetriebenen Softwareentwicklung ... 10

2.2 Modellebenen vom Geschäftsprozess zum Softwareartefakt ... 12

2.2.1 Allgemeine Beschreibung der Modellebenen ... 12

2.2.2 Allgemeine Beschreibung der Modelltransformationen ... 14

2.3 Spezifikation von Metamodellen und Modellen mit Eclipse ... 16

2.3.1 Technologien zur Modellierung mit Eclipse ... 16

2.3.2 Modellierung mit Xtext und Ecore ... 18

2.4 Spezifikation von Modelltransformationen mit Eclipse ... 22

2.4.1 Technologien zur Modelltransformation mit Eclipse ... 22

2.4.2 Spezifikation der Transformation mit QVTo und Xtend ... 25

2.5 Eine textuelle Notation zur Modellierung der Anwendungssystemspezifikation der SOM-Methodik ... 28

2.5.1 Die Xtext-Grammatik der Anwendungssystemspezifikation 29

2.5.2 Das Ecore-Metamodell der Anwendungssystemspezifikation ... 32

2.5.3 Erweiterung zur Modellierung des konzeptuellen Objektschemas und des Vorgangsobjektschemas 32

3 Ein konkreter Ansatz zur modellgetriebenen Entwicklung serviceorientierter Systeme 35

3.1 Gesamtüberblick über alle Modellebenen 35

3.2 Grundlagen der serviceorientierten Architektur 37

3.3 Gestaltung der Metamodelle 39

3.3.1 Gestaltung der fachlichen Ebene serviceorientierter Anwendungssysteme 39

3.3.2 Gestaltung des Softwaredesigns auf softwaretechnischer Ebene 42

3.4 Spezifikation der Modelltransformationen 46

3.4.1 Von der Spezifikation serviceorientierter Anwendungssysteme zum Softwaredesign 46

3.4.2 Vom Softwaredesign zum Softwareartefakt 51

4 Fallstudie im Rahmen eines e-Car-Szenarios 55

4.1 Einführung in das Teilszenario „Absatz von e-Cars“ 55

4.2 Modellierung der Geschäftsprozessebene 56

4.3 Spezifikation von Anwendungssystemen 58

4.3.1 Entwicklung des konzeptuellen Objektschemas und des Vorgangsobjektschemas 58

4.3.2 Entwicklung des Servicemodells 64

4.4 Ableitung des Softwaremodells: Design 69

4.5 Generierung von Softwareartefakten 74

5 Zusammenfassung und Diskussion 84

Anhang 87

Literaturverzeichnis 93

Erklärung 98

Abbildungsverzeichnis

Abbildung 1: Kernaktivitäten des Lösungsverfahrens der Systementwicklung (Ferstl und Sinz 2013, S. 499) 1
Abbildung 2: Betrachtete Modellebenen 2
Abbildung 3: Metamodelltransformation gemäß MDA (OMG 2003a, S. 3-9) mit Anmerkungen (blau) 2
Abbildung 4: Vorgehensmodell der SOM-Methodik 7
Abbildung 5: Modellebenen des Ansatzes 13
Abbildung 6: Metamodell-Transformationen gemäß MDA-Pattern............... 14
Abbildung 7: Modellierung mit Xtext und Ecore 19
Abbildung 8: ATL-Transformationsmodell (Jouault und Kurtev 2005, S. 2) . 23
Abbildung 9: Beispielhafte QVTo-Transformation 26
Abbildung 10: Beispiel für einen einfachen Xtend-Template-Ausdruck........ 27
Abbildung 11: Xtext-Grammatik für die Spezifikation von Anwendungssystemen (Teil 1)... 29
Abbildung 12: Xtext-Grammatik für die Spezifikation von Anwendungssystemen (Teil 2)... 31
Abbildung 13: Resultierendes Ecore-Metamodell 32
Abbildung 14: Erweiterte Grammatik für KOS und VOS 33
Abbildung 15: AwS-Ecore-Metamodell mit KOS und VOS........................... 33
Abbildung 16: Beispiel zur textuellen Modellierung 34
Abbildung 17: Anwendungsbeispiel der textuellen Modellierungssprache ... 34
Abbildung 18: Abbildung aller Transformationsebenen............................... 35
Abbildung 19: SOA-Architekturmodell (Teusch und Sinz 2011, S. 294) 39
Abbildung 20: SOA-AwS-Ecore-Metamodell.. 40
Abbildung 21: ADK-Strukturmodell 43
Abbildung 22: Architekturmodell für Java EE 44
Abbildung 23: Metamodell des Softwaredesigns 45
Abbildung 24: Beziehungen zwischen den Metamodellen SOA-AwS-Spezifikation (1) und Softwaredesign (2) 46

Abbildung 25: Beginn der QVTo-Transformationsspezifikation 47
Abbildung 26: Zuordnungsfunktionen für Entitäts-Services 47
Abbildung 27: Zuordnungsfunktionen für Vorgangs-Services 48
Abbildung 28: Zuordnung nicht-elementarer Vorgangs-Services zu Stateful Session Beans 50
Abbildung 29: Schema zur Generierung von Softwareartefakten 52
Abbildung 30: Interaktionsschema zum Absatz von e-Cars (Leunig et al. 2011, S. 25) 56
Abbildung 31: Interaktionsschema zum Absatz von e-Cars, Zerlegung (Leunig et al. 2011, S. 25) 56
Abbildung 32: Initiales konzeptuelles Objektschema 59
Abbildung 33: Resultierendes konzeptuelles Objektschema 60
Abbildung 34: Konzeptuelles Objektschema in textueller Notation 61
Abbildung 35: Resultierendes Vorgangsobjektschema 62
Abbildung 36: Vorgangsobjektschema zum VOS Direktvertrieb in textueller Notation 63
Abbildung 37: Service-Schnittstelle in textueller Notation 64
Abbildung 38: Identifikation von Entitäts-Services im konzeptuellen Objektschema 65
Abbildung 39: Definition von Entitäts-Services in textueller Notation 66
Abbildung 40: Identifikation von Vorgangs-Services im Vorgangsobjektschema 67
Abbildung 41: Definition von Vorgangs-Services in textueller Notation 68
Abbildung 42: Vorgangs-Service BestellungVS in textueller Notation 69
Abbildung 43: Konfiguration der QVT-Transformation 70
Abbildung 44: Statusmeldungen zur Transformation von Entitäts-Services . 71
Abbildung 45: Statusmeldungen zur Transformation von Vorgangs-Services 71
Abbildung 46: Ecore-Modell des Softwaredesigns (1) 72
Abbildung 47: Ecore-Modell des Softwaredesigns (2) 73
Abbildung 48: Zuweisung eines Datentyps für das Attribut Name 73

Abbildung 49: Werkzeug zur Codegenerierung 75
Abbildung 50: Statusausgabe bei der Generierung von Quellcode 75
Abbildung 51: Generierte Eclipse-Projekte 76
Abbildung 52: UML-Klassendiagramm eines Entitäts-Service 76
Abbildung 53: Generierte Java-Klasse eines Entitäts-Service 77
Abbildung 54: Implementierungsklasse eines Entitäts-Service 77
Abbildung 55: Generierte Klasse der Entität Kunde 78
Abbildung 56: Generierte Softwareartefakte eines Vorgangs-Service 79
Abbildung 57: UML-Klassendiagramm eines Vorgangs-Service 80
Abbildung 58: Generierte Java-Klasse eines Vorgangs-Service 80
Abbildung 59: Message Driven Bean zur asynchronen Kommunikation 81
Abbildung 60: Stateless Session Bean zur Annahme von Web-Shop-Bestellungen 82
Abbildung 61: Datenbankeinträge nach dem Anlegen von Kunden 82
Abbildung 62: Web-Client zur Bestellung von e-Cars 83
Abbildung 63: Ecore Metamodell (Eclipse 2013) 87
Abbildung 64: Grammatik des SOA-AwS-Metamodells 88
Abbildung 65: Grammatik des Softwaredesign-Metamodells 89
Abbildung 66: UML-Klassendiagramm der Templates zur Codegenerierung 90
Abbildung 67: UML-Klassendiagramm des Werkzeugs zur Codegenerierung 91
Abbildung 68: Vorgangs-Ereignis-Schema zum Absatz von e-Cars 92

Tabellenverzeichnis

Tabelle 1: Zuordnung von Meta-Objekten des SW-Designs auf Template-Klassen für SW-Artefakte 53
Tabelle 2: Umbenennung und Zusammenfassung konzeptueller Objekttypen 59
Tabelle 3: Umbenennung von Vorgangsobjekttypen 63

Abkürzungsverzeichnis

ANTLR	Another Tool for Language Recognition
ARIS	Architektur integrierter Informationssysteme
AST	Abstract Syntax Tree
ATL	Atlas Transformation Language
AwS	Anwendungssystem
BNF	Backus-Naur-Form
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CIM	Computation Independent Model
CORBA	Common Object Request Broker Architecture
CRUD	Create Read Update Delete
DBVS	Datenbankverwaltungssystem
DSL	Domain Specific Language
EBNF	Erweiterte Backus-Naur-Form
EJB	Enterprise Java Bean
EMF	Eclipse Modeling Framework
GP	Geschäftsprozess
IAS	Interaktionsschema
IS	Informationssystem
JET	Java Emitter Template
JMS	Java Messaging Service
JPA	Java Persistence API
JSP	Java Server Pages
JVM	Java Virtual Machine
KOS	Konzeptionelles Objektschema
KOT	Konzeptioneller Objekttyp
LOT	Leistungsspezifischer Objekttyp
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
MOF	Meta Object Facility
OCL	Object Constraint Language

OMG	Object Management Group
OML	Operational Mapping Language
OOT	Objektspezifischer Objekttyp
OR-Mapper	Objektrelationaler Mapper
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query View Transformation
SOA	Serviceorientierte Architektur
SOM	Semantisches Objektmodell
TCS	Textual Concrete Syntax
TMF	Textual Modeling Framework
TOT	Transaktionsspezifischer Objekttyp
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VES	Vorgangs-Ereignis-Schema
VOS	Vorgangsobjektschema
VOT	Vorgangsobjekttyp
W3C	World Wide Web Consortium
WfMS	Workflow-Management-System
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Einleitung

1.1 Problemstellung und Untersuchungsziel

Konzeptuell modellierte Geschäftsprozesse (GP) lassen sich mit Hilfe der Methodik des semantischen Objektmodells (SOM) (Ferstl und Sinz 2013, S. 194-236) zunächst in Struktur und Verhalten darstellen, wobei beide Systemmerkmale als Sichten eines Modells zu verstehen sind. Fachliche Spezifikationen für Anwendungssysteme (AwS) können anschließend durch eine wohldefinierte Modelltransformation aus dem initialen Modell abgeleitet werden. Diese Transformation orientiert sich an der Model Driven Architecture (MDA) und basiert auf der Metamodell-Abbildung: GP-Metamodell → AwS-Metamodell (Ferstl und Sinz 2013, S. 235).

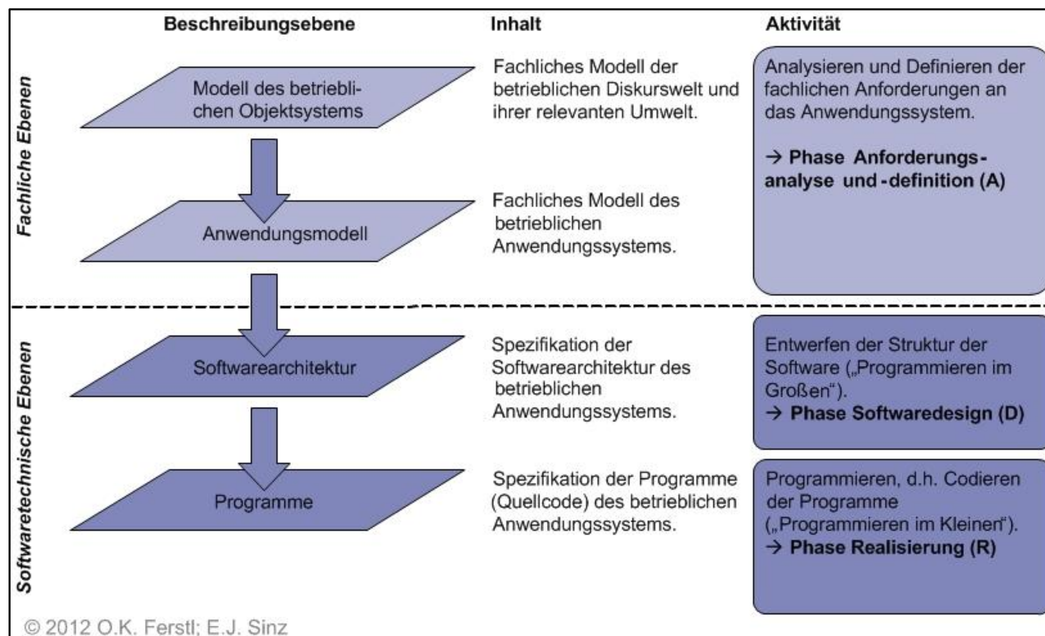

Abbildung 1: Kernaktivitäten des Lösungsverfahrens der Systementwicklung (Ferstl und Sinz 2013, S. 499)

Bei der Systementwicklung definiert das Anwendungsmodell die Spezifikation der fachlichen Anforderungen an die Anwendungssysteme (siehe Abbildung 1). Sie sind die Grundlage für den Softwareentwurf, der auf der softwaretechnischen Ebene erfolgt. In der SOM-Methodik wird eine AwS-Spezifikation mit Hilfe eins konzeptuellen Objektschemas (KOS) und eines Vorgangsobjektschemas (VOS) modelliert (Ferstl und Sinz 2013, S. 223-235). Aus Sicht der modellgetriebenen Systementwicklung kann hier abermals eine modellbasierte Transformation stattfinden: von der plattformunabhängigen AwS-Spezifikation mit KOS und VOS zu einem plattformabhängigen Modell, welches Softwareartefakte spezifiziert. Damit wäre die modellgetriebene Vorgehensweise bis in die softwaretechnische Ebene konsequent eingesetzt (siehe Abbildung 2). Die modellbasierte Transformation von

der Ebene der Geschäftsprozesse zur Ebene der Anwendungssysteme wird dabei nur am Rande betrachtet, da sie durch die SOM-Methodik bereits umfassend definiert ist.

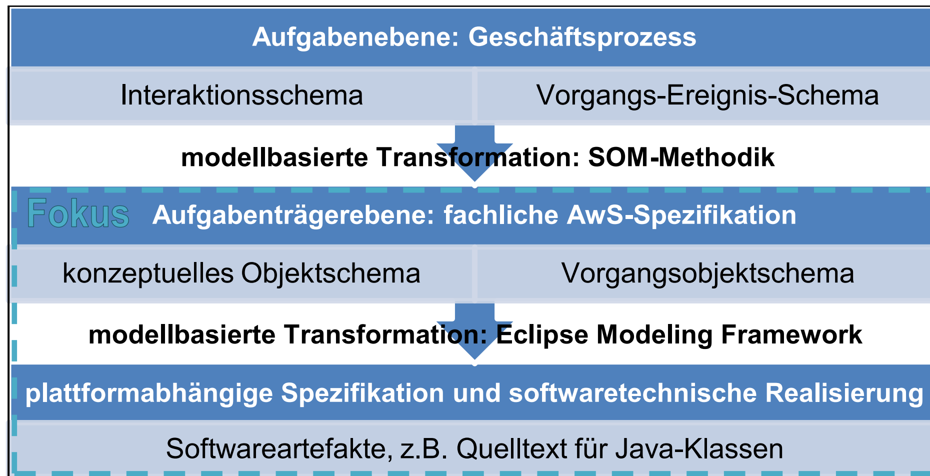

Abbildung 2: Betrachtete Modellebenen

Untersuchungsziel ist es, die semantische Lücke zwischen fachlicher und softwaretechnischer Ebene mit Methoden der modellgetriebenen Systementwicklung zu überwinden. Dies soll die hohe Komplexität der Systementwicklung besser beherrschbar machen.

Die Untersuchungsobjekte lassen sich im Kontext der metamodellbasierten Transformation darstellen (siehe Abbildung 3), die die Object Management Group (OMG) im Rahmen im Rahmen der Model Driven Architecture (MDA) standardisiert hat (2003a, S. 3-9). Die Untersuchungsobjekte sind die AwS-Spezifikation als plattformunabhängiges Modell (1), die plattformabhängige softwaretechnische Ebene mit Softwareartefakten (2), die metamodellbasierte Transformation (3) sowie die Spezifikation der Transformation (4).

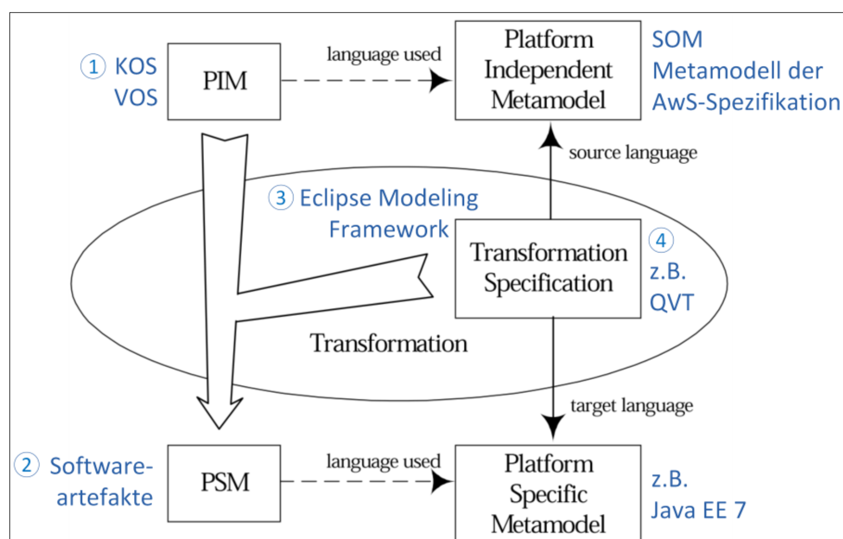

Abbildung 3: Metamodelltransformation gemäß MDA (OMG 2003a, S. 3-9) mit Anmerkungen (blau)

Das Untersuchungsobjekt (1) stellt mit KOS und VOS den Ausgangspunkt dar. Es ist konform zum Metamodell der AwS-Spezifikation des SOM. Daraus wird das plattformabhängige Modell (2) in Form von Softwareartefakten abgeleitet - z.B. Java-Klassen. Im diesem Fall ist das Softwareartefakt konform zur Spezifikation der Java Enterprise Edition (Java EE). Für die Transformation (3) wird das Eclipse Modeling Framework (EMF) betrachtet und verwendet. In diesem Open-Source-Projekt ging u.a. auch die ältere openArchitectureWare[1] auf, deren Bestandteile nun als Plattform für modellgetriebene Entwicklung innerhalb des Eclipse-Projekts genutzt werden. Es enthält umfangreiche Werkzeuge, u.a. zur Modellierung, Transformation und Generierung von Quellcode. Für die Spezifikation der Transformation (4) werden von Eclipse verschiedene Möglichkeiten angeboten - diese sollen hier ebenfalls betrachtet werden. Ein Beispiel für die Spezifikation der Transformation (4) ist der Standard Query View Transformation (QVT) der OMG (OMG 2011), mit dem sich EMF-Modelle transformieren lassen. Ein weiteres Beispiel einer Sprache zur Spezifikation der Transformation ist Xtend, mit sich aus EMF-Modellen Quellcode generieren lässt.

Die beschriebene Metamodelltransformation verdeutlicht hier die Grobstruktur der Transformation. Sie wird im Laufe der Arbeit verfeinert und mehrstufig angewendet.

1.2 Aufbau der Arbeit

Im zweiten Kapitel wird die Transformation vom Geschäftsprozess zum Softwareartefakt methodisch entwickelt. Zunächst folgt eine Betrachtung von Methoden und Konzepten, die für die Transformation im Rahmen der modellgetriebenen Entwicklung relevant sind und später in der Arbeit Anwendung finden. Anschließend werden Modellebenen und die dazugehörigen Transformationen abstrakt entwickelt, ohne eine konkrete Plattform einzubeziehen. Zur Unterstützung durch das EMF folgt anschließend eine Betrachtung verschiedener Technologien, mit denen die beschriebene Ebenen und Transformation implementiert werden können. Diese werden zum Abschluss des zweiten Kapitels verwendet, um eine textuelle Beschreibungssprache zur Modellierung der AwS-Spezifikation des SOM zu entwickeln.

Das dritte Kapitel konkretisiert den bisher nur abstrakt beschrieben Ansatz für die Entwicklung serviceorientierter Systeme. Hierzu werden die Modellebenen aus dem

[1] Siehe http://www.openarchitectureware.org/

zweiten Kapitel für die Beschreibung serviceorientierter AwS definiert. Zwei softwaretechnische Ebenen zur Beschreibung des Softwaredesigns und der Implementierung werden für die Plattform Java EE spezifiziert. Die Transformation zwischen den genannten Ebenen wird mit Hilfe des EMF und eines hierfür entwickelten Werkzeugs implementiert.

Das vierte Kapitel beinhaltet eine Fallstudie, mit der die Anwendung des entwickelten Ansatzes anhand eines e-Car-Szenarios gezeigt wird. Beginnend mit einem Geschäftsprozessmodell erfolgt die Ableitung der AwS-Spezifikation gemäß der SOM-Methodik. Nach Überarbeitung dieser Spezifikation folgt die Identifizierung von Services anhand des im vorherigen Kapitel entwickelten Modells zur Beschreibung serviceorientierter AwS. Auf Grundlage dieses Modells findet die Ableitung des Softwaredesigns mit Hilfe des EMF statt. Unter Verwendung des entwickelten Werkzeugs werden abschließend Softwareartefakte in Form von Quellcode für die Plattformspezifikation Java EE generiert.

2 Konzeption eines Ansatzes zur modellbasierten Transformation mit dem Eclipse Modeling Framework

2.1 Methoden und Konzepte der modellgetriebenen Systementwicklung

2.1.1 Der Modellbegriff

Zur Einordnung der modellgetriebenen Systementwicklung ist eine Betrachtung systemtheoretischer Grundlagen hilfreich. Insbesondere der Modellbegriff und dessen Bedeutung für die Systementwicklung sollen dabei deutlich werden.

Ferstl und Sinz (2013, S. 22) definieren ein Modell informell als „System, das ein anderes System zielorientiert abbildet“ und formal als 3-Tupel, bestehend aus einem zu modellierenden Objektsystem S_O, einem Modellsystem S_M und der Modellabbildung f. Letztere bildet Systemkomponenten V_O des Objektsystems auf Systemkomponenten V_M des Modellsystems ab. S_O kann dabei etwa ein Ausschnitt eines realen betrieblichen Systems sein, welcher zielorientiert auf S_M abgebildet wird. Besteht das Ziel in der Gestaltung neuer Anwendungssysteme, so wird die Analyse des realen Systems anhand des entsprechenden Modellsystems aufgrund der dort besser beherrschbaren Komplexität unterstützt. Allerdings muss S_O nicht notwendigerweise ein reales System sein, um zur Systementwicklung beizutragen. So ist gerade die Spezifikation von Software mit Hilfe von Modellen ein integraler Bestandteil der Systementwicklung. Die Software selbst kann dabei als formales System aufgefasst werden, welches zunächst als Modellsystem spezifiziert wird. Dies bedeutet jedoch nicht, dass auch das Modellsystem notwendigerweise formal ist, wie am Beispiel von Anforderungsdefinitionen in natürlicher Sprache deutlich wird (siehe z.B. Sommerville 2011, S. 95). Da selbst bei der Einhaltung gewisser sprachlicher Konventionen in der Regel nicht von einer exakt definierten Syntax und Semantik auszugehen ist, impliziert dies hohe Freiheitsgrade.

Demgegenüber ergeben sich bei Verwendung einer definierten Syntax in semi-formalen Modellen, wie etwa Diagrammen, weniger Freiheitsgrade. Modellbausteine und Beziehungen sind dann durch ein Metamodell definiert, ohne eine formale Semantik festzulegen. Dies ist insbesondere für die Modellierung komplexer Systeme von Vorteil, da sich so eine hinreichende Strukturierung zur

Komplexitätsbewältigung, aber auch eine gute Lesbarkeit erzielen lässt (Ferstl und Sinz 2013, S. 143).

Formale Modelle, definiert durch eine formale Syntax und Semantik, sind aufgrund ihrer inhärenten Komplexität nicht unmittelbar für die Analyse realer Systeme prädestiniert. Von Interesse sind diese Modelle indes für die reine Softwareentwicklung. Eine formalisierte Spezifikation lässt sich mittels *Model Checking* automatisiert prüfen (Clarke et al. 1999), aber auch als Basis für die Generierung lauffähiger Software heranziehen; schließlich handelt es sich bei Software, wie erwähnt, ebenfalls um ein formales Modell.

Anzumerken ist, dass ein formales Modell möglicherweise nur für Teile der Spezifikation nutzbar ist, wenn eine formalisierte Modellierung des Gesamtsystems eine zu hohe Komplexität aufweist. Somit ist eine Ableitung ausführbarer Software für diejenigen Teilsysteme möglich, welche formal in einem geeigneten Spezifikationsmodell vorliegen. Geeignet ist ein solches Modell, wenn es sich nach definierten Regeln in ein Modell transformieren lässt, das den Anforderungen der Zielplattform genügt. Dies bezieht sich allerdings nur auf die Ableitung von Softwareartefakten im Rahmen des Software Engineering, der eine konzeptuelle Entwicklung des Gesamtsystems mit höheren Freiheitsgraden vorausgehen sollte. Das semantische Objektmodell (SOM) schlägt ein solches Vorgehen im Rahmen einer umfassenden Modellierungsmethodik vor (Ferstl und Sinz 2013, S. 197-199).

2.1.2 Das modellgetriebene Vorgehensmodell der SOM-Methodik

Die SOM-Methodik basiert auf einem objekt- und geschäftsprozessorientierten Modellierungsansatz mit zugehöriger Unternehmensarchitektur (Ferstl und Sinz 2013, S. 194-236). Die Unternehmensarchitektur beinhaltet aus Außenperspektive den Unternehmensplan, aus Innenperspektive die Aufgaben- und Aufgabenträgerebene. Für eine modellgetriebene Entwicklung „vom Geschäftsprozess zum Softwareartefakt“ stellt das Geschäftsprozessmodell der Aufgabenebene den Ausgangspunkt dar. Bezüglich der Aufgabenträgerebene ist nur die Spezifikation von Anwendungssystemen relevant.

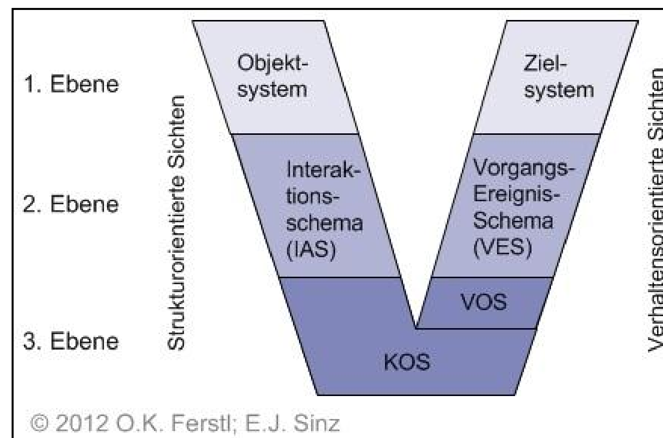

Abbildung 4: Vorgehensmodell der SOM-Methodik

Das Vorgehensmodell der SOM-Methodik sieht drei Modellebenen vor, welche jeweils eine Struktur- und Verhaltenssicht auf das Modell beinhalten (Ferstl und Sinz 2013, S. 197-199). Die Freiheitsgrade der jeweiligen Sichten einer Ebene nehmen dabei von oben nach unten ab; sie sind am Abstand der Schenkel in Abbildung 4 ablesbar. Nachdem auf oberster Ebene die Gesamtaufgabe in Form von Objektsystem und Zielsystem modelliert wurde, folgt auf der zweiten Ebene die Geschäftsprozessmodellierung. Ein solcher Prozess wird aus Struktursicht in Form eines Interaktionsschemas (IAS) mit betrieblichen Objekten und Transaktionen erfasst, aus Verhaltenssicht erfolgt eine Modellierung von Vorgangstypen und Ereignisbeziehungen in einem Vorgangs-Ereignis-Schema (VES). IAS und VES können auf dieser Ebene mit definierten Ersetzungsregeln schrittweise zerlegt werden, wobei die Regeln als Grammatik in Backus-Naur-Form (BNF) formal definiert sind. Nachdem ein ausreichender Detaillierungsgrad erreicht ist, kann der Übergang von der Aufgabenebene (2) zur Aufgabenträgerebene (3) erfolgen. Die dritte Ebene spezifiziert Anwendungssysteme, welche Lösungsverfahren für die auf Geschäftsprozessebene definierten Aufgaben implementieren, sofern eine Voll- oder Teilautomatisierung der jeweiligen Aufgabe möglich und gewünscht ist. Ein konzeptuelles Objektschema (KOS) setzt konzeptuelle Objekttypen (KOT) miteinander in Beziehung und definiert damit die Struktursicht dieser Ebene; das Verhalten ist in einem Vorgangsobjektschema (VOS) modelliert. Das VOS setzt Vorgangsobjekttypen (VOT) zueinander in Beziehung, die das Zusammenwirken der KOT beschreiben.

Bei dem mehrstufigen Vorgehensmodell der SOM-Methodik dient das Modell einer Ebene jeweils als Ausgangspunkt für die Modellierung der nächstunteren Ebene.

Somit wird ein durchgehend modellgetriebenes Vorgehen unterstützt, wobei beim Übergang von der Ebene der Geschäftsprozesse zur Ebene der Anwendungssysteme eine Ableitung des initialen KOS und VOS aus IAS und VES möglich ist (Ferstl und Sinz 2013. S. 235f.). Die Transformation ist auf Ebene der Metamodelle definiert und folgt damit der Model Driven Architecture, welche die metamodellbasierte Transformation 2003 standardisierte (OMG 2003a, S. 3-9). Beziehungsmetamodelle, die ebenfalls verschiedene Modellebenen auf Metaebene in Beziehung setzen, finden sich auch im generischen Architekturrahmen bei Sinz (2002, S. 1055-1068).

2.1.3 Der generische Architekturrahmen

Der generische Architekturrahmen dient als einheitliches Schema zur Beschreibung verschiedener Informationssystem-Architekturen (Sinz 2002, S. 1056). Nach Sinz umfasst die Architektur eines Informationssystems (IS) den Bauplan und die Konstruktionsregeln auf Ebenen der Aufgaben und Aufgabenträger. Der Bauplan stellt das Modellsystem eines IS dar (Objektsystem), die Konstruktionsregeln werden durch Metamodelle definiert. Letztere spezifizieren die im Modell verfügbaren Bausteine und Beziehungen.

Der generische Architekturrahmen erfasst dabei die Modellstruktur mit den drei Merkmalen Sichten, Metamodell und Strukturmuster. Orthogonal dazu wird außerdem die Modellhierarchie in Form von Modellebenen und deren Beziehungen erfasst. Aus den Merkmalen ergeben sich jeweils entsprechende Gestaltungsparameter für die Beschreibung konkreter Klassen von IS-Architekturen.

Ein Beziehungs-Metamodell kann verwendet werden, um „Zuordnungs- und ggf. Transformationsbeziehungen zwischen Modellebenen in Abhängigkeit von den gewählten Modellierungskonzepten zu beschreiben“ (Sinz 2002, S. 1058). Für die Methoden der modellgetriebenen Entwicklung ist die Verknüpfung verschiedener Modellebenen auf Metaebene unmittelbar relevant. Die Anwendung von Beziehungs-Metamodellen für metamodellbasierte Transformationen liegt damit nahe.

2.1.4 Die Model Driven Architecture der OMG

Die Object Management Group (OMG) verfolgt das übergeordnete Ziel, mit Hilfe von offenen plattformunabhängigen Standards Kompatibilität herzustellen und Integrationsprobleme zu lösen. Grundlegende Konzepte der modellgetriebenen

Entwicklung wurden 2001 als Model Driven Architecture (MDA) vorgestellt (OMG 2001, S. 1).

Die MDA soll die Spezifikation einer Anwendung von plattformabhängigen Implementierungsdetails trennen, indem zunächst eine plattformunabhängige Spezifikation der Anwendung erfolgt, die erst danach für eine separat spezifizierte Plattform transformiert wird. Die Ziele der MDA bestehen in höherer Portabilität, Interoperabilität und Wiederverwendbarkeit (OMG 2003a, S. 2-2).

Die Spezifikation erfolgt auf mehreren Modellebenen (OMG 2003a, S. 2-5f.). Auf oberster Ebene wird ein Computation Independent Model (CIM) als fachliches Modell verwendet, das von einem Domänenexperten modelliert wird. Das Platform Independent Model (PIM) spezifiziert die Anwendung unabhängig von der Plattform. Auf unterster Ebene implementiert das Platform Specific Model (PSM) die im PIM spezifizierte Anwendung für eine festgelegte Plattform. Die Plattform selbst kann in einem separaten Plattform-Modell hinterlegt sein.

Für den Übergang zwischen PIM und PSM sind verschiedene Möglichkeiten zur Transformation vorgesehen (OMG 2003a, S. 3-8 - 3-13):

- Beim *Marking* wird die plattformabhängige Spezifikation der Transformation für jedes einzelne Modellelement des PIM auf Instanzebene hinzugefügt (OMG 2003a, S. 3-3f.); somit entsteht ein transformierbares *marked PIM*.
- Die Metamodell-Transformation sieht Metamodelle für PIM und PSM vor und spezifiziert die Transformation als Beziehungs-Metamodell, so dass jedes konkrete PIM-Modell ohne Weiteres transformiert werden kann.
- Eine Modelltransformation ohne Metamodelle ist auf Typebene definiert. Jedes Element des PIM ist Subtyp eines plattformunabhängigen Typs, für den eine Transformation in einen plattformabhängigen Typ spezifiziert ist.
- *Patterns* können zusammen mit der typbasierten Modelltransformation oder dem *Marking* für die Transformation genutzt werden.
- *Model Merging* sieht vor, zusätzlich zum PIM weitere Modelle als Input für die Transformation zu einem PSM zu nutzen.

Die genannten Konzepte schließen sich nicht gegenseitig aus und können auch kombiniert werden (siehe z.B. OMG 2003a, S. 3-14).

Für die Modellierung werden verschiedene von der OMG standardisierte Technologien vorgeschlagen (OMG 2003a, S. 7-1). Modelle sollen mittels Unified Modeling Language (UML) modelliert werden, die in der Architektur der Meta Object Facility (MOF) eingebettet ist. Das UML-Metamodell ist zu dem Meta-Metamodell der MOF konform und lässt sich gemäß der MOF-Architektur unterhalb der Modellebene M3 (Meta-Metaebene) auf Ebene M2 (Metaebene) einordnen. Konkrete Modelle gehören zur Ebene M1, während M0 dem zu modellierenden System entspricht (OMG 2005a, S. 11f.). Weitere MOF-konforme Modelle können hier ebenfalls Anwendung finden, wie z.B. das Common Warehouse Metamodel (OMG 2003b). Des Weiteren wird die Nutzung von UML-Profilen vorgeschlagen, um in UML formulierte Modellierungssprachen zu erweitern oder einzuschränken. Bezüglich der Technologien auf Plattformebene werden verschiedene CORBA-Varianten (Common Object Request Broker Architecture) genannt (OMG 2003a, S. 7-2). Damit die von der OMG postulierte Plattformunabhängigkeit erreicht wird, kann der Einsatz bestimmter Technologien allerdings nur als Empfehlung angesehen werden.

2.1.5 Der Begriff der modellgetriebenen Softwareentwicklung

Zum Themenkomplex der modellgetriebenen Software-Entwicklung finden sich in der Literatur diverse Begriffe und Abkürzungen, die mitunter uneinheitlich verwendet werden.

Neben der englischsprachigen Bezeichnung *Model Driven Software Development* (MDSD, siehe z.B. Stahl et al. 2007, S. 11), wird auch der allgemeinere Begriff des *Model Driven Engineering* (MDE) im Zusammenhang mit modellgetriebener Entwicklung verwendet. Der letztgenannte Begriff kam nach der Veröffentlichung der MDA (OMG 2001) auf und wird in der Regel als Verallgemeinerung des MDA-Ansatzes verstanden (Kent 2002, Bézivin 2005, Schmidt 2006). Dabei schließt er den Ansatz der MDA ein, ohne dabei konkrete Technologien oder Methoden vorauszusetzen. Wie im vorherigen Abschnitt beschrieben, sieht die MDA die Verwendung der Technologien MOF und UML vor. Allerdings sind die zur Transformation definierten Muster auch ohne diese Technologien verwendbar.

Die modellgetriebene Systementwicklung entspricht dabei der allgemeineren Bezeichnung des *Model Driven Engineering*, wobei die Softwareentwicklung als ein Aspekt verstanden werden kann, der auch Bestandteil der allgemeinen Entwicklung

von Systemen ist. In der Literatur ist die Unterscheidung von System- und Softwareentwicklung jedoch in der Regel nicht zu finden, da oft Fragen der Gestaltung von Softwaresystemen im Vordergrund stehen und die Begriffe somit synonym verwendet werden.

Die modellgetriebene Softwareentwicklung wendet die in diesem Kapitel diskutierten Methoden im Rahmen des Software Engineering an. Stahl et al. definieren die modellgetriebene Softwareentwicklung als „Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen“ (2007, S. 11). Neben den bereits angesprochenen Konzepten zur Modellierung und Transformation von Metamodellen und Modellen sind hier vor allem domänenspezifische Sprachen und Werkzeuge zur Generierung lauffähiger Programme aus Modellen relevant (Stahl et al. 2007, S. 4).

Mit einer domänenspezifischen Sprache, auch DSL (Domain Specific Language), lassen sich Modelle einer bestimmten Domäne in einer dafür entwickelten Notation beschreiben. Eine textuelle Notation wird beispielsweise durch die Zuordnung einer konkreten Syntax zu einzelnen Elementen des Metamodells der Domäne definiert. Alternativ dazu kann auch eine grafische Notation angegeben werden (Stahl et al. 2007, S. 101). Im Falle der textuellen Beschreibung kann die Syntax beispielsweise anhand einer Grammatik definiert werden, für die ein Parsergenerator-Werkzeug einen entsprechenden Parser erstellt (Stahl et al. 2007, S. 104-109), mit dem sich das Modell instanziieren lässt.

Für die Erstellung lauffähiger Programme werden Codegeneratoren eingesetzt, die ein Modell, welches beispielsweise in einer textuellen DSL definiert ist, in Quellcode umwandeln. Stahl et al. definieren fünf Merkmale für einen solchen Generator (2007, S. 145). Informationen aus dem Modell müssen von Generierungsvorschriften getrennt sein (1), wobei letztere auf Ebene des Metamodells definiert sind (2). Der Generator erzeugt aus einem Modell beliebig viele Artefakte für die Zielplattform (3) und kann für beliebig viele Modelle genutzt werden (4), sofern diese konform zu dem entsprechenden Metamodell sind. Erzeugter Quellcode wird von handgeschriebenem Code getrennt (5), wobei eine Integration mit den Mitteln der verwendeten Programmiersprache erreicht werden soll.

2.2 Modellebenen vom Geschäftsprozess zum Softwareartefakt

2.2.1 Allgemeine Beschreibung der Modellebenen

Eine IS-Architektur ist auf den Ebenen der Aufgaben und Aufgabenträger durch einen Bauplan des Objektsystems sowie dessen Konstruktionsregeln beschrieben (Sinz 2002, S. 1055). Ausgehend von dieser Definition sollen zunächst die Modellebenen der IS-Architektur genauer beschrieben werden, um einen Rahmen für die zu entwickelnde Transformation zu schaffen. Die Konzeption des Rahmens orientiert sich am generischen Architekturrahmen nach Sinz (2002, S. 1056f.).

Die bereits erwähnten Ebenen für Aufgaben und Aufgabenträger werden in Form einer Geschäftsprozessebene und einer Ebene zur Spezifikation der Anwendungssysteme gemäß der SOM-Methodik übernommen (Ferstl und Sinz 2013, S. 196f.). Im Rahmen der Systementwicklung handelt es sich dabei um zwei fachliche Ebenen, die oberhalb der softwaretechnischen Ebenen einzuordnen sind (Ferstl und Sinz 2013, S. 482f.).

Auf fachlicher Ebene wird das Modell des betrieblichen Objektsystems damit aus Struktursicht durch ein Interaktionsschema (IAS) und aus Verhaltenssicht durch ein Vorgangs-Ereignis-Schema (VES) beschrieben. Das Anwendungsmodell ist in Form eines konzeptuellen Objektschemas (KOS) und eines Vorgangsobjektschemas (VOS) festgelegt. Modelle und Metamodelle beider Ebenen sind durch die SOM-Methodik bereits ausreichend definiert (Ferstl und Sinz 2013, S. 200-235).

Auf softwaretechnischer Ebene werden Softwarearchitektur und Programme beschrieben (Ferstl und Sinz 2013, S. 483). Die Softwarearchitektur spezifiziert das Design der Software in Form von Subsystemen und Komponenten mit deren Beziehungen. Die Elemente zur Bildung von Subsystemen und Komponenten hängen von der eingesetzten Plattform ab, d.h. das Modell der Software muss auf Ebene des Softwaredesigns zur Plattformspezifikation für Elemente der Architektur konform sein. Die Implementierung der Programme durch Quellcode bildet die unterste Ebene, mit der einzelne Softwareartefakte spezifiziert sind. Diese müssen ebenfalls zur Plattformspezifikation konform sein.

Die diskutierten Ebenen sind in Abbildung 5 dargestellt. Bei ihrer Betrachtung wird deutlich, dass eine plattformunabhängige Modellierung nur auf fachlicher Ebene

möglich ist. Die softwaretechnische Ebene muss somit für eine spezifische Zielplattform modelliert werden. Sofern mehrere Plattformen unterstützt werden sollen oder eine ursprüngliche Plattform abgelöst werden soll, ist die softwaretechnische Ebene jeweils neu zu erstellen, wobei das fachliche Modell jeweils als Ausgangspunkt dient.

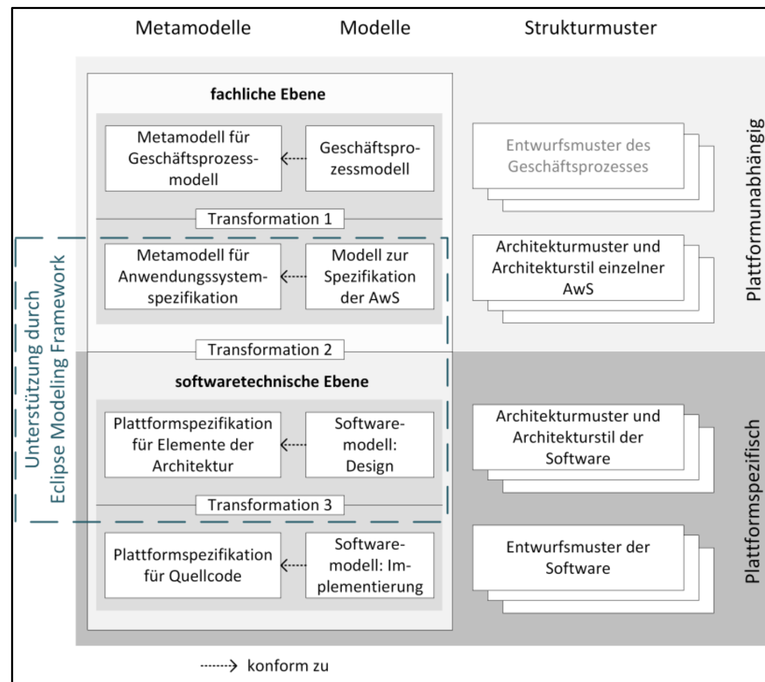

Abbildung 5: Modellebenen des Ansatzes

Zu jeder Modellebene können außerdem noch Strukturmuster angegeben werden (Sinz 2002, S. 1057), die das Zusammenwirken einzelner Elemente des Modells zur Lösung eines definierten Problems beschreiben. Auf Ebene der Implementierung bieten sich bei einer objektorientierten Plattform etwa Entwurfsmuster von Gamma et al. an (1994). Analog dazu ist bei der Gestaltung von Softwaredesign und Anwendungsmodell der Einsatz von Architekturmustern denkbar (Shaw und Garlan 1996). Außerdem kann ein bestimmter Architekturstil verfolgt werden, der ebenso Muster bzgl. der Strukturbildung vorschreibt. Ein Beispiel hierfür ist eine serviceorientierte Architektur (SOA), die u.a. die Bildung von Diensten vorsieht. Ein Beispiel für einen Architekturstil des Softwaredesigns ist etwa eine komponentenbasierte Architektur (siehe z.B. Heineman et al. 2001). Strukturmuster auf Geschäftsprozessebene (Ferstl et al. 1998) werden im vorliegenden Ansatz nicht betrachtet.

Die Unterstützung der Modellierung von Transformationen durch das Eclipse Modeling Framework (EMF) setzt auf der Ebene der AwS-Spezifikation ein. Für

Modellierung und Transformation auf fachlicher Ebene existieren bereits Werkzeuge, wie etwa ein Multi-View-Modellierungswerkzeug für die SOM-Methodik, das auf der Meta-Modellierungsplattform ADOxx basiert (Bork und Sinz 2011). Entsprechend der Zielsetzung der Arbeit wird die modellgetriebene Entwicklung von Geschäftsprozessmodell und AwS-Spezifikation mit Hilfe des EMF auf softwaretechnischer Ebene konsequent weitergeführt.

2.2.2 Allgemeine Beschreibung der Modelltransformationen

Die Spezifikation der Transformation wird anhand der Metamodelle definiert; sie ist in Abbildung 6 dargestellt. Dabei werden Meta-Objekte einer Ebene (Quell-Metamodell) auf die Meta-Objekte der nächstunteren Ebene (Ziel-Metamodell) im Sinne der Metamodell-Transformation der MDA (OMG 2003a, S. 3-9f.) abgebildet.

Die erste Modelltransformation von der Ebene der Geschäftsprozesse zur Ebene der AwS-Spezifikation ist bereits durch die SOM-Methodik vorgegeben. Aus IAS und VES lässt sich eine initiale AwS-Spezifikation mit KOS und VOS direkt ableiten. Diese Ableitung folgt als metamodellbasierte Transformation (Ferstl und Sinz 2013, S. 235).

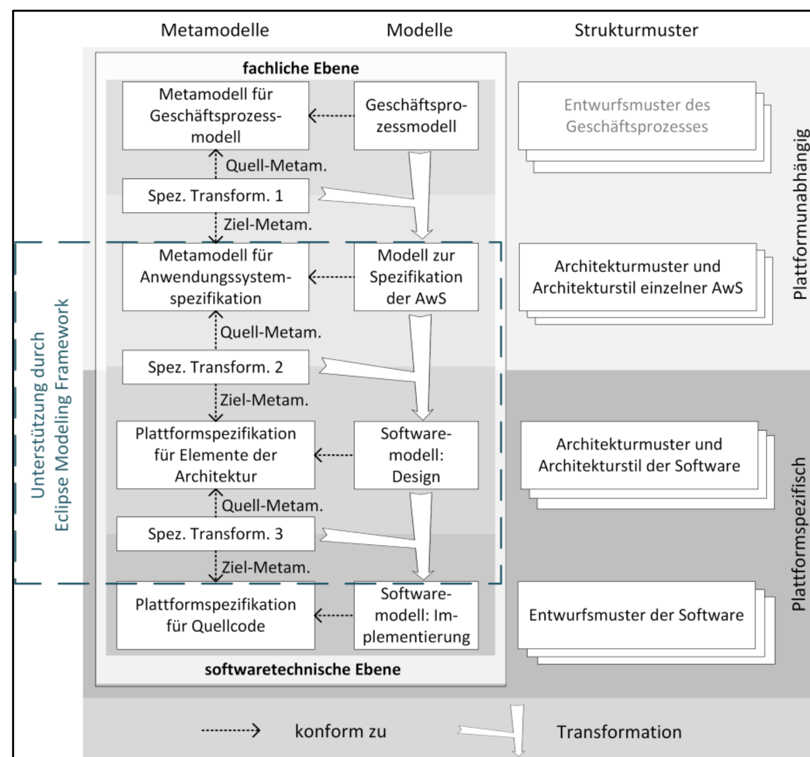

Abbildung 6: Metamodell-Transformationen gemäß MDA-Pattern

Die zweite und dritte Transformation sollen ebenfalls metamodellbasiert angegeben werden. Die zweite Transformation bildet das fachliche Modell der

Anwendungssysteme auf ein Modell der Software ab, welches das Design spezifiziert. In der dritten Transformation wird die Software auf Grundlage des Designs mittels Implementierung realisiert. Da dies die Generierung von Quellcode erfordert, wird mitunter auch von einer Modell-zu-Text-Transformation gesprochen (siehe z.B. Gronback 2009, S. 277).

Die plattformunabhängige fachliche Modellierung mit einer spezifischen softwaretechnischen Ebene erfolgt im Sinne von PIM und PSM der MDA. Eine plattformunabhängige Modellierung erfolgt dort mittels eines PIM, welches anschließend in ein plattformspezifisches PSM transformiert wird.

Die zweite und dritte Transformation sind von den angegebenen Quell- und Ziel-Metamodellen abhängig, die aufgrund der Plattformabhängigkeit der softwaretechnischen Ebene an dieser Stelle noch nicht definiert sind. Sie wird im dritten Kapitel für die Plattform Java EE spezifiziert.

2.3 Spezifikation von Metamodellen und Modellen mit Eclipse

2.3.1 Technologien zur Modellierung mit Eclipse

Zur softwaretechnischen Werkzeugunterstützung kommt im Rahmen dieser Arbeit Eclipse zum Einsatz. Die Open-Source-Software bietet im Kern eine Entwicklungsumgebung, dient aber auch als flexible Plattform für die Integration beliebiger weiterer Werkzeuge. Die Entwicklung wird von der Eclipse Foundation (Steinberg et al. 2008, S. 3), einer Non-Profit-Organisation mit über 100 beteiligten Unternehmen, unterstützt. Eclipse ist in diverse Projekte unterteilt (Steinberg et al. 2008, S. 4-5). In erster Linie ist das Eclipse-Projekt zu nennen, der Kern der Entwicklungsumgebung, die auch das Java Development Environment einschließt. Das Modeling-Projekt (Gronback 2009, S. 8-9) umfasst verschiedene Technologien zur modellgetriebenen Entwicklung mit der Eclipse-Plattform und ist in diesem Zusammenhang unmittelbar relevant.

Die zentrale Komponente des Modeling-Projekts ist das Eclipse Modeling Framework (EMF) (Steinberg et al. 2008, S. 11-38). Mit EMF lassen sich in erster Linie Modelle und Metamodelle erstellen, editieren, speichern und einlesen. Neben entsprechenden Benutzerschnittstellen bietet EMF in seiner Funktion als Framework Programmierschnittstellen an. Daher wird EMF auch als Basis für weitere Technologien des Modeling-Projekts und für andere externe Erweiterungen genutzt.

In EMF definierte Modelle und Metamodelle basieren auf Ecore, einem Metamodell (bzw. Meta-Metamodell), das selbst als EMF-Modell definiert ist (Steinberg et al. 2008, S. 17, 103-105). Das Metamodell von Ecore ist in Abbildung 63 (siehe Anhang) dargestellt. Die wichtigsten Elemente sind die Meta-Objekte *EClassifier* mit den Spezialisierungen *EClass* und *EDataType*, sowie *EStructuralFeature* in Form von *EReference* und *EAttribute*. Eine *EClass* beinhaltet beliebig viele Objekte des Typs *EStructuralFeature*. Das Metamodell eignet sich damit etwa zur Modellierung einfacher Klassen (*EClass*), die miteinander in Beziehung stehen (*EReference*). Beziehungen sind entweder Assoziations- oder Kompositionsbeziehungen (*containment*-Attribut wahr bzw. falsch), besitzen eine Kardinalität (Attribute *upper*- und *lowerBound*) und können auch bidirektional sein. Ein Klasse kann Attribute (*EAttribute*) mit einem bestimmten Datentyp (*EDataType*) enthalten.

Die Generierung von Klassen aus solchen Modellen ist damit eine naheliegende Anwendung. EMF bringt hierfür auch einen Generator mit, der template-basiert Quellcode in entsprechend formulierte Vorlagen umsetzt. Für die Formulierung von Vorlagen werden Java Emitter Templates (JET) mit JSP-Syntax verwendet (Steinberg et al. 2008, S. 341f.).

Der Code-Generator hat neben der Umwandlung von Modellen in Quellcode noch weitere Anwendungen innerhalb des EMF (Steinberg et al. 2008, S. 65-71). Für die Erstellung, Manipulation und den Zugriff auf Modellinstanzen werden entsprechende Klassen, gepackt als Edit-Plugin, generiert, welche die genannten Funktionen speziell für die generierten Modellklassen anbieten. Zur Integration der Modellanzeige und -bearbeitung in die Eclipse-Plattform wird außerdem ein Editor-Plugin generiert. Dieses bietet beispielsweise einen einfachen *Tree View Editor*, mit dem das Modell als Baumstruktur angezeigt und bearbeitet werden kann.

Der *Tree View Editor* lässt sich auch zur Erstellung neuer Modelle (Steinberg et al. 2008, S. 19, 71) auf Basis des Ecore-Metamodells verwenden. Auch ein Import ausgehend von annotierten Java-Dateien, XML-Schemata oder UML-Modellen ist möglich. Ecore-Modelle werden in dem von der OMG standardisierten (2005b) Format *XML Metadata Interchange* (XMI) abgelegt und können somit auch von anderen Werkzeugen erstellt werden (Steinberg et al. 2008, S. 20f.). Ferner ist auch eine dynamische Erzeugung von Modellen zur Laufzeit vorgesehen (Steinberg et al. 2008, S. 36-38).

Durch die Einbindung von generierten Modellen mit Edit- und Editor-Plugins ist eine direkte Integration der erstellten Artefakte mit der Eclipse-Plattform möglich. EMF ist damit als Basis für weitere Technologien gut geeignet. Bei der Entwicklung domänenspezifischer Sprachen können Ecore-Modelle die abstrakte Syntax spezifizieren. Diese beschreibt die Syntax der zu entwickelnden Sprache auf der Metaebene, ohne dabei konkrete Schlüsselwörter zu verwenden (Voelter 2013, S. 177). Aus der Perspektive der Modellierung definiert die abstrakte Syntax somit ein Metamodell.

Die konkrete Syntax textueller Modelle definiert dagegen Schlüsselwörter und Regeln für deren Verwendung, z.B. in Form einer Grammatik. Hierfür ist im Rahmen des Modeling-Projekts das Textual Modeling Framework (TMF)

vorgesehen (Gronback 2009, S. 227-229). Die Werkzeuge des Frameworks bieten Möglichkeiten zur Bearbeitung textueller Modelle mit Syntaxhervorhebung und -vervollständigung, zur Generierung von Parsern für das Einlesen textueller Modelle und zur Instanziierung dieser Modelle gemäß der abstrakten Syntax mit EMF. Textuelle Modelle in einer eigens definierten domänenspezifischen Sprache sind somit als Ecore-Modell verfügbar und mit EMF sowie darauf aufbauenden Technologien nutzbar.

TMF besteht aus den alternativ einsetzbaren Komponenten Xtext und Textual Concrete Syntax (TCS), welche unterschiedliche Ansätze zur Entwicklung der Syntax verfolgen. Xtext bietet die Möglichkeit, eine domänenspezifische Sprache vollständig anhand einer Grammatik in erweiterter Backus-Naur-Form (EBNF) zu spezifizieren, die zu einem gleichzeitig generierten Ecore-Metamodell konform ist. TCS geht dagegen von einem existierenden Ecore-Metamodell aus und generiert dieses nicht. Daher muss eine manuelle syntaktische Zuordnung zwischen der konkreten Syntax des TCS-Modells und der abstrakten Syntax des Ecore-Modells erfolgen.

Die Entscheidung zwischen TCS oder Xtext ist im Wesentlichen davon abhängig, ob abstrakte und konkrete Syntax voneinander getrennt oder zusammen, durch Ableitung aus einer Grammatik, definiert werden sollen. Sofern noch kein Ecore-Metamodell definiert ist, bietet sich die gemeinsame Definition mittels einer Grammatik an.

2.3.2 Modellierung mit Xtext und Ecore

Für die Modellierung der AwS-Spezifikation und der softwaretechnischen Ebene wird Xtext als Teil des Textual Modeling Framework (TMF) zusammen mit Ecore, dem Metamodell der EMF, verwendet. Wie im vorherigen Abschnitt beschrieben, bietet Xtext den Vorteil, die abstrakte und die konkrete Syntax einer domänenspezifischen Sprache (DSL) aus einer Grammatik ableiten zu können. Die für das EMF benötigen Ecore-Modelle werden dann von Xtext generiert. Der Einsatz von Ecore-Modellen begründet sich durch die EMF-Integration in die Eclipse-Plattform.

Mit der Kombination von Xtext und EMF wird eine textuelle Modellierung möglich. Dies erlaubt eine unkomplizierte Eingabe von Modellen, da eine DSL zum Einsatz

kommt, die individuell für die verwendete Modellierungssprache entwickelt werden kann. Die Generierung der Ecore-Modelle stellt sicher, dass zwischen textuellen Modellen und Ecore-Modellen keine Inkonsistenzen auftreten.

Das Vorgehen bei der Modellierung mit Xtext und Ecore wird anhand des Schemas in Abbildung 7 erläutert.

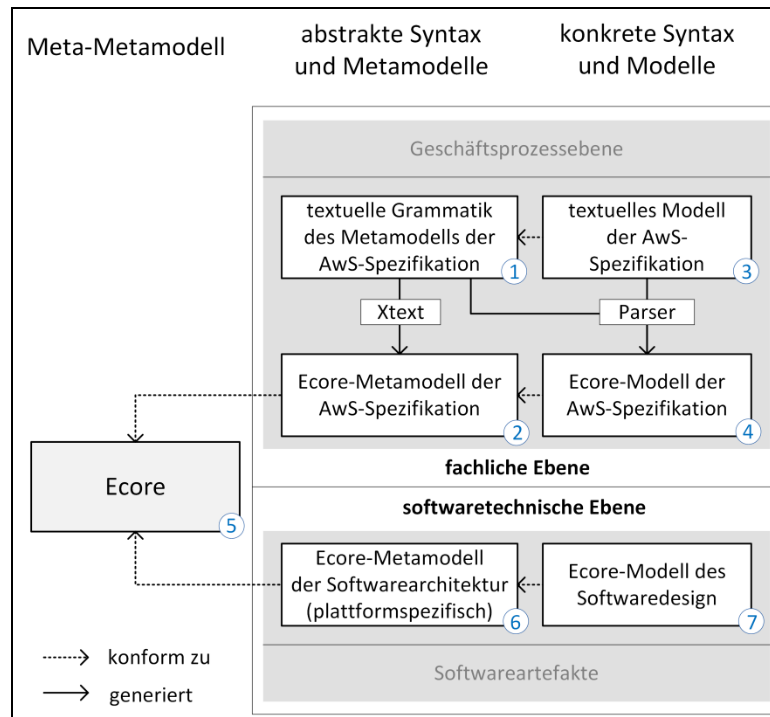

Abbildung 7: Modellierung mit Xtext und Ecore

Gemäß den definierten Ebenen besteht die unterste Schicht der fachlichen Ebene aus der Spezifikation von Anwendungssystemen. Dort beginnt die Unterstützung durch Eclipse und damit die Eingabe des AwS-Modells in der Form eines textuellen Modelles.

Zuvor muss jedoch eine der Ebene entsprechende textuelle Xtext-Grammatik der DSL erstellt werden (1). In diesem Fall ist das Metamodell der Spezifikation von Anwendungssystemen des SOM (Ferstl und Sinz 2013, S. 233) in Form einer Grammatik zu implementieren; dies wird im nächsten Abschnitt umgesetzt. Xtext sieht für die Notation der Grammatik eine eigene Sprache vor, die der EBNF ähnelt (Gronback 2009, S. 228). Eine solche Grammatik beschreibt sowohl die abstrakte als auch konkrete Syntax.

Die abstrakte Syntax entspricht hier dem Metamodell zur Spezifikation von Anwendungssystemen im Ecore-Format (2) (AwS-Ecore-Metamodell). Die

Grammatik wird deshalb als Input für die Generierung des AwS-Ecore-Metamodells benötigt. Zur Generierung wird die Grammatik mit einem Parser eingelesen und in einen abstrakten Syntaxbaum (AST) überführt (Voelter 2013, S. 177-188). Um das AwS-Ecore-Metamodell zu erstellen, wird für jeden Knoten ein Meta-Objekt und für jede Kante eine Meta-Beziehung generiert.

Auch die konkrete Syntax ist durch die Grammatik festgelegt. Sie definiert, welche Schlüsselwörter nach welchen Regeln bei der textuellen Modellierung verwendet werden. Damit kann das textuelle AwS-Modell (3) definiert werden. Eclipse und Xtext stellen auf Grundlage des AwS-Ecore-Metamodells einen Editor mit Funktionen wie Syntaxhervorhebung und -vervollständigung zur Verfügung. Bei der Eingabe, die gemäß der konkreten Syntax erfolgen muss, wird die Konformität zum AwS-Ecore-Metamodell, und damit auch zur Grammatik im Hintergrund, laufend geprüft. Um aus dem textuellen Modell ein Ecore-Modell zu generieren, wird wiederum ein Parser benötigt, der diesmal allerdings spezifisch für die DSL bzw. für die Grammatik ist. Folglich muss auch der Parser vorab generiert werden. Xtext nutzt hierfür den ANTLR-Parsergenerator[2] (Gronback 2009, S. 228), welcher mit der Grammatik als Input einen für die DSL passenden Parser liefert. Beliebige zur Grammatik konforme AwS-Modelle können damit nun in entsprechende Ecore-Modelle (4) überführt werden.

Die auf diese Weise erstellten Ecore-Modelle sind zu dem aus der abstrakten Syntax generierten AwS-Ecore-Metamodell konform. Dieses ist konform zu dem sich selbst definierenden Ecore-Modell (5) des EMF, das somit in diesem Zusammenhang ein Meta-Metamodell darstellt.

Auf softwaretechnischer Ebene wird gemäß des beschriebenen Ebenenmodells ein Modell der Software bzgl. Design und Implementierung definiert. Für die Beschreibung des Softwaredesigns kommen ausschließlich Ecore-Modelle zum Einsatz, da die Modelle nicht textuell zu erstellen sind, sondern automatisch durch Modelltransformation abgeleitet werden.

Das Metamodell dieser Ebene (6) liegt in Form einer Plattformspezifikation vor, die für das Design verwendbare Elemente der Architektur definiert. Entsprechend dieser Spezifikation ist ein Ecore-Metamodell zu erstellen, mit dem das Design der

---

[2] „Another Tool for Language Recognition“, siehe http://www.antlr.org

Software modelliert werden kann (Design-Ecore-Metamodell). Da dieses Metamodell für jede Plattform separat entworfen werden muss und nicht, wie auf fachlicher Ebene, plattformunabhängig ist, wird an dieser Stelle noch nicht definiert, wie es zu erstellen ist. Verschiedene Möglichkeiten zur Modellierung von Ecore-Modellen wurden bereits im vorherigen Abschnitt diskutiert. Allerdings ist anzumerken, dass es auch hier sinnvoll sein kann, statt des Architektur-Ecore-Metamodells eine Xtext-Grammatik zu definieren, da die konzise EBNF-artige Beschreibung möglicherweise gegenüber der Erstellung mit EMF vorzuziehen ist. Das Design-Ecore-Metamodell könnte dann, wie auch auf fachlicher Ebene, einfach abgeleitet werden. Im Unterschied zur fachlichen Ebene ist hier eine durch die Grammatik definierte Sprache aber ein ungenutztes Nebenprodukt. Schließlich ist eine textuelle Modellierung auf dieser Ebene nicht vorgesehen.

Das Ecore-Modell des Softwaredesigns (7) muss, analog zur fachlichen Ebene, zu dem eben diskutierten Design-Ecore-Metamodell konform sein, das selbst wiederum konform zum Ecore-Modell der Meta-Metaebene ist.

Für die Ebene der Softwareartefakte werden keine Ecore-Modelle benötigt. Das Modell besteht hier aus Quellcode, während das Metamodell dieser Ebene in Form einer Plattformspezifikation bereits vorgegeben ist.

2.4 Spezifikation von Modelltransformationen mit Eclipse

2.4.1 Technologien zur Modelltransformation mit Eclipse

Im Rahmen des Ansatzes wird die Modelltransformation, wie auch die Modellierung, softwaretechnisch durch Werkzeuge der Eclipse-Plattform unterstützt. Nach der Betrachtung der Technologien zur Modellierung wird nun diskutiert, welche Technologien sich für die Spezifikation der Transformation eignen.

Die Unterstützung durch die Eclipse-Plattform beginnt mit dem Übergang zur softwaretechnischen Ebene (vgl. Abbildung 6). Im Rahmen der besprochenen Transformationsschritte ergeben sich zwei verschiedene Arten der Transformation. Von der Ebene des AwS-Modells zur Ebene des Softwaredesigns erfolgt eine Modell-zu-Modell-Transformation. Der Übergang von Softwaredesign zur Implementierung ist dagegen eine Modell-zu-Text-Transformation. Input und Output der Modell-zu-Modell-Transformation sind Ecore-Modelle des EMF, bei der Modell-zu-Text-Transformation ist der Input ebenfalls ein Ecore-Modell, während es sich bei dem Output um Quellcode handelt. Die Transformationen folgen dabei dem Muster der Metamodell-Transformation der MDA (OMG 2003a, S. 3-9), so dass die Spezifikation der jeweiligen Transformation auf Ebene der Metamodelle erfolgt. Dies muss auch durch die verwendeten Technologien unterstützt werden, damit eine Transformation ausgehend vom entwickelten AwS-Ecore-Metamodell spezifiziert werden kann. Eine weitere Anforderung ist die Kompatibilität zum EMF, welches die Transformation durchführen soll.

Technologien zur Modell-zu-Modell-Transformation

Das Modeling-Projekt der Eclipse-Plattform bietet für Modell-zu-Modell-Transformationen die *Atlas Transformation Language* (ATL) und *Query View Transformation* (QVT) an (Steinberg et al. 2008, S. 231).

ATL wurde an der Universität Nantes entwickelt und soll eine zu QVT ähnliche Transformationssprache für MDE bereitstellen (Jouault et al. 2006). Die Sprache setzt auf dem EMF auf und verwendet dessen Schnittstellen zur Modellierung. Als Format für Quell- und Zielmodell wird, wie auch bei EMF, XMI eingesetzt.

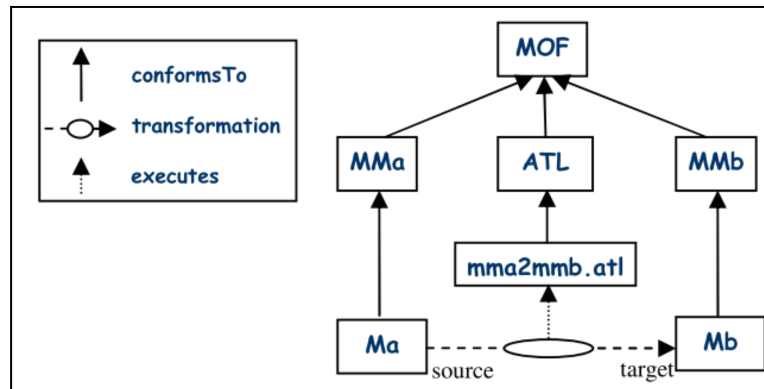

Abbildung 8: ATL-Transformationsmodell (Jouault und Kurtev 2005, S. 2)

Abbildung 8 zeigt das Muster, nach dem eine ATL-Transformation durchgeführt wird. Der beschriebene Ansatz orientiert sich an den Ebenen M1, M2 und M3 der MOF (OMG 2005a, S. 23). Die unterste Modellebene M1 enthält ein Quellmodell Ma, ein Zielmodell Mb und eine Transformationsspezifikation mma2mmb.atl. Die Metaebene enthält die entsprechenden Metamodelle, wobei Ma zu dem Metamodell MMa, Mb zu dem Metamodell MMb und mma2mmb.atl zum Metamodell der ATL konform ist. Alle drei Metamodelle sind konform zum Meta-Metamodell der MOF. Neben MOF orientiert sich der beschriebene Ansatz auch an der MDA, die das Muster der metamodellbasierten Transformation vorschlägt (OMG 2003a, S. 3-9).

QVT wird von der OMG als Standard für die Modelltransformationen im Rahmen der MOF 2.0 entwickelt (OMG 2011). Die Abkürzung deutet die unterstützten Merkmale Query, View und Transformation an. Das Abfragen einzelner Modellelemente (Query) ist mittels der *Object Constraint Language* (OCL) der OMG (2010) möglich. Selbstverständlich lassen sich die Konzepte des MDA-Ansatzes der OMG auch mit QVT umsetzen – die Technologie wird dafür explizit als Beispiel genannt (OMG 2003a, S. 3-10). QVT unterstützt damit auch die metamodellbasierte Transformation. Da diese bereits diskutiert wurde, wird hier auf eine Abbildung verzichtet[3].

QVT besteht aus drei Komponenten. *QVT-Relations* nutzt Relationen zur Beschreibung der Transformationen und arbeitet deklarativ (OMG 2011, S. 13-62). Die *Operational Mapping Language* (OML) ist eine imperative Sprache zur Beschreibung von Zuordnungsbeziehungen zwischen Modellen (Mappings) (OMG 2011, S. 63-144). *QVT-Core* enthält Elemente, die keinen hohen Abstraktionsgrad

[3] Siehe Kapitel 2.1.4 und 1.1

aufweisen; es wird von den anderen beiden Sprachen als Basis genutzt (OMG 2011, S. 145-163).

Im Gegensatz zu ATL handelt es sich bei QVT nicht um eine Implementierung einer Transformationssprache, sondern um eine Spezifikation im Rahmen eines Standards. Die OMG verfolgt mit der Standardisierung das übergeordnete Ziel der Kompatibilität zwischen Werkzeugen (OMG 2001, S. 1) und stellt selbst keine Implementierung in Form von Software zur Verfügung.

Implementierungen von *QVT-Relations* und der *Operational Mapping Language* sind für die Eclipse-Plattform verfügbar (Gronback 2009, S. 231-243), wobei fraglich ist, wie umfassend der vorgegebene Standard umgesetzt wurde. Die Eclipse-Implementierung greift auf die Schnittstellen des EMF zurück und unterstützt damit auch Ecore-Modelle.

Insgesamt erfüllen sowohl ATL als auch QVT die gestellten Anforderungen an eine Transformationssprache. QVT bietet allerdings den Vorteil der Standardisierung, die zukünftig möglicherweise zur besseren Kompatibilität unterschiedlicher Werkzeuge beitragen kann.

Technologien zur Modell-zu-Text-Transformation

Für die Modell-zu-Text-Transformation sieht das Modeling-Projekt der Eclipse-Plattform die Nutzung von Java Emitter Templates (JET) oder Xtend vor[4] (Gronback 2009, S. 277f.).

Bei JET handelt es sich um Vorlagen, die z.B. die Definition für Java-Klassen enthalten, wobei variable Inhalte, wie etwa der Name einer Klasse, bei Instanziierung des Templates je nach Modell eingefügt werden.

Das EMF bringt einen Generator mit (Steinberg et al. 2008, S. 341-346), der mit Hilfe der Java Emitter Templates ohne Weiteres aus Ecore-Modellen Java-Code generieren kann. So werden Ecore-Elemente, wie etwa *EClasses*, in Klassen umgewandelt, die Assoziations- und Kompositionsbeziehungen über *EReferences* herstellen. Die Nutzung von JET ist damit allerdings nur für Modelle geeignet, die Ecore als Metamodell verwenden und somit eine Klassenstruktur direkt modellieren.

[4] Ehemals wurde zusätzlich noch die mittlerweile eingestellte Sprache Xpand unterstützt, siehe http://www.openarchitectureware.org/

Ecore soll hier allerdings als Meta-Metamodell genutzt werden, so dass die in Form von JET bereitgestellten Vorlagen nicht genutzt werden können.

Das Erstellen eigener JET-kompatibler Vorlagen ist zwar ebenfalls möglich, allerdings bringt die an Java Server Pages angelehnte Template-Sprache laut Stahl et al. einige Probleme mit sich. Beispielsweise können nicht mehrere Dateien aus einem Template geöffnet werden (Stahl et al. 2007, S. 147).

Alternativ zu JET bietet sich die Sprache Xtend (Voelter 2013, S. 274-278) an, die unter anderem die Definition von Templates beherrscht. Es handelt sich um eine Java-artige Programmiersprache, deren Umfang allerdings weit über die Beschreibung von Templates hinausgeht. Der Xtend-Compiler übersetzt Xtend-Quellcode zunächst in Java-Quellcode, der dann mit einem Java-Compiler erneut übersetzt und in einer JVM ausgeführt werden kann. Die Technologie basiert auf Xtext, das bereits zur Beschreibung der Grammatik herangezogen wurde.

Die Codegenerierung kann in Xtend mit Hilfe von *Template Expressions*, wie auch mit JET, auf der Basis von Vorlagen erfolgen. Code-Generatoren sind im Gegensatz zu JET frei programmierbar, was mehr Implementierungsaufwand bedeutet, aber auch höhere Flexibilität verspricht.

Xtend verwendet die Schnittstellen des EMF und bietet damit auch die benötigte Integration in die Eclipse-Plattform an. Die Verwendung von Ecore-Modellen als Input, wie sie hier für die Modell-zu-Text-Transformation benötigt wird, ist ohne Weiteres möglich.

2.4.2 Spezifikation der Transformation mit QVTo und Xtend

Auf Grundlage der vorgegangenen Diskussion werden nun Technologien zur Spezifikation der Transformationsschritte ausgewählt.

Die Modell-zu-Modell-Transformation soll mit QVT erfolgen. Vorteilhaft ist hier die Standardisierung durch die OMG, die Kompatibilität verspricht, sowie die Tatsache, dass auch die MDA von der OMG standardisiert wurde. Der Einsatz von QVT im Rahmen der MDA sollte somit ohne Weiteres möglich sein. Als Sprache für die Metamodell-Transformation wird OML gewählt, die für die Spezifikation von Mappings ausgelegt ist. Die Eclipse-Plattform bietet mit QVTo eine Implementierung dieses Standards an (Gronback 2009, S. 238).

Für Modell-zu-Text-Transformationen kommt die aus dem Xtext-Projekt stammende Programmiersprache Xtend zum Einsatz. Sie besitzt einen großen Umfang und bietet bei der Implementierung von Code-Generatoren im Vergleich zu JET deutlich mehr Flexibilität. Eine Eclipse-Integration ist verfügbar (Voelter 2013, S. 274-278).

Da die Transformation plattformabhängig ist, wird an dieser Stelle nur ein Beispiel gegeben. Es soll verdeutlichen, wie sich mit QVTo Zuordnungsbeziehungen zwischen Metamodellen definieren lassen.

Abbildung 9 zeigt die beispielhafte QVTo-Transformation konzeptueller Objekttypen zu Ecore-Klassen. Für Quelle und Ziel müssen Ecore-Metamodelle im EMF registriert sein, die in der ersten Zeile definiert werden (source bzw. target). In diesem Fall ist das Input-Metamodell das durch Xtext generierte Ecore-Metamodell der AwS-Spezifikation, das Output-Metamodell ist Ecore.

```
// Definition von Quell- und Ziel-Metamodell
transformation Kos_Classes(in source:KosVos, out target:ECore);

// Einsprungspunkt: Aufruf der Transformation KOS -> EPackage
main() {
  source.objectsOfType(KOS)[KOS] -> ePackage();  }
// Zuordnungsfunktion: KOS -> EPackage
mapping KOS::ePackage() : EPackage {
  result.name := self.name;
  result.eClassifiers += self.allSubobjectsOfKind(KOT)[KOT] -> map eClass();  }
// Zuordnungsfunktion: KOT -> EClass
mapping KOT::eClass() : EClass {
  result.name := self.name;
  result.eStructuralFeatures += self.attribute -> map eAttribute();         }
```

Abbildung 9: Beispielhafte QVTo-Transformation

In der main-Methode, die QVTo als Einsprungspunkt dient, werden auf dem Quellmodell alle Objekte des Typs *KOS* abgefragt (Query). Für jedes Objekt der zurückgegebenen Menge wird dann die Zuordnungsfunktion *ePackage* aufgerufen. Diese Funktion definiert ein Mapping zwischen dem Meta-Objekt *KOS* und dem Meta-Objekt *EPackage*, so dass für jedes konzeptuelle Objektschema ein Paket erstellt wird. Innerhalb der Definition der Zuordnungsfunktion werden die zugehörigen Attribute zugeordnet, hier der Name und die im KOS enthaltenen KOT. Für letztere ist ein weiteres Mapping *eClass* definiert, welches für jeden KOT eine Klasse erstellt.

Die Modell-zu-Text-Transformation mit Xtend folgt als zweiter Transformationsschritt, der ebenfalls plattformabhängig ist. Abbildung 10 zeigt als einfaches Beispiel die Implementierung eines einfachen Java-Quellcode-Generators

in Xtend, der ausgehend von einem Ecore-Modell mit Objekten des Typs *EClass* Java-Klassen mit öffentlichen Methoden erstellt. Der Generator verwendet dazu Template-Ausdrücke (Voelter 2013, S. 274-278).

```
def generate(EClass c) {
''' public class «c.name» {
      «FOR method : c.methods»
      public «method.return» «method.name» («method.parameters») {
        «method.body»
      }
      «ENDFOR»
''' }
```

Abbildung 10: Beispiel für einen einfachen Xtend-Template-Ausdruck

Die *generate*-Methode erhält eine Instanz des Meta-Objekts Class. Die drei Apostrophe schließen in Xtend eine eingebettete Vorlage mit Java-Quellcode ein. Innerhalb der Vorlage werden alle Zeichen in den auszugebenden Text (Quellcode) übernommen, sofern sie nicht von Guillemets [5] umgeben sind. Guillemets umschließen Befehle, die in der Ausgabe durch deren Rückgabewerte ersetzt werden. So wird etwa der Klassenname durch den Aufruf von c.name zurückgegeben und in die Ausgabe eingefügt. Kontrollstrukturen wie *FOR* sind ebenfalls einsetzbar; im Beispiel wird für jede Methode in der Class-Instanz Code erzeugt.

[5] „französische Anführungszeichen": « »

2.5 Eine textuelle Notation zur Modellierung der Anwendungssystemspezifikation der SOM-Methodik

Eine vorliegende Spezifikation von Anwendungssystemen (Ferstl und Sinz 2013, S. 220-236), abgeleitet aus einem ausreichend detaillierten Geschäftsprozess mit Interaktionsschema (IAS) und Vorgangs-Ereignis-Schema (VES), soll als textuelles Modell mit Hilfe der zu entwickelnden Beschreibungssprache modelliert werden. Hierfür wird eine Xtext-Grammatik definiert, die das Metamodell der AwS-Spezifikation zusammen mit einer konkreten Syntax beschreibt.

Die Xtext-Grammatik spezifiziert relevante Objekttypen und Beziehungen in Form einer EBNF-artigen Syntax, geht aber über die EBNF hinaus und ist auch für die Beschreibung von Modellierungssprachen geeignet. So kann auf Grundlage der Grammatik ein Ecore-Metamodell abgeleitet werden (Brambilla et al. 2012, S. 103). Die Grammatik selbst basiert auf Regeln (Eclipse 2011, S. 54-69), die nun, sofern sie hier relevant sind, kurz erläutert werden:

- Eine *Parser Rule* führt zur Produktion eines Meta-Objekts im Metamodell. Die Regel kann *Assignments*, *Cross References* und *Terminal Rules* enthalten. Außerdem ist ein Name anzugeben, den das Meta-Objekt übernimmt.
- *Terminal Rules* definieren Zeichen oder Zeichenketten, die z.B. ein Alphabet erlaubter Werte oder Schlüsselwörter bestimmen (konkrete Syntax).
- *Assignments* sind Zuweisungen von Parser Rules oder Terminal Rules. Die Zuweisung einer *Parser Rule* entspricht einer Attribut-Zuordnungsbeziehung (*has*-Beziehung) im Metamodell.
- *Cross References* sind Referenzen und führen zu Assoziationen (*connects*-Beziehung).
- *Unassigned Rule Calls* generalisieren mehrere Parser Rules mit *is_a*-Beziehungen. Sie definieren keine *Assignments* oder *Cross References*.

Die Grammatik wird nun in zwei Teilen anhand dieser Regeln erläutert. Als Ausgangspunkt dient das Metamodell für die Spezifikation von Anwendungssystemen (AwS-Spezifikation) der SOM-Methodik (Ferstl und Sinz 2013, S. 233). Betrachtet werden die für das konzeptuelle Objektschema (KOS) und das Vorgangsobjektschema (VOS) relevanten Objekttypen mit allen möglichen Arten von Beziehungen.

2.5.1 Die Xtext-Grammatik der Anwendungssystemspezifikation

Abbildung 11 illustriert den ersten Teil der Grammatik anhand der Definition von Objekttypen mit Attributen und Operatoren. In den ersten beiden Zeilen wird der Grammatik zunächst ein Namespace zugewiesen, außerdem werden mittels *Grammer Mixin* (Eclipse 2011, S. 70f.) häufig verwendete Terminal Rules, z.B. Whitespaces und Kommentarzeichen, importiert. Der Namespace des zu erstellenden Ecore-Metamodells wird in der zweiten Zeile zusammen mit einem Namen als URI (Uniform Resource Indicator) angegeben.

```
grammar de.uniba.wiai.seda.fha.som.aws.Aws with org.eclipse.xtext.common.Terminals
generate aws "http://www.uniba.de/wiai/seda/fha/som/aws/Aws"

// Parser Rule: AwS has Objekttypen, Beziehungen
Aws: (objekttypen+=Objekttyp | beziehungen+=Beziehung)*;

Objekttyp: KOT | VOT; // Unassigned Rule Call: KOT/VOT is_a Objekttyp
KOT: OOT | TOT | LOT; // Unassigned Rule Call: OOT/TOT/LOT is_a KOT

// Parser Rules: OOT/TOT/LOT/VOT has Attribute, Operatoren
OOT: "OOT" name=ID "{" (attribute+=Attribut | operatoren+=Operator)* "}";
TOT: "TOT" name=ID "{" (attribute+=Attribut | operatoren+=Operator)* "}";
LOT: "LOT" name=ID "{" (attribute+=Attribut | operatoren+=Operator)* "}";
VOT: "VOT" name=ID "{" (attribute+=Attribut | operatoren+=Operator)* "}";

// Parser Rule: Attribut has Name
Attribut: name=ID ";";
// Parser Rule: Operator has Name, Output-/Input-Parameter
Operator: (outParam=ID)? name=ID "("inParam+=ID*")"";";

// Terminal Rule: ID
terminal ID: '^'?('a'..'z'|'A'..'Z'|'_'|'<'|'>')+;
```

Abbildung 11: Xtext-Grammatik für die Spezifikation von Anwendungssystemen (Teil 1)

Die erste Parser Rule *Aws* enthält zwei *Assignments*, die im Metamodell je eine *has*-Beziehung zwischen *Aws* und *Objekttyp* bzw. *Beziehung* definieren – ein Anwendungssystem besteht somit aus Objekttypen und Beziehungen. Die konkrete Syntax legt in diesem Fall nur fest, nach welchen Regeln die *Assignments* angeordnet sind, ohne Schlüsselwörter zu definieren. Die Zuweisungen durch den Operator | stellen Alternativen dar, die Klammerung und der Operator * beschreiben eine sich beliebig oft wiederholende Sequenz der Alternativen. Beide *Assignments* ermöglichen über den Zuweisungsoperator += die Zuweisung mehrerer Werte des jeweiligen Typs (Objekttyp bzw. Beziehung). Bezüglich des Metamodells legen die beschriebenen Operatoren die Kardinalität der beiden *has*-Beziehungen auf (0,*) fest.

Es fällt auf, dass im Unterschied zum Metamodell gemäß der SOM-Methodik (Ferstl und Sinz 2013, S. 233) hier das zusätzliche Element *AwS* vorhanden ist. Dieses Element wird zur Definition der ersten Regel benötigt, die als Einstiegsregel fungiert. Der Einstieg mit mehr als einer Regel (z.B. Objekttyp und Beziehung) ist nicht möglich. Der Grund dafür liegt in der Verarbeitung der Grammatik durch den Parser, der einen abstrakten Syntaxbaum (AST) mit genau einer Wurzel aufbaut (Voelter 2013, S. 177f.).

Die Regel für den Objekttyp ist ein *Unassigned Rule Call*, der KOT und VOT in einem Objekttyp generalisiert. Ebenso verhält es sich mit der KOT-Regel, die die objekt-, transaktions- und leistungspezifischen Objekttypen (OOT, TOT und LOT) generalisiert.

Da bei einem *Unassigned Rule Call* keine *Assignments* möglich sind, werden die Attributzuweisungsbeziehungen von Objekttyp zu Operator und Attribut in allen spezialisierten Objekttypen (OOT, TOT, LOT und VOT) definiert. *Assignments*, die von allen Subtypen eines Supertyps definiert werden, ordnet Xtext nur dem Supertyp zu (Eclipse 2011, S. 68f.) – dadurch ergeben sich im Ecore-Metamodell die in der Grammatik fehlenden Beziehungen von Objekttyp zu Operator und Attribut. OOT, TOT, LOT und VOT besitzen neben den entsprechenden Assignments jeweils noch einen Namen, dessen erlaubte Zeichen mit der Terminal Rule *ID* bestimmt sind. Darauf folgen beliebig viele Attribute und Operatoren in geschweiften Klammern.

Attribute besitzen einen Typ und einen Namen, während Operatoren aus einem Namen und Klammern mit beliebig vielen Parametern bestehen (Kardinalität 0,*). Die Terminal Rule *ID* legt auch bei Attribut und Operator die zulässigen Zeichen fest. Diese sind durch einen Ausdruck definiert, der alphanumerische Zeichen, Unterstriche und spitze Klammern erlaubt.

Abbildung 12 zeigt den zweiten Teil der entwickelten Xtext-Grammatik. Dort werden Beziehungen zwischen Objekttypen beschrieben.

```
// Interacts_with/Is_a/Is_part_of is_a Beziehung
Beziehung: Interacts_with | Is_a | Is_part_of;

// Interacts_with has Kardinalitaet; connects Objekttyp1, Objekttyp2
Interacts_with: "interacts_with:"? objekttyp1=[Objekttyp]
       kardinalitaet=(Kardinalitaet01 | Kardinalitaet0x | Kardinalitaet1x)
       objekttyp2=[Objekttyp] ";";
// Is_a has Kardinalitaet; connects Objekttyp1, Objekttyp2
Is_a: "is_a:" objekttyp1=[Objekttyp]
       kardinalitaet=(Kardinalitaet01 | Kardinalitaet11) objekttyp2=[Objekttyp] ";";
// Is_part_of has Kardinalitaet; connects Objekttyp1, Objekttyp2
Is_part_of: "is_part_of:" objekttyp1=[Objekttyp]
       kardinalitaet=(Kardinalitaet01 | Kardinalitaet0x | Kardinalitaet1x)
       objekttyp2=[Objekttyp] ";";

// Kardinalitaet-(0,1)/(0,*)/(1,1)/(1,*) is_a Kardinalitaet
Kardinalitaet: Kardinalitaet01 | Kardinalitaet0x | Kardinalitaet11 |
                Kardinalitaet1x;
Kardinalitaet01: ("--" | "(0,1)") {Kardinalitaet01};
Kardinalitaet0x: ("->" | "(0,*)") {Kardinalitaet0x};
Kardinalitaet11: ("==" | "(1,1)") {Kardinalitaet11};
Kardinalitaet1x: ("=>" | "(1,*)") {Kardinalitaet1x};
```

Abbildung 12: Xtext-Grammatik für die Spezifikation von Anwendungssystemen (Teil 2)

Eine Beziehung wird mittels *Unassigned Rule Call* zu *Interacts_with*, *Is_a* und *Is_part_of* spezialisiert. Alle drei Spezialisierungen besitzen *Cross References* für zwei Objekttypen und eine Kardinalität. Wie auch bei den spezialisierten Objekttypen, erstellt Xtext für die drei Subtypen keine Attributzuweisungsbeziehungen, sondern ordnet sie im Metamodell dem Supertyp *Beziehung* zu.

Die Regel *Interacts_with* beginnt mit einem optionalen Schlüsselwort, während *Is_a* und *Is_part_of* immer ein Schlüsselwort erwarten. Eine Beziehung, die ohne Schlüsselwort definiert wird, ist somit immer vom Subtyp *Interacts_with*. Dies ermöglicht eine konzise Modellierung von Beziehungen im VOS, da dort nur dieser Subtyp vorkommt (Ferstl und Sinz 2013, S. 230).

Die Kardinalität ist in Grammatik und Metamodell aus zwei Gründen explizit modelliert. Zum Einen sind damit Restriktionen für jeden Subtyp von Beziehung möglich. *Interacts_with* und *Is_part_of* erlauben die Kardinalitäten (0,1), (0,*) und (1,*) – für *Is_a* ist jedoch nur (0,1) und (1,1) zulässig (Ferstl und Sinz 2013, S. 224f.). Zum Anderen ergibt sich so die Möglichkeit, Abkürzungsschreibweisen für Kardinalitäten in die Syntax der Notation aufzunehmen, etwa ein Pfeil (`->`) für die Kardinalität (0,*).

2.5.2 Das Ecore-Metamodell der Anwendungssystemspezifikation

Das resultierende AwS-Ecore-Metamodell ist in Abbildung 13 dargestellt, die grafische Darstellung wurde mit dem EMF geniert.

Die verwendeten Modellelemente ergeben sich aus dem bereits besprochenen Ecore-Meta-Metamodell[6], es handelt sich um die Elemente

- *EClass* für Meta-Objekte,
- *EReference* ohne *Containment* (kleiner Pfeil) für Assoziationen,
- *EReference* mit *Containment* (Pfeil mit ausgefüllter Raute) für Attributzuweisungsbeziehungen und
- *EClass* mit *eSuperType* für Generalisierungsbeziehungen (großer Pfeil).

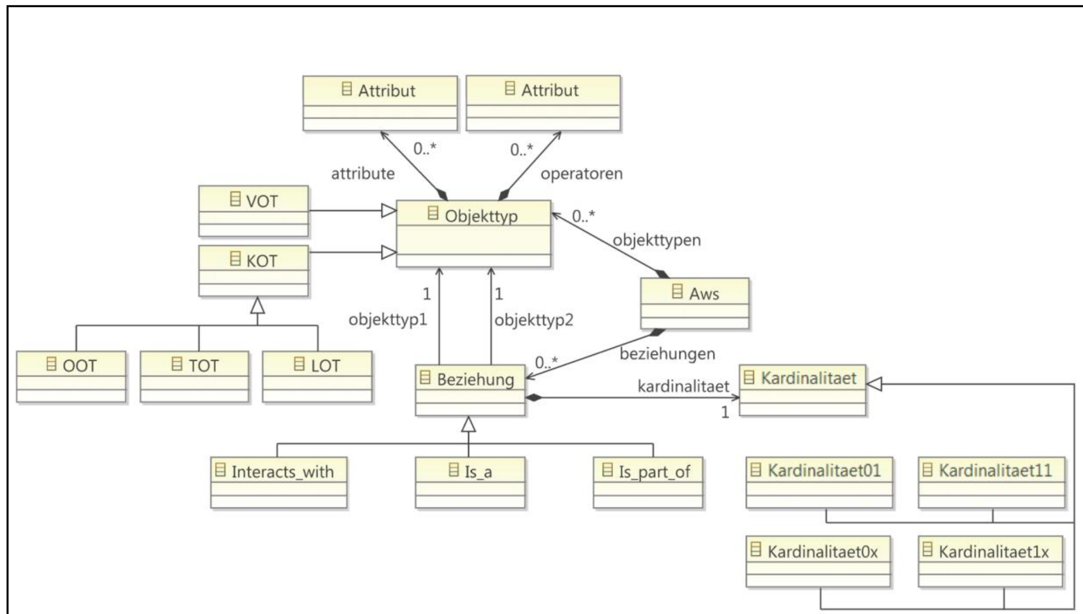

Abbildung 13: Resultierendes Ecore-Metamodell

2.5.3 Erweiterung zur Modellierung des konzeptuellen Objektschemas und des Vorgangsobjektschemas

Um mehrere KOS und VOS explizit abbilden zu können, erfolgt noch eine Erweiterung der Grammatik und damit auch des AwS-Ecore-Metamodells. Sie ermöglicht neben der Bildung eines KOS und mehreren VOS ferner die Vergabe von Versionsnummern. Letztere sind bei Änderungen auf Geschäftsprozessebene nach erneuter Ableitung der AwS-Spezifikation zur Unterscheidbarkeit von alten Modellen sinnvoll.

[6] Siehe Kapitel 2.3.1

Abbildung 14 zeigt die von der bereits definierten Grammatik erbende erweiterte Grammatik. Wie den Kommentaren zu entnehmen ist, enthält ein KOS beliebig viele KOT und Beziehungen jeder Art, ein VOS beliebig viele VOT und *Interacts_with*-Beziehungen.

Das resultierende erweiterte AwS-Ecore-Metamodell ist in Abbildung 15 dargestellt. Die aus dem AwS-Metamodell vererbten Elemente sind grau eingefärbt.

```
grammar de.uniba.wiai.seda.fha.som.aws.kosvos.Kosvos with
        de.uniba.wiai.seda.fha.som.aws.Aws
import "http://www.uniba.de/wiai/seda/fha/som/aws/Aws" as aws
generate kosvos "http://www.uniba.de/wiai/seda/fha/som/aws/kosvos/Kosvos"

// KosVos has (1,1) KOS, (1,*) VOS
KosVos: "AWS " (name=ID "v"version=VER_NR)? kos=KOS vos+=VOS+;

// KOS has Objekttypen (KOT), Beziehungen
KOS: (name="KOS" | "KOS:" name=ID) "{"
     (objekttypen+=KOT | beziehungen+=Beziehung)* "}";
// VOS has Objekttypen (VOT), Beziehungen (Interacts_with)
VOS: (name="VOS" | "VOS:" name=ID) "{"
     (objekttypen+=VOT | beziehungen+=Interacts_with)* "}";

terminal VER_NR: ('0'..'9')+'.'('0'..'9')+;
```

Abbildung 14: Erweiterte Grammatik für KOS und VOS

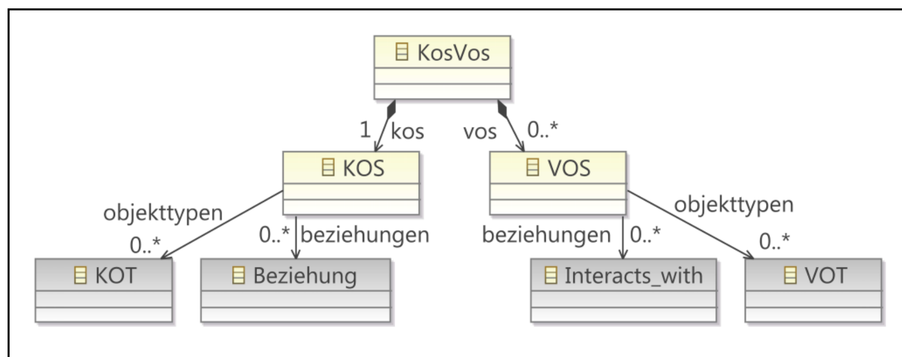

Abbildung 15: AwS-Ecore-Metamodell mit KOS und VOS

Die Syntax der erweiterten Grammatik wird in Abbildung 17 durch ein abstraktes Beispiel verdeutlicht, welches das Modell aus Abbildung 16 textuell modelliert. Es enthält die Objekte *Server* und *Client*, zwischen denen eine Vereinbarungs- und eine Durchführungstransaktion verlaufen. Die Beschreibung mittels textueller Notation enthält im Gegensatz zur Modellierung mittels Diagramm zusätzlich noch Attribute und Operatoren einiger Objekttypen. Attribute wurden beispielhaft für den OOT *Client*, den TOT *Vereinbarung* und den VOT *>Vereinbarung* definiert. Ein exemplarischer Operator ist für den VOT *>Vereinbarung* eingetragen. Attribute und Operatoren dienen als Grundlage für die Implementierung. Sie stehen jeweils in

geschweiften Klammern und sind per Semikolon getrennt. Operatoren weisen Klammern für etwaige Parameter auf; sie enthalten an dieser Stelle noch kein Lösungsverfahren.

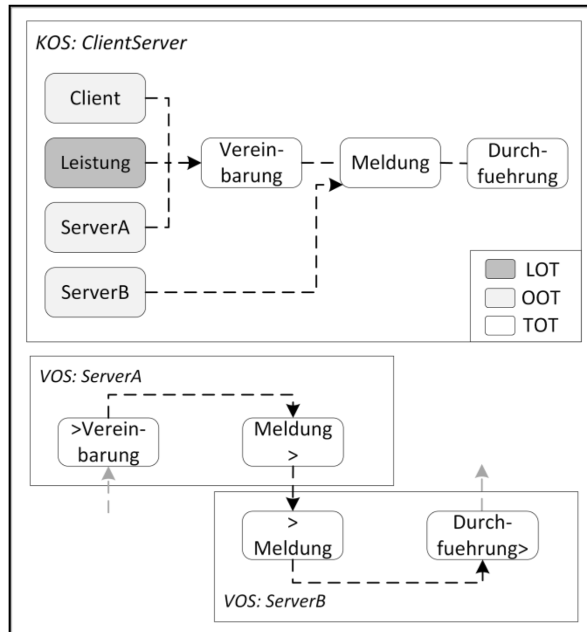

Abbildung 16: Beispiel zur textuellen Modellierung

```
AWS Beispiel v 1.0
KOS: ClientServer {
  OOT Client { Name; }
  LOT Leistung { … }
  OOT ServerA { … }
  OOT ServerB { … }
  TOT Vereinbarung { Menge; }
  TOT Meldung { … }
  TOT Durchfuehrung { … }
  ServerA -> Vereinbarung;
  Leistung -> Vereinbarung;
  Client -> Vereinbarung;
  Vereinbarung -- Meldung;
  Meldung -- Durchfuehrung;
  ServerB -> Meldung;
}
```

```
VOS: ServerA {
  VOT >Vereinbarung {
    Client; Server; Leistung;
    MengeVereinbaren();
  }
  VOT Meldung> { … }
  Vereinbarung> -> Meldung>;
  Meldung> -> >Meldung;
}
VOS: ServerB {
  VOT >Meldung { … }
  VOT Durchfuehrung> { … }
  >Meldung -> Durchfuehrung>;
}
```

Abbildung 17: Anwendungsbeispiel der textuellen Modellierungssprache

Da KOS und VOS Teil der Grammatik sind, können diese auch explizit modelliert werden. Im Beispiel wird ein KOS mit dem Namen *ClientServer* mit zwei VOS *ServerA* und *ServerB* modelliert. *ServerA* ist an der Vereinbarung mit dem Client beteiligt und erstattet danach Meldung an *ServerB*. Letzter stößt die Durchführungstransaktion an, welche zum Client verläuft.

3 Ein konkreter Ansatz zur modellgetriebenen Entwicklung serviceorientierter Systeme

3.1 Gesamtüberblick über alle Modellebenen

Der bisher nur auf Ebene der Anwendungssysteme definierte Ansatz soll nun anhand eines konkreten Architekturstils und einer bestimmten Plattform instanziiert werden. Dazu wird eine beispielhafte Architektur für serviceorientierte Systeme entwickelt und auf der Plattform Java Enterprise Edition (Java EE) umgesetzt.

Die Transformationen lassen sich zusammen mit den plattformspezifischen Ebenen nun vollständig beschreiben. Bevor die Ebenen im Einzelnen entwickelt werden, wird das Ergebnis überblicksartig vorweggenommen (Abbildung 18).

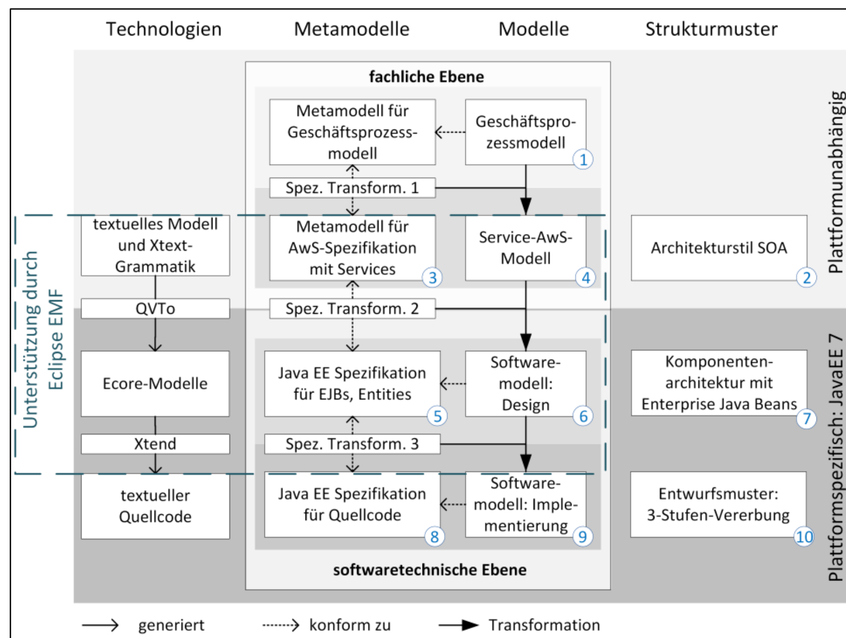

Abbildung 18: Abbildung aller Transformationsebenen

Nachdem auf Geschäftsprozessebene die Diskurswelt abgegrenzt und aus Struktur- und Verhaltenssicht (IAS und VES) modelliert ist (1), folgt eine modellbasiere Ableitung der initialen Spezifikation der Anwendungssysteme durch KOS und VOS. Um die Merkmale des Architekturstils SOA (2) zu berücksichtigen, wird das in Ecore implementierte AwS-Metamodell um ein Metamodell für Services erweitert (3). Letzteres ermöglicht die Abgrenzung von serviceorientierten Anwendungssystemen und die Spezifikation von Services auf Modellebene (4). Services kapseln hier konzeptuelle Objekte des KOS und VOS, definieren Schnittstellen und stehen untereinander in Choreografie- und Orchestrierungs-

beziehungen (siehe nächster Abschnitt). Zwischen dem Metamodell des serviceorientierten Anwendungssystems und dem Softwaremodell des Designs wird ein Beziehungs-Metamodell definiert, das eine Transformation ermöglicht. Das Ziel-Metamodell dieser Transformation bildet die Plattformspezifikation für Elemente der Softwarearchitektur ab, in diesem Fall für Java EE (5). Das Softwaremodell des Designs (6) definiert sich auf dieser Ebene durch Module, die Java-EE-Komponenten enthalten. Services der vorherigen Ebene finden sich als Session Beans wieder. Sie stehen mit synchronen oder asynchronen Aufrufbeziehungen in Verbindung und tauschen Nachrichten mit definierten Datentypen aus. Insgesamt folgt das Softwaremodell des Designs damit einer komponentenbasierten Architektur (7). Im letzten Transformationsschritt wird Quellcode für alle Komponenten generiert. Dabei kommen Vorlagen zum Einsatz, welche die Architektur entsprechend der Spezifikationen von Java EE 7 (8) programmtechnisch umsetzen (9). Der generierte Quellcode wird abschließend mittels dreistufiger Vererbung erweitert (10). Dadurch werden die Lösungsverfahren einzelner Komponenten implementiert.

3.2 Grundlagen der serviceorientierten Architektur

Ausgangspunkt für die Entwicklung des serviceorientierten AwS ist eine Spezifikation in Form von KOS und VOS. Ziel ist es, Merkmale des Architekturstils SOA auf fachlicher Ebene zu berücksichtigen. Die Interpretation von SOA als Architekturstil auf dieser Ebene geht über das ursprünglich von IBM vorgestellte Konzept hinaus (Ferstl und Sinz 2013, S. 438f.). Die von der W3C standardisierten Web-Services (2004) sind als technologische Lösung von hoher Bedeutung, sie werden auf dieser (fachlichen) Ebene nicht berücksichtigt. Stattdessen wird SOA als verallgemeinertes Konzept interpretiert, welches anhand zu bestimmender Merkmale definiert ist.

Nach Siedersleben (2007, S. 110f.) gestaltet eine SOA kein einzelnes System, sondern Systemlandschaften. Diese können auch aus Altsystemen bestehen, Daten- oder Funktionsredundanz aufweisen und sind bezüglich ihrer Komponenten und Subsysteme oft heterogen. Daraus ergibt sich die Anforderung, die Systemlandschaft in Form von Teil-AwS modellieren zu können. Heterogene Teilsysteme müssen einheitlich ansprechbar sein, damit Funktionen und Daten von allen Systemen zugreifbar sind, die sie benötigen. Dies unterstützt SOA durch drei Grundideen (2007, S. 111f.):

1. Systeme sind komponentenorientiert und verfügen über definierte Schnittstellen, mit denen sie untereinander kommunizieren. Dadurch sind Systeme und Komponenten unabhängig von ihrer Implementierung als Service ansprechbar, der Funktionen des dahinterstehenden Systems als Serviceoperationen anbietet. Problematisch sind Seiteneffekte und die Beschreibung von Services.
2. Systeme sind lose gekoppelt und kommunizieren über Nachrichten asynchron. Ein System, welches eine Nachricht empfängt, muss nicht verfügbar sein, wenn die Nachricht gesendet wird. Synchrone Kommunikation ist innerhalb von Komponenten möglich.
3. Die Ablaufsteuerung wird durch eine Workflow-Komponente realisiert, die beispielsweise eine Workflow-Definition als BPEL-Prozess (OASIS 2007) ausführt.

Die standardisierten Schnittstellen schaffen eine Unabhängigkeit von der Implementierung; eine Abhängigkeit von der Technologie der Service-Schnittstelle ist aber unvermeidbar. SOA verlagert gewissermaßen die Abhängigkeit von der Implementierung potenziell alter Systeme zu einer Abhängigkeit von der eingesetzten Servicetechnologie; letztere ist aufgrund von standardisierten Schnittstellen und höherer Flexibilität aber in der Regel vorzuziehen. Weitere Vorteile ergeben sich dadurch, dass alle Systeme beliebige Schnittstellen nutzen können. Funktionalität und Daten müssen somit nicht mehr redundant in lokalen Systemen vorgehalten werden. Der Dienstschnitt kann auch so erfolgen, dass vorhandene Komponenten oft benötigte Funktionalitäten zusammenfassen. In diesem Fall würde der aufgerufene Service selbst mehrere Komponenten oder auch andere Services aufrufen.

Beziehungen zwischen einzelnen Services werden oft mit den Begriffen Orchestrierung und Choreografie beschrieben. Peltz definiert die ursprünglich auf Web Services bezogenen Begriffe als zwei Arten der Interaktion von Diensten, die aus mehreren weiteren Diensten zusammengesetzt sein können (2003, S. 46f.). Demnach ist die Choreografie durch den Ablauf der meist öffentlich ausgetauschten Nachrichten zwischen einzelnen Kommunikationspartnern, wie etwa betrieblichen Objekten, charakterisiert. Beteiligte Objekte sind gleichberechtigt und selbstständig, wobei sie definieren, wie sie zur Zusammenarbeit beitragen. Die Orchestrierung führt im Gegensatz dazu weitere Services aus, welche den Geschäftsprozess innerhalb eines Kommunikationspartners, z.B. eines betrieblichen Objekts, steuern. Die Orchestrierung entspricht dem Regelungsprinzip betrieblicher Objekte in der SOM-Methodik, die Choreografie dem Verhandlungsprinzip (Ferstl und Sinz 2013, S. 441).

3.3 Gestaltung der Metamodelle

3.3.1 Gestaltung der fachlichen Ebene serviceorientierter Anwendungssysteme

Zur Spezifikation serviceorientierter AwS wird das SOA-Architekturmodell von Teusch und Sinz (2011, S. 293f.) herangezogen (siehe Abbildung 19). Es unterscheidet Services der Klassen Entitäts- und Vorgangs-Service (2011, S. 295). Ein Entitäts-Service stellt demnach Schnittstellen für KOT zur Verfügung, so dass diese gekapselt werden. Attribute und Operatoren eines KOT sind dann über die Schnittstelle von anderen Systemen oder Services zugreifbar. Vorgangs-Services steuern das Zusammenwirken der Entitäts-Services und damit letztendlich die Koordination der KOT. In diesem Fall wird von elementaren Vorgangs-Services gesprochen. Nicht-Elementare Vorgangs-Services können zur Komposition von Services mehrere elementare Vorgangs-Services orchestrieren. Die nicht-elementaren Vorgangs-Services kapseln VOT, die einem betrieblichen Objekt zuzuordnen sind, aber nicht notwendigerweise alle. Beziehungen zu anderen nicht-elementaren Vorgangs-Services bilden eine Choreografie.

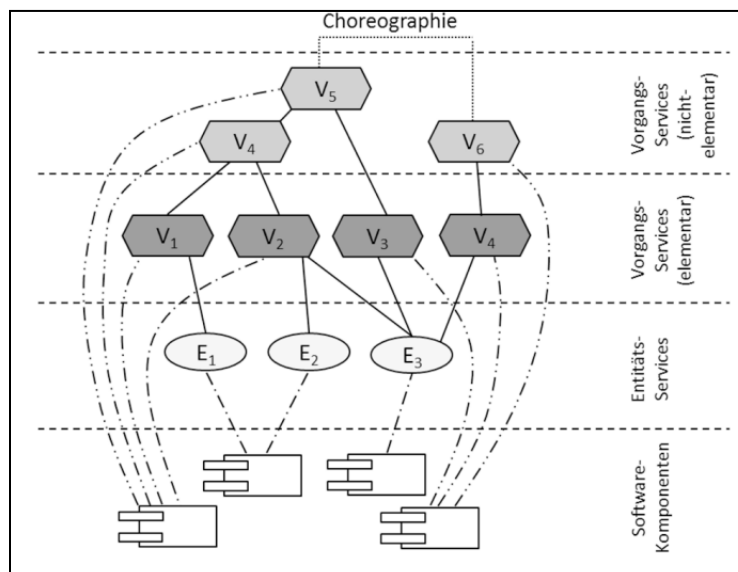

Abbildung 19: SOA-Architekturmodell (Teusch und Sinz 2011, S. 294)

Ziel ist es, ein Metamodell für die Spezifikation serviceorientierter AwS zu entwickeln, welches die Service-Klassen Entitäts- und Vorgangs-Service sowie die bereits diskutierten Merkmale nach Siedersleben berücksichtigt. Da das Metamodell zur Spezifikation von AwS im Ecore-Format (AwS-Ecore-Metamodell) bereits KOT und VOT spezifiziert, bietet es sich an, dieses Metamodell zu erweitern. Neben den

genannten Service-Klassen sind die Hauptmerkmale, die es zu unterstützen gilt, Komponentenorientierung, lose Koppelung und Workflow.

Dazu wird zunächst eine Xtext-Grammatik entwickelt, welche die Grammatik des AwS-Ecore-Metamodells[7] importiert und somit erweitert. Analog zum Vorgehen in Kapitel 2.3.2 definiert diese Grammatik sowohl die abstrakte als auch die konkrete Syntax einer textuellen Modellierungssprache. Anschließend wird mit Xtext aus der erweiterten Grammatik ein neues Metamodell für serviceorientierte AwS generiert (SOA-AwS-Ecore-Metamodell), das konform zu Ecore[8] ist. Aus Gründen der Übersichtlichkeit wird zunächst das generierte Metamodell vorgestellt (siehe Abbildung 20) und danach die Grammatik besprochen.

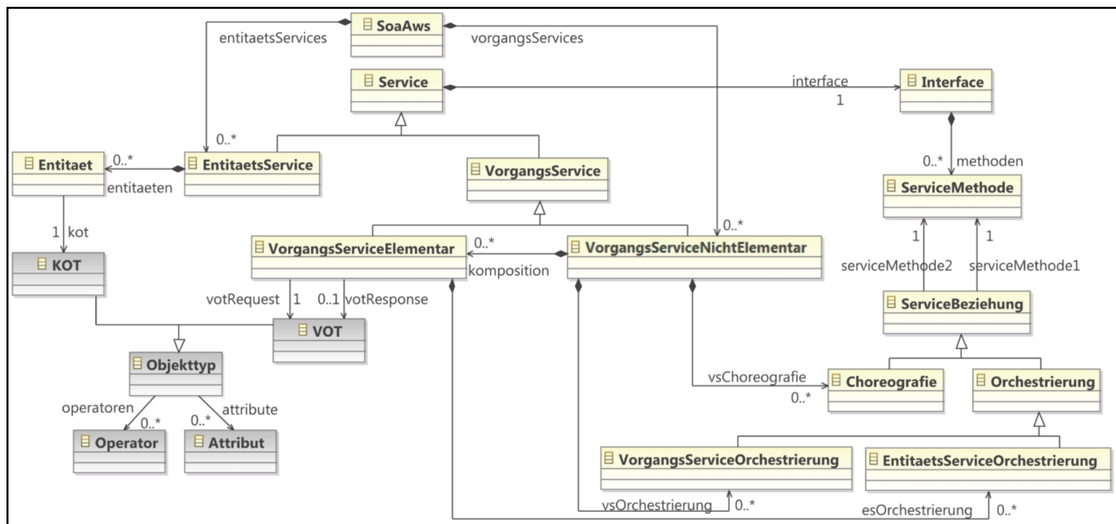

Abbildung 20: SOA-AwS-Ecore-Metamodell

Wie schon das AwS-Ecore-Metamodell, besitzt die Erweiterung für serviceorientierte AwS genau einen Grundbaustein, da das Metamodell als Baumstruktur abgebildet wird[9]. Das Meta-Metamodell stellt das EMF in Form von Ecore bereit.

Die EClass *SoaAws* repräsentiert die gesamte SOA mit allen Teil-AwS und steht über Attributzuweisungsbeziehungen (*has*), dargestellt als *EReference* mit Containment (ausgefüllte Raute), mit den *EClasses VorgangsServiceNichtElementar*, *EntitaetsService* in Verbindung. Das *SoaAws* enthält also beliebig viele (0,*) VorgangsServices und Entitäts-Services. Ein Entitäts-Service ist mit beliebig vielen Entitäten assoziiert, die je ein KOT des KOS kapseln und somit den Zugriff

[7] Siehe Kapitel 2.5.2
[8] Siehe Kapitel 2.3.1
[9] Siehe S. 20, Stichwort *AST*

vereinheitlichen. Der Entitäts-Service ist ein Subtyp von *Service* und besitzt somit, wie jeder Service, genau ein Interface. Das Interface definiert die externen Schnittstellen, mit denen alle Funktionen der gekapselten Objekttypen aufgerufen werden können. Die Schnittstellendefinition einzelner Funktionen ist in Form von Service-Methoden im Interface definiert.

Gemäß dem Service-Modell stellt die zweite Service-Klasse den Vorgangs-Service dar. Dementsprechend besitzt auch der Vorgangs-Service genau ein Interface. Der nicht-elementare Vorgangs-Service enthält beliebig viele elementare Vorgangs-Services (Kardinalität 0,*).

Elementare Vorgangs-Services sind mit einem oder zwei VOT assoziiert, so dass auch dieser Objekttyp für den einheitlichen Zugriff gekapselt wird. Ein einzelner Objekttyp wird gekapselt, sofern die Message Exchange Pattern (W3C 2007) *Input-Only* oder *Output-Only* zum Einsatz kommen (vgl. Teusch und Sinz 2011, S. 299). In diesem Fall sendet oder empfängt der Service eine einzelne Nachricht. Wird eine Anfrage empfangen und darauf geantwortet, so kommt das Pattern *Input-Output* zum Einsatz. In diesem Fall werden zwei VOT für Anfrage und Antwort gekapselt.

Die Bildung von Vorgangsnetzen erfolgt mit Hilfe der nicht-elementaren Vorgangs-Services. Die Ablaufsteuerung zur Orchestrierung mehrerer elementarer VOT wird mit der Service-Beziehung *VorgangsServiceOrchestrierung* realisiert. Diese definiert einen Workflow, der elementare Vorgangs-Services aufruft.

Eine Service-Beziehung stellt eine Beziehung zwischen den Service-Methoden zweier Services her. Die Beziehung ist stets im aufrufenden Vorgangs-Service enthalten. Im Beispiel ist daher die Methode *serviceMethode1*, aus der der Aufruf erfolgt, mit dem Service assoziiert. Ebenfalls assoziiert ist der aufgerufene Service, in dem die Methode *serviceMethode2* ausgeführt wird.

Die Service-Beziehung *Choreografie* eines nicht-elementare Vorgangs-Service definiert den Teil der Choreografie, an dem der nicht-elementare Vorgangs-Service, der die Service-Beziehung beinhaltet, beteiligt ist. Da es sich um eine Service-Beziehung handelt, definiert auch hier die *serviceMethode1* die Methode, von der die Nachrichtenübermittlung ausgeht. Die *serviceMethode2* definiert dann die Methode und damit auch den Service, zu dem die Nachricht übermittelt werden soll.

Die drei Merkmale nach Siedersleben[10] werden in diesem Metamodell auf fachlicher Ebene berücksichtigt. Die Komponentenorientierung wird durch die Kapselung von KOT und VOT in Entitäts- und Vorgangs-Services erreicht. Jeder dieser Objekttypen kann als Komponente separat angesprochen werden. Den Zugriff auf die Komponenten vereinheitlichen externe Schnittstellen. Die Dienstkomposition ist durch die Orchestrierung von elementaren Vorgangs- und Entitäts-Services definiert, sowie durch die Choreografie, die nicht-elementare Vorgangs-Services in Beziehung setzt. Eine solche Beziehung erfüllt auch die Definition der losen Kopplung, dem zweiten Merkmal. Der aufgerufene Service muss zum Zeitpunkt des Aufrufs nicht verfügbar sein. Im Sinne der SOM-Methodik erfolgt die Koordination in diesem Fall nach dem Verhandlungsprinzip. Das dritte Merkmal, der Workflow, ist in Form von Service-Beziehungen enthalten, die Choreografie und Orchestrierung definieren.

Xtext-Grammatik zur Spezifikation serviceorientierter AwS

Analog zur textuellen Spezifikation von KOS und VOS ist auch ein serviceorientiertes AwS mit dem vorgestellten Metamodell in einer dazu entwickelten domänenspezifischen Sprache modellierbar. Die Grammatik definiert die abstrakte und konkrete Syntax dieser Sprache, woraus sich das mit Xtext abgeleitete Metamodell ergibt. Abbildung 64 (siehe Anhang) zeigt die EBNF-ähnliche Grammatik. Sie stützt sich auf die bereits vorgestellten Grammatiken für die Spezifikation des AwS mit KOS und VOS ab.

Sofern in der Grammatik Generalisierungen definiert werden, müssen Attribute des Supertyps in allen Subtypen zugewiesen werden. Beispielsweise definiert der Supertyp Service nur eine Spezialisierung zu Vorgangs- und Entitäts-Service, nicht aber das im Metamodell auftauchende Element Interface. Die Attributzuweisung von Interface erfolgt deshalb separat in allen Subtypen von Service. Xtext erstellt bei der Generierung des Metamodells selbst eine Attributzuweisung für den Supertyp.

3.3.2 Gestaltung des Softwaredesigns auf softwaretechnischer Ebene

Für die Definition des Softwaredesigns wird ein Metamodell erstellt, welches Elemente definiert, mit denen eine zu Java Enterprise Edition 7 (Java EE) kompatible Softwarearchitektur entworfen werden kann.

[10] Siehe Kapitel 3.2

Das Designmodell wird zunächst abstrakt entwickelt und daraufhin wie bisher als Xtext-Grammatik definiert, so dass sich ein Ecore-Metamodell daraus ableiten lässt. Ein einzelnes Anwendungssystem lässt sich nach dem ADK-Strukturmodell (Ferstl und Sinz 2013, S. 321) in Teilsysteme für Kommunikation (K), Anwendungsfunktionen (A) sowie Datenverwaltung (D) aufteilen. Der K-Teil lässt sich weiterhin in maschinelle (K_M) und personelle (K_P) Kommunikation untergliedern.

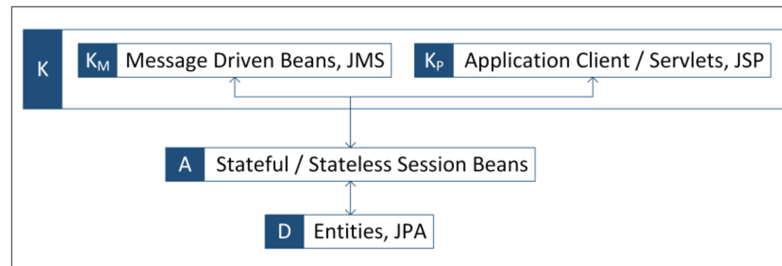

Abbildung 21: ADK-Strukturmodell

Abbildung 21 zeigt, welche Java-EE-Technologien für die Teilsysteme A, D und K genutzt werden. K_M und D werden mit Enterprise Beans (Oracle 2013a, S. 30f.) realisiert. Message Driven Beans realisieren die externen Schnittstellen eines AwS, indem Nachrichten asynchron gesendet und empfangen werden. Die Übermittlung wird später mit dem Java Message Service (JMS) (Oracle 2013a, S. 135) realisiert, über den Nachrichten an Queues verschickt werden. Die personelle Schnittstelle K_P wird, sofern benötigt, von einem Application Client oder mit Servlets und Java Server Pages (JSP) (Oracle 2013a, S. 223f.) umgesetzt.

Die Java-EE-Architektur des serviceorientierten Gesamtsystems ist in Abbildung 22 dargestellt; sie richtet sich nach dem SOA-Architekturmodell aus Abbildung 19. Nicht-elementare Vorgangs-Services tauschen Nachrichten über Message Driven Beans aus und realisieren auf diese Weise Choreografie-Beziehungen. Eine dem Vorgangs-Service zugehörige Stateful Session Bean orchestriert elementare Vorgangs-Services gemäß dem Workflow, der im Rahmen der AwS-Spezifikation definiert wurde. Er ergibt sich aus den definierten Service-Beziehungen, die Übergänge zwischen Aktivitäten darstellen.

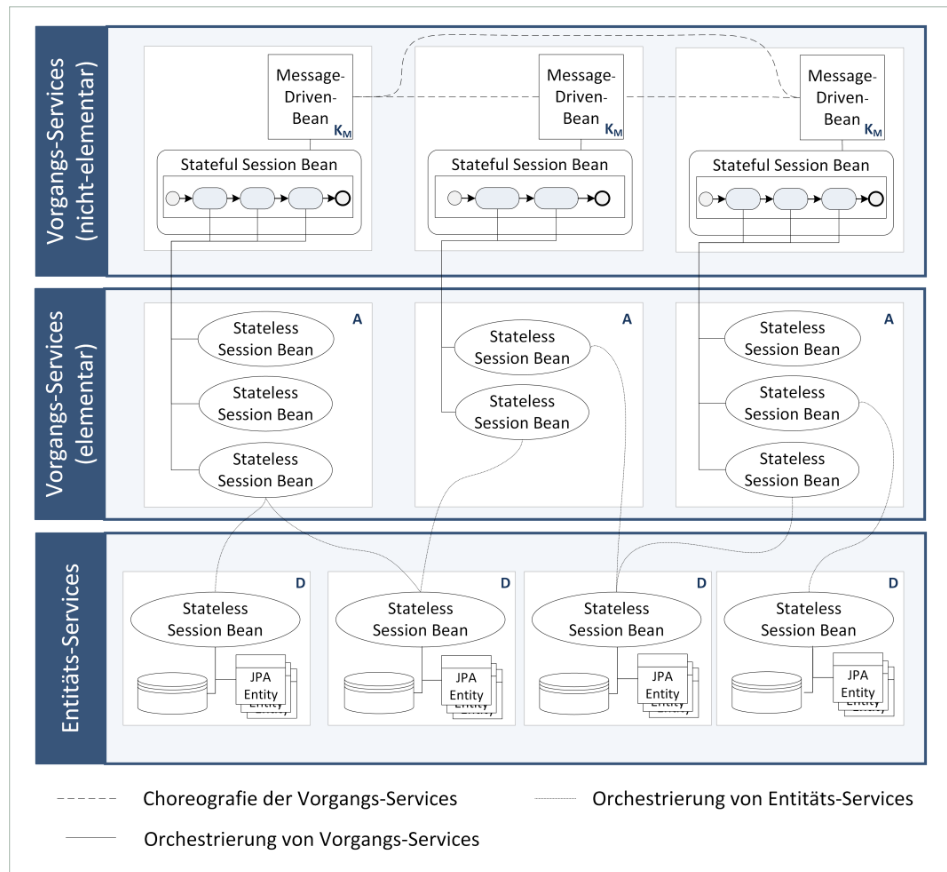

Abbildung 22: Architekturmodell für Java EE

In der Regel wird das Workflow-Management von einer dedizierten Workflow-Engine eines Workflow-Management-Systems (WfMS) übernommen, welches die Workflow-Definitionen ausführt. Der vorliegende Ansatz verwendet jedoch kein dediziertes WfMS, sondern implementiert den Workflow in einer Stateful Session Bean. Dies ist möglich, da der Workflow zwar hart codiert ist, aber im Zuge der Modelltransformation jederzeit neu generiert werden kann.

Elementare Vorgangs-Services beinhalten Anwendungsfunktionen. Sie sind durch Stateless Session Beans implementiert und koordinieren Entitäts-Services in Form einer Orchestrierung. Entitäts-Services sind ebenso als Stateless Session Beans implementiert. Sie verwalten Entities, welche mittels Java Persistence API (Oracle 2013b) in einem Datenbanksystem persistiert werden.

Für das zu entwickelnde Metamodell des Softwaredesigns ergibt sich damit die Anforderung, die genannten Java-Konzepte zu unterstützen. Das aus einer Xtext-Grammatik generierte Ecore-Metamodell ist in Abbildung 23 zu sehen. Der Grundbaustein des Modells ist die EClass *Design*, welche aus EJB-Modulen besteht. Ein solches *EJBModule* kann EJBs oder Business Interfaces beinhalten. Bei EJBs

handelt es sich um Session Beans, untergliedert in Stateful und Stateless Beans, sowie Message Driven Beans. Ein Business Interface besteht aus Methodensignaturen, die als Attribute einen Namen, den Typ des Rückgabewertes und die Typen der Parameter besitzen. Methoden besitzen diese Attribute ebenso, beinhalten aber zusätzlich noch Service-Aufrufe (Invoke). Ein solcher Aufruf kann synchron oder asynchron erfolgen. Für die Modellierung eines synchronen Aufrufs wird ein Objekt angegeben, auf dem eine Methode, definiert durch eine Methodensignatur, aufgerufen wird. Ein asynchroner Aufruf versendet eine Nachricht an eine Message Queue. Neben der Queue wird außerdem die Angabe der zu versendenden Typen der Nachrichten benötigt. Die Queue selbst wird im Metamodell als Attribut der Message Driven Bean modelliert, so dass sie der Bean fest zugeordnet ist.

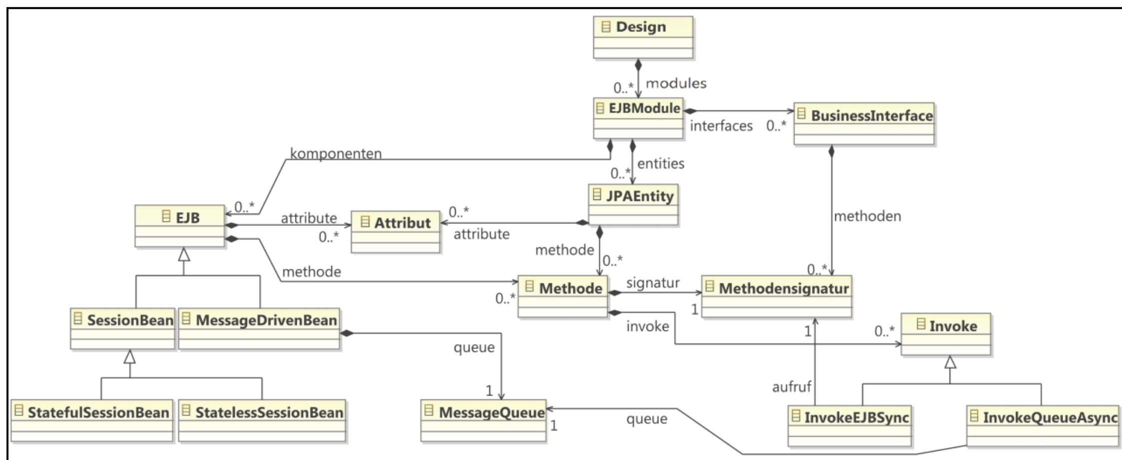

Abbildung 23: Metamodell des Softwaredesigns

Die Grammatik zur textuellen Modellierung spielt auf dieser Ebene eine untergeordnete Rolle. Da ein Ecore-Modell aus dem im vorherigen Kapitel vorgestellten Metamodell direkt abgeleitet wird, besteht keine Notwendigkeit zur textuellen Modellierung. Das transformierte Ecore-Modell kann direkt in Eclipse mit einem integrierten Editor bearbeitet werden, wobei prinzipiell auch eine textuelle Modellierung möglich ist. Der Vollständigkeit halber ist die zugrundeliegende Grammatik in Abbildung 65 des Anhangs zu finden.

3.4 Spezifikation der Modelltransformationen

3.4.1 Von der Spezifikation serviceorientierter Anwendungssysteme zum Softwaredesign

Der vorliegende Abschnitt erläutert die Regeln zur Transformation der Spezifikation serviceorientierter AwS, die gemäß des Metamodells aus Abschnitt 3.3.1 definiert sind. Das Ziel-Metamodell der Transformation ist auf Ebene des Softwaredesign in Abschnitt 3.3.2 definiert. Das Vorgehen folgt der Metamodell-Transformation der MDA (OMG 2003a, S. 3-9). Die Spezifikation der Transformation erfolgt durch das Herstellen von Beziehungen zwischen beiden Metamodellen. Somit wird ein Beziehungs-Metamodell (Sinz 2002, S. 1058) definiert.

Abbildung 24 zeigt die Beziehungen zwischen den beiden Metamodellen. Einzelne Zuordnungsbeziehungen sind gemäß Abschnitt 2.4.2 in QVTo definiert. Die wichtigsten Zuordnungen werden nun anhand der QVTo-Spezifikation erläutert[11].

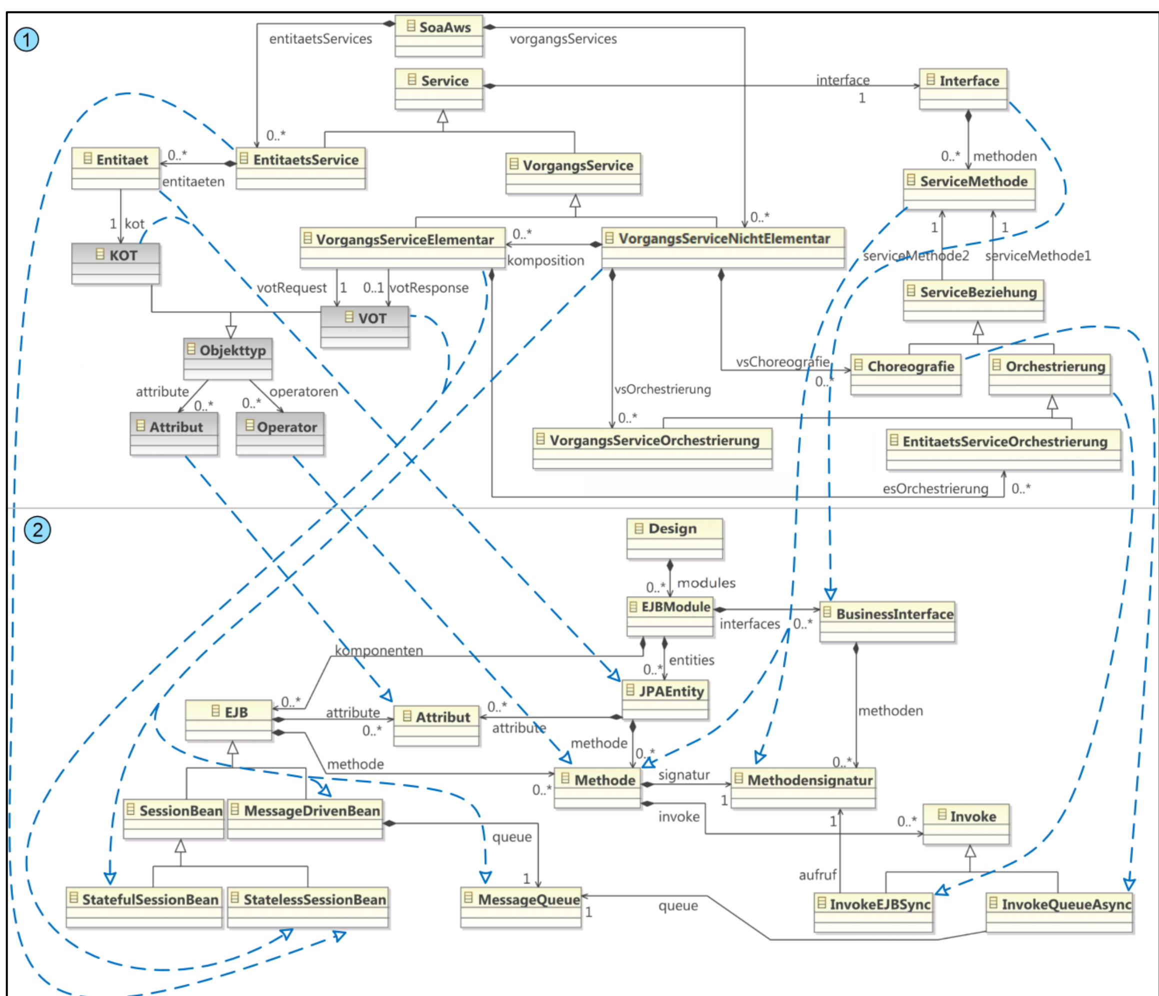

Abbildung 24: Beziehungen zwischen den Metamodellen SOA-AwS-Spezifikation (1) und Softwaredesign (2)

[11] Die vollständige Spezifikation ist mit Kommentaren auf DVD enthalten.

```
main() { source.rootObjects()[SoaAws] -> map design(); }
mapping SoaAws::design() : Design {
  result.name := self.aws.name;
  result.modules += self.entitaetsServices -> map ejbModule();
  result.modules += self.vorgangsServices -> map ejbModule();    }
```

Abbildung 25: Beginn der QVTo-Transformationsspezifikation

Auf oberster Ebene der QVTo-Spezifikation erfolgt die Zuordnung zwischen den Meta-Objekten *SoaAws* und *Design* (Abbildung 25). Die gezeigte Funktion übernimmt den Namen des AwS und ruft Funktionen zur Transformation von Entitäts- und Vorgangs-Services auf, die jedem Service je ein EJB-Modul zuordnen.

Transformation von Entitäts-Services

Die Funktion zur Zuordnung eines Entitäts-Services zu einem EJB-Modul (Abbildung 26, Nr. 1) beinhaltet neben Namenszuweisungen den Aufruf weiterer Zuordnungsfunktionen, die das im Entitäts-Service enthaltene Interface und die Entitäten in ein Business Interface bzw. JPA-Entities transformieren. Der Entitäts-Service selbst wird in eine Stateless Session Bean gewandelt, welche den Service realisiert.

Die Funktion zur Zuordnung von Interface nach Business Interface (2) erhält Name und Package als Parameter; einzelne im Interface definierte Methoden werden zu Methodensignaturen transformiert. Bei der Transformation einer Entität (3), die einen KOT kapselt, wird eine JPA-Entity erzeugt. Letztere übernimmt Attribute und Operatoren des gekapselten KOT in die Elemente Attribut bzw. Methode.

```
// 1: Zuordnungsfunktion: Entitäts-Service -> EJB-Modul
mapping EntitaetsService::ejbModule() : EJBModule {
  result.name := "EntitaetsService_" + self.name;
  var package := "entitaetsservices." + self.name.toLowerCase();
  result.interfaces += self.interface -> map businessInterface(self.name, package);
  result.entities += self.entitaeten -> map jpaEntity(package);
  result.ejb += self -> map statelessSessionBean(package);      }
// 2: Zuordnungsfunktion: Interface -> Business Interface
mapping Interface::businessInterface(name:String,package:String) : BusinessInterface {
  result.name := name;
  result.package := package;
  result.methoden += self.methoden -> map methodensignatur();    }
// 3: Zuordnungsfunktion: Entität -> JPA-Entity
mapping Entitaet::jpaEntity(package:String) : JPAEntity {
  result.name := self.kot.name;
  result.package := package + ".entitaeten";
  result.attribute += self.kot.attribute -> map attribut();
  result.methode += self.kot.operatoren -> map methode();       }
// 4: Zuordnungsfunktion: Entitäts-Service -> Stateless Session Bean
mapping EntitaetsService::statelessSessionBean(package:String) : StatelessSessionBean
inherits Service::ejb {
  result.methode += self.interface.methoden -> map methode();    }
```

Abbildung 26: Zuordnungsfunktionen für Entitäts-Services

Die Funktion zur Erzeugung der Stateless Session Bean aus einem Entitäts-Service (4) erbt die Zuordnungen von Namen und Paketbezeichner aus einer generalisierten Funktion, die *Service* auf *EJB* abbildet. Ebenso transformiert werden die Methoden des Business Interface, da die Session Bean diese implementieren muss. Eine auf diese Weise erzeugte Methode ist nur mit der Signatur aus dem entsprechenden Interface definiert und enthält keinen weiteren Inhalt. Ein Lösungsverfahren ist auf Ebene der Softwareartefakte zu implementieren.

Transformation von Vorgangs-Services

Die Zuordnungsfunktion für nicht-elementare Vorgangs-Services (Abbildung 27, Nr. 1) fügt dem erstellten EJB-Modul ein Business Interface hinzu. Außerdem wird jeder enthaltene elementare Vorgangs-Service transformiert. Dem EJB-Modul werden die Business Interfaces ausgehend von Interfaces der enthaltenen Vorgangs-Services hinzugefügt, sowie Stateless Session Beans, die die eben genannten Vorgangs-Services implementieren. Der nicht-elementare Vorgangs-Service wird anschließend zu einer Message Driven Bean und einer Stateful Session Bean transformiert. Die Stateful Session Bean implementiert den Vorgangs-Service, der die eben erzeugten Stateless Session Beans gemäß der Workflow-Definition orchestriert.

```
// 1: Zuordnungsfunktion: Vorgangs-Service (n.-elem.) -> EJB-Modul
mapping VorgangsServiceNichtElementar::ejbModule() : EJBModule {
  result.name := "VorgangsService_" + self.name;
  var package := "vorgangsservices." + self.name.toLowerCase();
  result.interfaces += self.interface -> map businessInterface(self.name,package);
  self.komposition -> forEach ( service ) {
    result.interfaces += service.interface -> map businessInterface(service,package);
    result.ejb += service -> map statelessSessionBean(package); };
  result.ejb += self -> map messageDrivenBean(package);
  result.ejb += self -> map statefulSessionBean(package);                    }
// 2: Zuordnungsfunktion: Vorgangs-Service (elem.) -> Stateless Session Bean
mapping VorgangsServiceElementar::statelessSessionBean(package:String) :
StatelessSessionBean inherits Service::ejb {
  self.esOrchestrierung -> forEach( orch ) {
    if ( queryAttributInBean(orch.service,result) = null ) then {
       result.attribute += attributService(orch.service); } endif };
  result.attribute += self.votRequest.attribute -> map attribut();
  result.methode += self.votRequest.operatoren -> map methode();
  if ( self.votResponse != null ) then {
     result.attribute += self.votResponse.attribute -> map attribut();
     result.methode += self.votResponse.operatoren -> map methode(); } endif;  }
// 3: Zuordnungsfunktion: Vorgangs-Service (n.-elem.) -> Message Driven Bean
mapping VorgangsServiceNichtElementar::messageDrivenBean(package:String) :
MessageDrivenBean inherits Service::ejb {
  result.methode += self.interface.methoden -> map methode();
  result.queue := self -> map messageQueue()![MessageQueue];
  result.attribute += attributService(self);                                 }
```

Abbildung 27: Zuordnungsfunktionen für Vorgangs-Services

Die Message Driven Bean ermöglicht die asynchrone Kommunikation mit anderen nicht-elementaren Vorgangs-Services und führt damit ihren Teil der Choreografie aus.

Die Zuordnungsfunktion für elementare Vorgangs-Services (Abbildung 27, Nr. 2) erzeugt eine Stateless Session Bean, die den Service implementiert. Neben den geerbten Zuordnungsbeziehungen der generalisierten Funktion, die einen Service auf eine EJB abbildet, beinhaltet die resultierende Session Bean zunächst eine *forEach*-Schleife mit einer *if*-Kontrollstruktur. Diese fügt der Session Bean ein Attribut für jeden orchestrierten Entitäts-Service hinzu, so dass eine Referenz auf den Service entsteht. Das Attribut wird allerdings nur dann hinzugefügt, wenn es noch nicht in der Session Bean enthalten ist. Um dies zu prüfen, wird in der Bedingung eine Abfrage *queryAttributInBean* ausgeführt, die das betreffende Attribut der Session Bean heraussucht und zurückgibt. Ist der Rückgabewert *null*, existiert das Attribut noch nicht und kann hinzugefügt werden. Ohne Prüfung würde das gleiche Attribut mehrmals erstellt werden, wenn es mehrere Orchestrierungsbeziehungen zu einem Service gibt. Anschließend werden Attribute und Operatoren der VOT transformiert. Sofern das Message Exchange Pattern *Input-Output* (W3C 2007) verwendet wird, erfolgt die Zuordnung für die zwei VOT *votRequest* und *votResponse.* Andernfalls kapselt der Service nur den VOT, der dann alleine transformiert wird.

Wie erwähnt, werden Message Driven Beans zur asynchronen Kommunikation für jeden nicht-elementaren Vorgangs-Service erstellt (Abbildung 27, Nr. 3). Nachrichten, die von einem solchen Service entgegengenommen werden, sind durch die Methodensignaturen im Interface des Vorgangs-Service definiert. Die Zuordnungsfunktion zur Ableitung von Message Driven Beans fügt daher für jede Methodensignatur des Interface eine Methode ein. Die Methode wird bei Eintreffen einer entsprechenden Nachricht aufgerufen und enthält selbst einen synchronen Aufruf der gleichnamigen Methode des Vorgangs-Service (nicht-elementar). Letzterer startet einen Workflow, der geeignete elementare Vorgangs-Services für die Durchführung des Vorgangs orchestriert. Um überhaupt Nachrichten entgegennehmen zu können, muss außerdem eine Message Queue erstellt werden, die in der Message Driven Bean referenziert wird. Des Weiteren muss letztgenannte EJB auch eine Referenz zum Aufruf der Methoden des Vorgangs-Service besitzen, für die deshalb ein Attribut erstellt wird.

Nicht-elementare Vorgangs-Services werden mittels einer entsprechenden Zuordnungsfunktion (Abbildung 28) zu Stateful Session Beans transformiert. Zunächst werden die im Interface des nicht-elementaren Vorgangs-Service angebotenen Methoden transformiert, wobei die Signatur der konvertierten Methoden aus dem Interface übernommen wird. Da der Vorgangs-Service weitere elementare Vorgangs-Services orchestriert, wird anschließend ein Attribut für jeden zu orchestrierenden Service und die Message Driven Bean erstellt, um Referenzen für den Zugriff zu definieren.

```
// Zuordnungsfunktion Vorgangs-Service (n.-elem.) -> Stateful Session Bean
mapping VorgangsServiceNichtElementar::statefulSessionBean(package:String) :
StatefulSessionBean inherits Service::ejb {
  result.methode += self.interface.methoden -> map methode();
  result.attribute += attributeOrchestrierteServices(self);
  result.attribute += attributMessageDrivenBean(self);
  self.vsChoreografie -> forEach(beziehung) {
    var methode1 := queryMethodeInBean(beziehung.serviceMethode1,result);
    if (methode1 = null) then {
        methode1 := beziehung.serviceMethode1 -> map methode()![Methode];
        result.methode += methode1; } endif;
    methode1.invoke += beziehung -> map invokeQueueAsync();  };
  self.vsOrchestrierung -> forEach(beziehung) {
    var methode1 := queryMethodeInBean(beziehung.serviceMethode1,result);
    var methode2 := queryMethodeInBean(beziehung.serviceMethode2,result);
    if (methode1 = null) then {
        methode1 := beziehung.serviceMethode1 -> map methode()![Methode];
        result.methode += methode1; } endif;
    if (methode2 = null) then {
        methode2 := beziehung.serviceMethode2 -> map methode()![Methode];
        result.methode += methode2; } endif;
    methode1.invoke += beziehung -> map invokeEjbSync();     };             }
```

Abbildung 28: Zuordnung nicht-elementarer Vorgangs-Services zu Stateful Session Beans

In den einzelnen Methoden wird anschließend der definierte Workflow implementiert. Dabei werden zuerst Beziehungen zur Choreografie betrachtet. Eine einzelne Beziehung der Choreografie besteht aus zwei Service-Methoden, wobei die zweite Methode im Interface eines anderen nicht-elementaren Vorgangs-Service definiert ist. Ein Beispiel für die konkrete Syntax einer solchen Beziehung gemäß Grammatik [12] ist „Methode_1 -> Service.Methode_2“. Für jede Beziehung der Choreografie („self.vsChoreografie -> forEach“) wird nun geprüft, ob die erste Methode, von der die Beziehung ausgeht, existiert. Dies ist nur bei Methoden der Fall, die im Interface definiert sind. Um eine doppelte Erstellung zu vermeiden, wird mit der Abfrage *queryMethodeInBean* die betreffende Methode aus der Session Bean herausgesucht. Bei dem Rückgabewert „null“ wurde keine Methode gefunden, so dass die Zuordnungsfunktion von Service-Methode auf Methode zur Transformation

[12] Siehe Abbildung 64 (Anhang)

aufgerufen werden muss. Nachdem die Methoden der Service-Beziehung entweder durch Abfrage ermittelt, oder durch Transformation erstellt worden sind, ist die Beziehung in Form eines Aufrufs („Invoke“) zu realisieren. Bei der Choreografie erfolgt die Kommunikation asynchron. Die erste Methode, von der die Kommunikation ausgeht, sendet eine Nachricht an die Queue des Kommunikationspartners. Die Auslösung des Versands der Nachricht wird durch einen entsprechenden Aufruf definiert („invokeQueueAsync“). Der Versand wird von der referenzierten Message Driven Bean durchgeführt.

Anschließend werden die Beziehungen zur Orchestrierung betrachet, die gemäß Grammatik[13] in der Form „Methode_1 -> Methode_2“ definiert sind. Für jede solche Beziehung („self.vsOrchestrierung -> forEach“) zwischen zwei Service-Methoden wird analog zu dem Vorgehen im vorigen Absatz zunächst zunächst geprüft, ob beide Methoden bereits transformiert sind, indem die Abfrage *queryMethodeInBean* aufgerufen wird. Im Gegensatz zur besprochenen Choreografie-Beziehung sind hier allerdings beide Methoden in den Interfaces der elementaren Vorgangs-Services definiert, so dass auch beide zu transformieren sind. Wenn beide Methoden der Service-Beziehung entweder durch Abfrage ermittelt, oder durch Transformation erstellt worden sind, wird die Beziehung wieder in Form eines Aufrufs („Invoke“) realisiert. Die Beziehung zur Orchestrierung stellt eine Reihenfolgebeziehung dar, welche zwischen den beiden assoziierten Methoden besteht. Ein entsprechender Aufruf der zweiten Methode nach der ersten hat aufgrund der Orchestrierung hierarchischen Charakter. Die Kommunikation mit dem aufzurufenden EJB erfolgt deshalb synchron („invokeEjbSync“).

3.4.2 Vom Softwaredesign zum Softwareartefakt

Der letzte Transformationsschritt besteht aus der Generierung von Softwareartefakten. Ecore-Modelle, die zu dem definierten Metamodell des Softwaredesigns konform sind, können als Eingabe für ein individuell entwickeltes Generator-Werkzeug genutzt werden. Letzteres generiert Softwareartefakte auf Basis von Vorlagen.

[13] Siehe Abbildung 64 (Anhang)

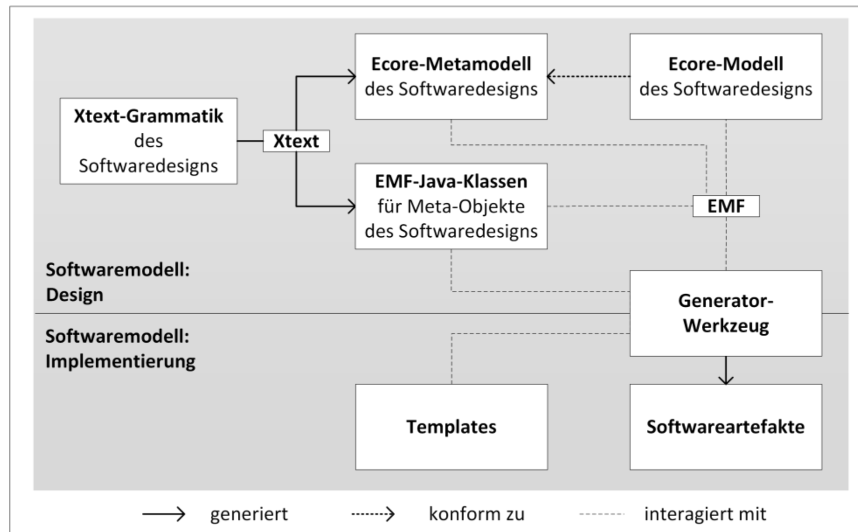

Abbildung 29: Schema zur Generierung von Softwareartefakten

Die Transformation ist in Abbildung 29 schematisch dargestellt. Das Ecore-Modell des Softwaredesign ist konform zu dem aus der Xtext-Grammatik generierten Ecore-Metamodell. Xtext generiert außerdem Java-Klassen für alle Meta-Objekte des genannten Metamodells, die Klassenbibliotheken des EMF nutzen. Das EMF dient außerdem als Schnittstelle, über die das Generator-Werkzeug auf Metamodell und Modell zugreifen kann. So wird ein im XMI-Format vorliegendes Ecore-Modell durch das EMF eingelesen. Das konkrete Modell instanziiert die vorher erzeugten Java-Klassen, so dass das Modell dann in Form von Java-Objekten vorliegt. Das Generator-Werkzeug ordnet den einzelnen Objekten entsprechend ihrer Klasse mindestens ein Template zu und generiert schließlich Quellcode, indem variable Bestandteile der Vorlage mit Informationen aus dem Modell gefüllt werden.

Für die Definition der Vorlagen kommt die Programmiersprache Xtend [14] zum Einsatz. Sämtliche Vorlagen sind als Template-Ausdruck in Xtend-Klassen definiert. Klassen, die eine Vorlage repräsentieren, implementieren eine einheitliche Schnittstelle, welche von einer abstrakten Basisklasse geerbt wird. Die Basisklasse bringt Attribute und Methoden zur Festlegung des Speicherorts nach der Generierung mit, während das Interface eine Methode zur Generierung von Text vorschreibt.

Die Zuordnung von Objekten des Metamodells des Softwaredesigns zu Template-Klassen ist in Tabelle 1 aufgelistet. Weitere Hilfsklassen können dem Klassendiagramm in Abbildung 66 (siehe Anhang) entnommen werden. Die einzelnen Zuordnungen werden nun kurz beschrieben.

[14] Siehe Kapitel 2.4.2

<table>
<tr><th>Nr.</th><th>Meta-Objekt des SW-Designs</th><th>Template-Klasse für SW-Artefakte</th></tr>
<tr><td>1</td><td>EJBModule</td><td>EjbModuleTemplate</td></tr>
<tr><td>2</td><td>BusinessInterface</td><td>BusinessInterfaceTemplate</td></tr>
<tr><td rowspan="2">3</td><td rowspan="2">SessionBean</td><td>AbstractSessionBeanTemplate</td></tr>
<tr><td>SessionBeanTemplate</td></tr>
<tr><td>4</td><td>MessageDrivenBean</td><td>MessageDrivenBeanTemplate</td></tr>
<tr><td>5</td><td>JPAEntity</td><td>JpaEntityTemplate</td></tr>
</table>

Tabelle 1: Zuordnung von Meta-Objekten des SW-Designs auf Template-Klassen für SW-Artefakte

1. **EJB-Module**: Das Template erstellt ein Eclipse-Projekt, das ein entsprechend benanntes Modul beinhaltet. Jedes Modul enthält Session Beans zu einem nicht-elementaren Vorgangs-Service oder einem Entitäts-Service. Um Kommunikation mit anderen Services zu ermöglichen, werden außerdem Schnittstellendefinitionen anderer Services und Entitäten, die als Datentransferobjekte dienen, generiert. Hierfür werden die entsprechend benannten Templates aufgerufen.
2. **Business Interface**: Das Template generiert ein Java-Interface, das mittels Annotation als Remote-Interface zur Schnittstellendefinition der Services genutzt wird.
3. **Session Bean**: Zur besseren Erweiterbarkeit wird das Muster der dreistufigen Vererbung verwendet (Stahl et al. 2007, S. 161). Die erste Stufe wird durch eine nicht generierte Datei definiert. Sie wird lediglich in jedes Projekt kopiert und beinhaltet gemeinsame Funktionalität für alle Session Beans. Die beiden Templates zur Session Bean entsprechen der zweiten und dritten Stufe der Vererbungshierarchie. Das erste Template, welches mit dem Begriff „Abstract“ beginnt, generiert eine abstrakte Klasse, die von der im vorherigen Absatz beschriebenen Klasse erbt. Die Methoden dieser Klasse beinhalten die eigentliche Funktionalität der Session Bean. Die durch das zweite Template generierte Klasse erbt wiederum von der eben beschriebenen abstrakten Klasse. Auf dieser dritten Stufe der Vererbungshierarchie kann ein Entwickler nun manuell Erweiterungen vornehmen. Die Klasse wird daher ohne Inhalt generiert und wird nicht überschrieben, sofern sie bei der Generierung bereits existiert. Die dreistufige Vererbung schafft damit einen definierten Punkt zur Erweiterung generierter Methoden. Die abstrakten Klassen müssen daher nicht geändert werden, so dass sich generierter Quelltext nicht mit manuell geschriebenem vermischt.

4. **Message Driven Bean:** Das Template generiert eine entsprechend annotierte Klasse, die das Interface *MessageListener* implementiert. Sie enthält Methoden zum Versand und zur Empfang von Nachrichten aus Message Queues.
5. **JPA-Entity:** Das Template für Entitäten erzeugt eine normale Java-Klasse mit Java-EE-Annotationen, die die Verwendung mit der JPA ermöglichen.

Diese Vorlagen werden von dem ebenfalls in Xtend entwickelten Generator-Werkzeug instanziiert, dessen Klassenstruktur in Abbildung 67 (siehe Anhang) dargestellt ist.

Die Generator-Klasse initialisiert die Anwendung und steuert die Generierung auf höchster Ebene. Die Klasse lädt mit Hilfe des EMF zunächst ein Ecore-Modell, für das Quelltext zu generieren ist. Die durch das EMF zur Verfügung gestellten Referenzen auf Java-Objekte der Modellinstanz werden daraufhin durchlaufen. Auf oberster Ebene handelt es sich dabei um Instanzen der Klasse EJB-Module, die Instanzen von Session Beans, Message Driven Beans, Interfaces und Entities beinhalten können. Die Klasse *JavaEEFileGenerator* instanziiert für ein vorliegendes Java-Objekt das der jeweiligen Klasse zugeordnete Template. Weitere Konfigurationsdateien werden mit der Klasse *DeploymentFileGenerator* erzeugt.

Zur Verwaltung von Namespaces, die hier den Namensraum einzelner Java-Pakete definieren, werden die Klassen *Namespace* und *NamespaceManager* genutzt. Template-Instanzen können mit dem *TemplateFileWriter* als Datei ausgegeben werden.

Die Bedienung des Werkzeugs wird am Beispiel der Fallstudie im nachfolgenden Kapitel erläutert.

4 Fallstudie im Rahmen eines e-Car-Szenarios

4.1 Einführung in das Teilszenario „Absatz von e-Cars“

Das fiktive wandlungsfähige Wertschöpfungsnetz „e-Car Net“ besteht aus mehreren unabhängigen Unternehmen, die Produkte und Dienstleistungen im Zusammenhang mit Elektroautos anbieten (Leunig et al. 2011, S. 15-35). Vier verschiedene Teilszenarien beschäftigen sich mit der Produktion von e-Cars (1), dem Absatz (2), der Mobile Maintenance (3) und der Mobility Provision (4). Während *Produktion* und *Absatz* für die Aufgaben der entsprechenden betrieblichen Grundfunktionen verantwortlich sind, übernimmt die *Mobile Maintenance* Aufgaben der Wartung und Pannenhilfe von Fahrzeugen. *Mobility Provision* erbringt die Dienstleistung der Mobilität für Endkunden, indem unter anderem e-Cars vermietet werden.

Das zweite Teilszenario, der Absatz von e-Cars (Leunig et al. 2011, S. 23-27), sieht den Verkauf von e-Cars an Endkunden vor. Der Vertrieb erfolgt entweder direkt über einen Web-Shop oder über einen Intermediär. Als Intermediär kommen verschiedene Makler infrage, die e-Cars eigenständig vertreiben. Der Makler erhält ein Verhandlungsmandat und vermittelt daraufhin Aufträge, für die er eine Provision erhält. Neue Aufträge übermittelt der Makler an die Auftragsbearbeitung der e-Car AG. Bestellt der Kunde über den Web-Shop, so wird die Bestellung von dort direkt an die Auftragsbearbeitung weitergeleitet. Für vorliegende Aufträge erteilt die Auftragsbearbeitung einen Produktionsauftrag, der an das Partnerunternehmen *Produktion* übermittelt wird.

Die Individualisierung bestellter e-Cars erfolgt durch Wahl einzelner Ausstattungsmerkmale, die der Kunde entweder selbst oder zusammen mit dem Makler festlegt. Die nachträglich benötigten Individualteile werden, wie zuvor der Auftrag, an die *Produktion* übermittelt. Hierfür entstehen zusätzliche Kosten für den Kunden. Eine entsprechende Forderung wird dem Finanzwesen des *Absatz* mitgeteilt.

Fertige e-Cars werden durch die *Produktion* beim *Absatz* angeliefert und anschließend bereitgestellt.

4.2 Modellierung der Geschäftsprozessebene

Das initiale Interaktionsschema (IAS) stellt das Geschäftsprozessmodell aus Struktursicht dar (Abbildung 30, vgl. Leunig et al. 2011, S. 25). Die gezeigten Objekte und Transaktionen werden anschließend weiter verfeinert (siehe Abbildung 31).

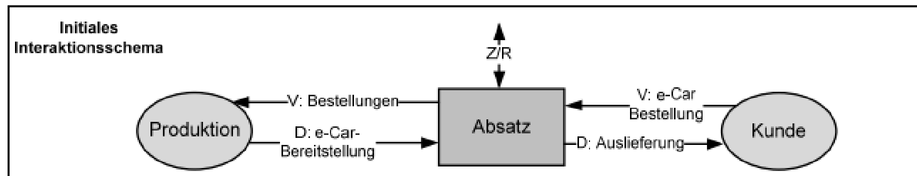

Abbildung 30: Interaktionsschema zum Absatz von e-Cars (Leunig et al. 2011, S. 25)

Der *Absatz* wird zunächst nach dem Verhandlungsprinzip zerlegt. Die Zerlegung ergibt neben dem *Finanzwesen* ein Objekt zur Bearbeitung von Aufträgen, welches nach der Zerlegung gemäß Regelungsprinzip in die Objekte *Auftragsabwicklung* als Regler und *Auslieferung* als Regelstrecke zerfällt. Eine weitere Zerlegung der *Auftragsabwicklung* nach dem Verhandlungsprinzip führt zu den Objekten *Auftragsbearbeitung* und *Vertrieb*, wobei letzterer noch in Objekte für die beiden Vertriebskanäle untergliedert wird.

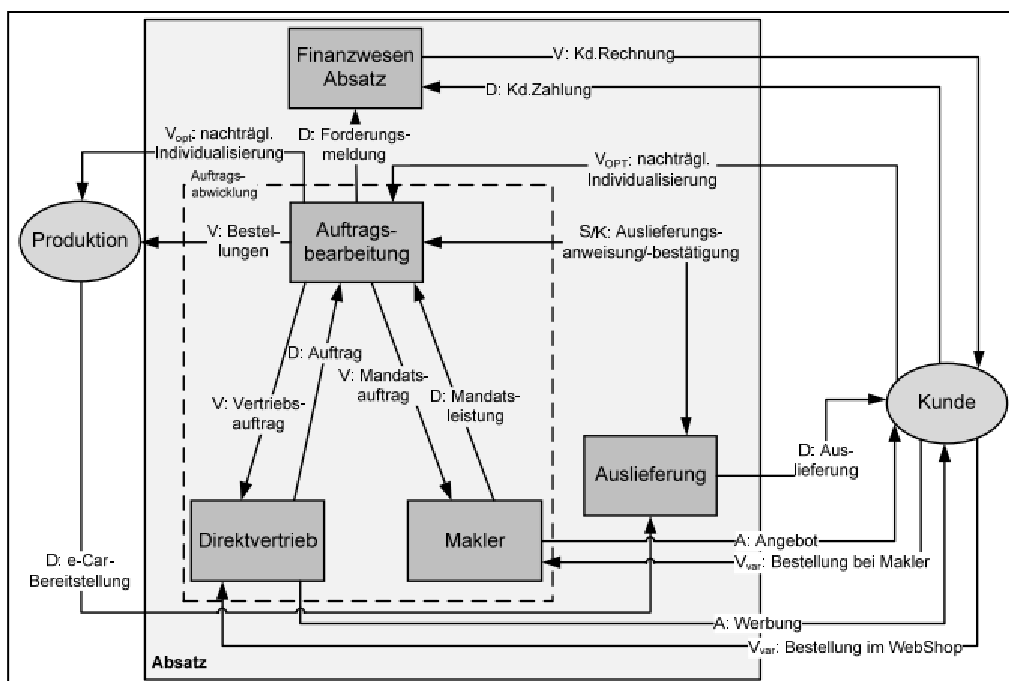

Abbildung 31: Interaktionsschema zum Absatz von e-Cars, Zerlegung (Leunig et al. 2011, S. 25)

Die Transaktion *e-Car Bestellung* deckt nach der Zerlegung weitere Transaktionen für die beiden Vertriebskanäle auf, die der Bestellung beim *Makler* oder einem Web-Shop dienen. Die Möglichkeit zur nachträglichen Individualisierung spiegelt sich in

Transaktionen zwischen *Kunde* und *Auftragsbearbeitung* sowie *Auftragsbearbeitung* und *Produktion*. Zwischen *Finanzwesen* und *Kunde* ergeben sich Transaktionen für Rechnung und Zahlung. Schließlich erfolgt die Auslieferung des von der *Produktion* bereitgestellten Fahrzeugs für den Kunden in Form einer Durchführungstransaktion.

Neben der Darstellung der Struktursicht in Form eines IAS wird das Geschäftsprozessmodell in Abbildung 68 (siehe Anhang) außerdem aus Verhaltenssicht in Form eines VES gezeigt. Die Ebene der Geschäftsprozesse ist damit modelliert.

4.3 Spezifikation von Anwendungssystemen

4.3.1 Entwicklung des konzeptuellen Objektschemas und des Vorgangsobjektschemas

Auf Grundlage des Geschäftsprozessmodells erfolgt nun die Spezifikation von AwS. Nach der Abgrenzung einzelner AwS wird die Spezifikation in Form von KOS und VOS aus IAS bzw. VES abgeleitet und anschließend überarbeitet.

Abgrenzung von Anwendungssystemen

Im Rahmen der Abgrenzung werden einzelne betriebliche Objekte einem AwS zugeordnet, sofern die Durchführung der zugehörigen betrieblichen Aufgabe ein AwS erfordert. Dies ist der Fall, sofern die entsprechende Aufgabe vollständig oder teilweise automatisiert ist. Für den Absatz von e-Cars ergeben sich Anwendungssysteme für den Direktvertrieb, die Auftragsbearbeitung, das Finanzwesen und die Auslieferung. Alle zugehörigen betrieblichen Aufgaben sind teilautomatisiert. Der Direktvertrieb ist aufgrund der Transaktionen Werbung und Vertriebsauftrag nicht vollständig automatisierbar. Die Auftragsbearbeitung muss u.a. Individualisierungen manuell prüfen und ist daher ebenfalls nicht vollautomatisiert. Das Finanzwesen kann nur die Transaktion Forderungsmeldung automatisiert entgegennehmen und ist ansonsten nicht automatisiert. Dies ist auch bei der Auslieferung der Fall, die über das AwS nur die Transaktion zur Auslieferung erhält. Das betriebliche Objekt Makler ist nicht-automatisiert und wird daher nicht von einem AwS unterstützt. Ansonsten ist für jedes betriebliche Objekt der Diskurswelt ist ein eigenes Anwendungssystem vorgesehen.

Konzeptuelles Objektschema

Für die Ableitung des initialen KOS werden betriebliche Objekte und Transaktionen des IAS in KOT überführt (Ferstl und Sinz 2013, S. 226). Links stehende leistungsspezifische KOT (LOT) ergeben sich aus den initialen Leistungstransaktionen, ebenfalls links angeordnete objektspezifische KOT (OOT) folgen aus den betrieblichen Objekten. Betriebliche Transaktionen führen zu transaktionsspezifischen Objekttypen (TOT), die von den genannten LOT und OOT abhängig sind. Das Schema ist in Abbildung 32 dargestellt.

Das initiale Schema wird nun überarbeitet (Ferstl und Sinz 2013, S. 227), wobei zunächst KOT entfernt werden, deren zugehörige Aufgaben oder Transaktionen nicht automatisiert werden. Anschließend erfolgen die Ermittlung von Kardinalitäten und

die Zuordnung von Attributen, Nachrichtendefinitionen und Operatoren. Zur Vermeidung von Funktions- und Datenredundanz werden einzelne KOT abschließend zusammengefasst.

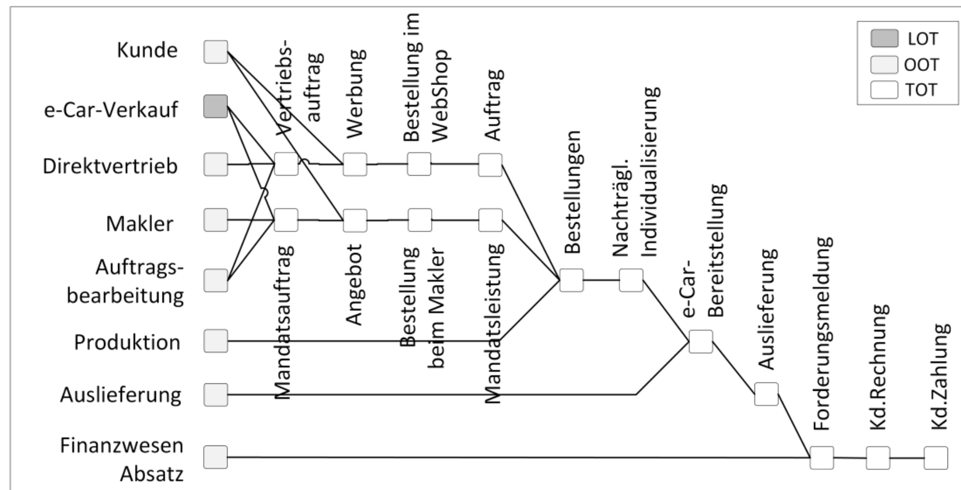

Abbildung 32: Initiales konzeptuelles Objektschema

Die Entfernung von KOT betrifft den OOT *Makler* und die damit in Zusammenhang stehenden TOT *Angebot*, *Mandatsauftrag* und *Mandatsleistung*. Analog dazu entfallen auch die TOT *Werbung* und *Vertriebsauftrag*, die mit dem anderen Vertriebskanal zusammenhängen. Des Weiteren ist die *Produktion* zusammen mit der *Bereitstellung* von e-Cars nicht automatisiert, so dass die entsprechenden KOT wegfallen. Schließlich wird noch der TOT *Kd.Zahlung* entfernt, da das *Finanzwesen* eingehende Zahlungen manuell bearbeitet und mit einem Attribut des KOT *Rechnung* vermerkt.

Um weitere Überarbeitungsschritte direkt an KOT des resultierenden KOS darstellen zu können, wird der letzte Schritt der Überarbeitung, die Zusammenfassung und Umbenennung von KOT, an dieser Stelle in Tabelle 2 vorweggenommen.

	KOT des initialen KOS	**KOT des überarbeiteten KOS**
OOT	Finanzwesen Absatz	Finanzwesen
LOT	e-Car-Verkauf	e-Car-Preisverzeichnis
TOT	Bestellung im Web-Shop	Bestellung
	Auftrag	
	Bestellungen	Produktionsauftrag
	Nachträgl. Individualisierung	Individualisierung
	Auslieferung	Lieferung
	Forderungsmeldung	Forderung
	Kd.Rechnung	Rechnung/Zahlung

Tabelle 2: Umbenennung und Zusammenfassung konzeptueller Objekttypen

Der TOT *Bestellung* des überarbeiteten KOS fasst den TOT zur Bestellung im Webshop sowie den TOT *Auftrag* zusammen, da sich sämtliche Attribute decken. Eine *Bestellung* bezieht sich dabei stets auf die Einheitsplattform (Leunig et al. 2011, S. 19) eines e-Cars, die erst später individualisiert wird. Da keine Wahlmöglichkeit zwischen mehreren Produkten besteht, ist die Bestellung der Einheitsplattform alleine durch den TOT *Bestellung* realisierbar.

Der TOT *Bestellungen*, dessen zugehörige Transaktion die Übermittlung von Aufträgen an die Produktion repräsentiert, wird in *Produktionsauftrag* umbenannt. Die Umbenennung des TOT *Kd.Rechnung* zu *Rechnung/Zahlung* begründet sich durch die Entfernung des TOT *Zahlung*, dessen zugehörigen Transaktion nicht automatisiert ist. Alle weiteren Umbenennungen werden vorgenommen, um die repräsentierten Objekte aus Sicht des AwS konziser zu beschrieben.

Die Kardinalitäten der Beziehungen zwischen einzelnen KOT können Abbildung 33 entnommen werden. Beziehungen, die von unabhängigen linksstehenden KOT ausgehen, besitzen stets die Kardinalität (0,*), ansonsten wird die Kardinalität (0,1) zugeordnet. Ein Kunde kann beispielsweise beliebig viele Bestellungen tätigen.

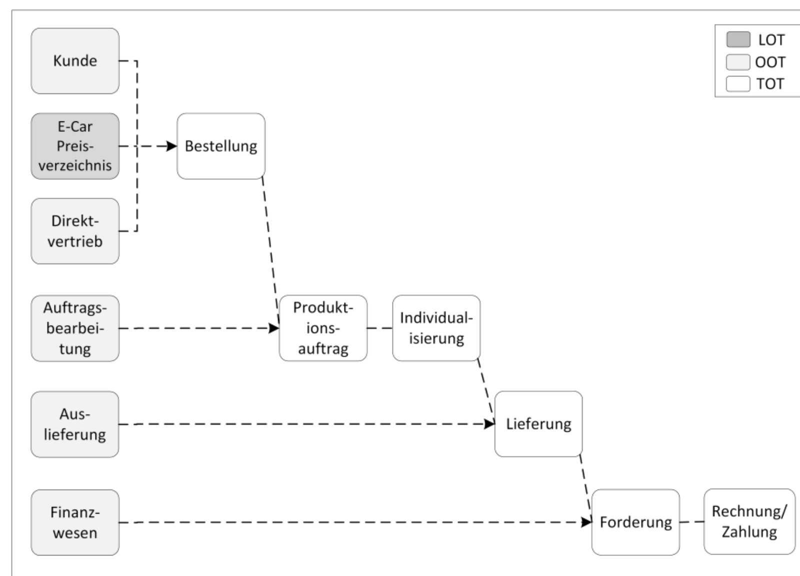

Abbildung 33: Resultierendes konzeptuelles Objektschema

Das resultierende KOS kann nun unter Zuhilfenahme der entwickelten textuellen Notation für das EMF zugänglich gemacht werden. Dabei wird die Zuordnung von Attributen und Operatoren beispielhaft für den OOT *Kunde* erläutert[15].

[15] Das vollständige Modell ist in textueller Notation auf der DVD enthalten.

Abbildung 34 zeigt das konzeptuelle Objektschema in textueller Notation, wobei einzelne OOT, LOT und TOT aus Platzgründen verdeckt sind. Der Inhalt des OOT Kunde ist in einer separaten Umrandung (blau) ausschnittsweise dargestellt.

```
KOS {
  // konzeptuelle Objekttypen
  OOT Kunde {…}
  LOT ECarPreisverzeichnis {…}
  OOT Direktvertrieb {…}
  OOT Auftragsbearbeitung {…}
  OOT Finanzwesen {…}
  OOT Auslieferung {…}
  OOT Produktion {…}

  TOT Bestellung {…}
  TOT Produktionsauftrag {…}
  TOT Individualisierung {…}
  TOT Lieferung {…}
  TOT Forderung {…}
  TOT RechnungZahlung {…}

  // Beziehungen
  Kunde -> Bestellung;
  ECarPreisverzeichnis -> Bestellung;
  Direktvertrieb -> Bestellung;
  Auftragsbearbeitung -> Produktionsauftrag;
  Auslieferung -> Lieferung;
  Finanzwesen -> Forderung;

  Bestellung -- Produktionsauftrag;
  Produktionsauftrag -- Individualisierung;
  Individualisierung -- Lieferung;
  Lieferung -- Forderung;
  Forderung -- RechnungZahlung;
}
```

```
OOT Kunde {
  ID;
  Name;
  Vorname;
  Strasse;
  Hausnummer;
  PLZ;
  Ort;
  Email;
  Telefonnr;
  ID getID();
  Name getName();
  Vorname getVorname();
  Strasse getStrasse();
  Hausnummer getHausnummer();
  PLZ getPLZ();
  Ort getOrt();
  Email getEmail();
  Telefonnr getTelefonnr();
  setName(Name);
  setVorname(Vorname);
  …
}
```

Abbildung 34: Konzeptuelles Objektschema in textueller Notation

Das textuell notierte KOS enthält neben den durch die entsprechenden Schlüsselwörter gekennzeichneten OOT, LOT und TOT noch *interacts_with*-Beziehungen, bei denen die Kardinalität (0,*) durch einen Pfeil und die Kardinalität (0,1) durch zwei Striche abgekürzt wird.

Der OOT *Kunde* enthält die Definition von Attributen, wie etwa ID, Name oder Vorname. Neben Attributen sind außerdem Operatoren definiert, die Eingabeparameter in runden Klammern sowie einen Ausgabeparameter besitzen können. Der Ausgabeparameter steht, sofern vorhanden, vor der Bezeichnung des Operators [16]. Ein- und Ausgabeparameter definieren Nachrichten, die von den Operatoren empfangen bzw. gesendet werden können. Bei den hier definierten Operatoren handelt es sich um Getter- und Setter-Operatoren, die die entsprechenden Attribute auslesen und zurückgeben bzw. übergeben und speichern.

[16] Siehe Grammatik in Kapitel 2.5.1

Vorgangsobjektschema

Bei der Ableitung des Vorgangsobjektschemas (VOS) werden betriebliche Aufgaben auf Vorgangsobjekttypen (VOT) abgebildet, die das Zusammenwirken von KOT zur Durchführung der jeweiligen Aufgabe beschreiben (Ferstl und Sinz 2013, S. 230f.).

Das initiale VOS besitzt jeweils einen VOT für jede Aufgabe eines VES-Ausschnitts sowie eine *interacts_with*-Beziehung für jede Ereignisbeziehung. Aus Gründen der Übersichtlichkeit werden an dieser Stelle bereits resultierende VOS gezeigt, welches analog zum KOS aus einer mehrschrittigen Überarbeitung entstehen (Ferstl und Sinz 2013, S. 231).

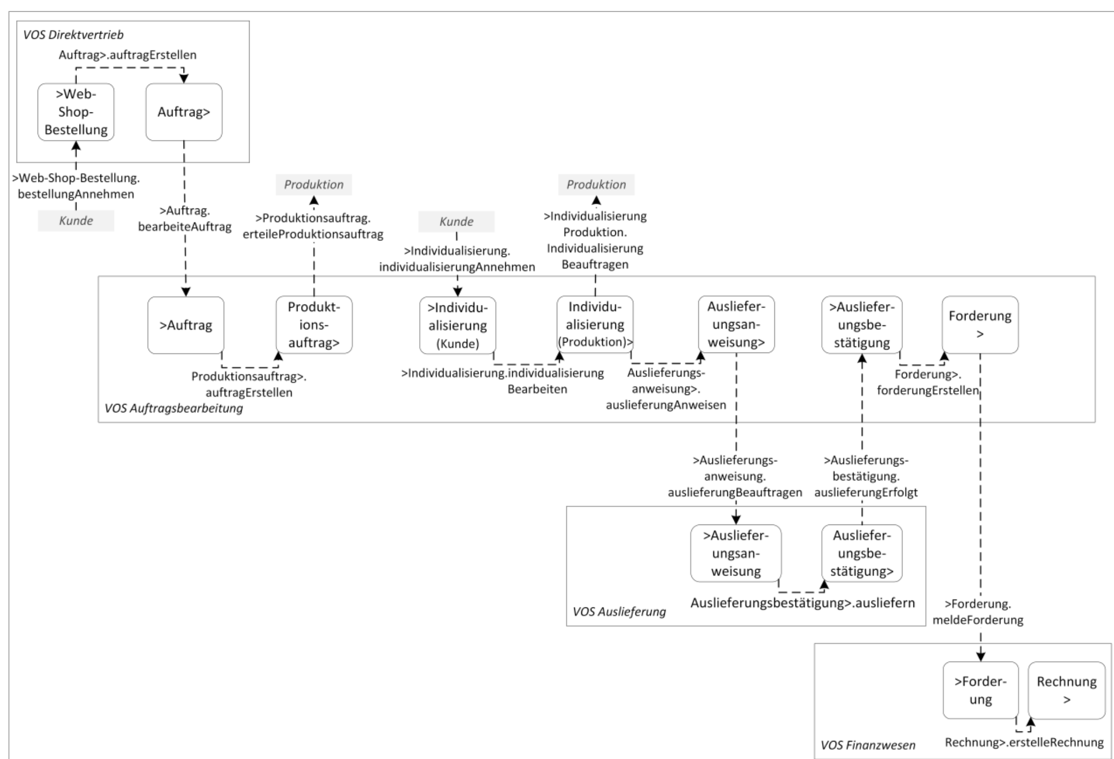

Abbildung 35: Resultierendes Vorgangsobjektschema

VOS, die mit den betrieblichen Objekten *Direktvertrieb*, *Auftragsbearbeitung*, *Auslieferung* und *Finanzwesen* korrespondieren, sind in Abbildung 35 im Zusammenhang dargestellt.

Aus dem VOS *Direktvertrieb* werden im Unterschied zum initialen VOS die VOT *>Vertriebsauftrag* und *Werbung>* entfernt, da die entsprechenden Aufgaben nicht automatisiert sind. Dem VOS *Auftragsbearbeitung* fehlen aus diesem Grund die VOT *Mandatsauftrag>*, *>Mandatsleistung* und *Vertriebsauftrag>*. Des Weiteren sind im VOS *Auslieferung* die VOT *>e-Car-Bereitstellung* und *Auslieferung>* weggefallen. Dem VOS *Finanzwesen* fehlt schließlich der VOT *>Kd.Zahlung*.

Bei der Überarbeitung der VOS werden Umbenennungen vorgenommen (siehe Tabelle 3), um die repräsentierten Objekte aus Sicht des AwS konziser zu beschrieben. Auf eine Zusammenfassung mehrerer VOT wird verzichtet.

VOS	VOT des initialen VOS	VOT des überarbeiteten VOS
Direktvertrieb	>Bestellung im Web-Shop	>Web-Shop-Bestellung
Auftrags-bearbeitung	Bestellungen>	Produktionsauftrag>
	>Nachträgl. Individualisierung (Kunde)	>Individualisierung (Kunde)
	Nachträgl. Individualisierung (Produktion)>	Individualisierung (Produktion)>
	Forderungsmeldung>	Forderung>
Finanzwesen	>Forderungsmeldung	>Forderung

Tabelle 3: Umbenennung von Vorgangsobjekttypen

Die Zuweisung von Attributen und Operatoren wird zusammen mit der entwickelten textuellen Notation des VOS ausschnittsweise gezeigt. Das auf diese Weise dargestellte Modell dient als Eingabe für die Transformation mittels EMF.

Abbildung 36 zeigt die textuelle Notation, wobei nur der Inhalt des VOS *Direktvertrieb* und ein Teil des VOS *Auftragsbearbeitung* zu sehen ist. Weitere VOS sind aus Platzgründen verdeckt[17].

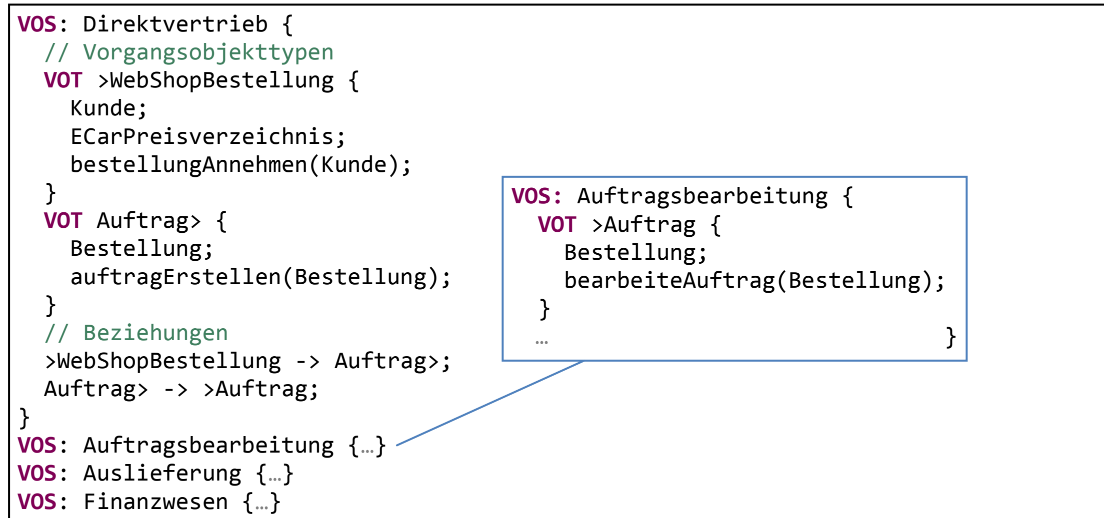

Abbildung 36: Vorgangsobjektschema zum VOS Direktvertrieb in textueller Notation

Die textuelle Notation definiert einzelne benannte VOS. Innerhalb eines VOS lassen sich mit dem Schlüsselwort VOT entsprechende Objekttypen definieren sowie *interacts_with*-Beziehungen. Die hier definierten Beziehungen besitzen die Kardinalität (0,*), die durch einen Pfeil ausgedrückt wird. Die Beziehung Auftrag> -> >Auftrag verläuft von dem VOS *Direktvertrieb* zu dem nebenstehenden VOS

[17] Das vollständige Modell ist in textueller Notation auf der DVD enthalten.

Auftragsbearbeitung, welches hier nur ausschnittsweise dargestellt ist. Jeder VOT enthält außerdem Attribute, die gemeinsam einen Teilgraphen des KOS definieren (Ferstl und Sinz 2013, S. 230f.). Ein Attribut gibt einen KOT an, der zusammen mit den KOT der anderen Attribute die Knotenmenge des Teilgraphen bildet. Die Kanten ergeben sich aus den im KOS definierten Beziehungen. Des Weiteren sind Operatoren definiert, bei denen jeweils in Klammern Eingabeparameter angegeben werden können. Parameter definieren Nachrichten, die von einem VOT empfangen werden können. Die Operatoren entsprechen denen, die bereits in Abbildung 35 definiert sind. Die später folgende Implementierung der Operatoren definiert die Navigation auf dem durch die Attribute festgelegten KOS-Teilgraphen.

4.3.2 Entwicklung des Servicemodells

Nachdem der fachliche Entwurf in Form von KOS und VOS spezifiziert ist, kann nun das darauf aufsetzende Servicemodell entwickelt werden. In Anlehnung an das mehrschrittige Vorgehen von Teusch und Sinz (2012, S. 5) folgt die Betrachtung externer Prozess- und Datenschnittstellen sowie die Identifizierung von Services.

Für jedes der abgegrenzten AwS werden Prozessschnittstellen vorgesehen. Die Bestimmung einzelner Schnittstellen ergibt sich aus den ein- und ausgehenden Nachrichtenflüssen einzelner VOT (Teusch und Sinz 2012, S. 8). Die Schnittstellen werden analog zu den bereits zugeordneten Operatoren mit Parametern definiert, die gemeinsam eine Nachricht für eine Schnittstelle definieren.

Beispielsweise besitzt das VOS *Auftragsbearbeitung* den Operator *bearbeiteAuftrag* mit dem Parameter *Bestellung*[18]. Die daraus abgeleitete Schnittstelle erhält den gleichen Namen wie der Operator und nimmt Nachrichten des Typs *Bestellung* entgegen. In der textuellen Notation des Servicemodells wird die Schnittstelle mit dem Schlüsselwort *Interface* definiert (Abbildung 37).

```
Interface {  bearbeiteAuftrag(Bestellung);  }
```

Abbildung 37: Service-Schnittstelle in textueller Notation

Die Definition von Datenschnittstellen folgt der gleichen Syntax. Die Identifizierung erfolgt dabei durch Betrachtung der Attribute einzelner KOT. So ist eine Schnittstelle erforderlich, wenn Attribute eines KOT auch von den umliegenden KOT verwaltet werden (Teusch und Sinz 2012, S. 8). Dies würde eine Datenredundanz bedeuten, die

[18] Siehe: blau umrahmtes VOS in Abbildung 36

dazu führt, dass eine Änderung des redundanten Attributwertes über die definierte Schnittstelle mit anderen Services abgeglichen werden muss. Im vorliegenden Fall sind keine redundanten Attribute vorhanden.

Die Definition weiterer Schnittstellen erfolgt mit der Identifikation von Services. Hierzu wird das SOA-Architekturmodell (Teusch und Sinz 2011, S. 294-296) zusammen mit dem entwickelten Metamodell zur Spezifikation von Services genutzt.

Identifikation von Entitäts-Services

Die aus mehreren KOT gebildeten Entitäts-Services sind zusammen mit dem überarbeiteten KOS in Abbildung 38 dargestellt. Die KOT *Kunde* und *Lieferung* sind jeweils durch eigene Entitäts-Services gekapselt. Die Services *AuftragES* und *FinanzwesenES* enthalten jeweils mehrere KOT, da die zugrundeliegenden betrieblichen Objekte beider Services inhaltlich zusammenhängen und eine feingranulare Aufteilung, wie etwa bei *KundeES*, somit nicht sinnvoll wäre.

In der textuellen Beschreibungssprache für serviceorientierte AwS werden die Entitäts-Services gemäß der entwickelten Grammatik[19] notiert. Die identifizierten Services sind in Abbildung 39 dargestellt. Schnittstellen sind nur für den Entitäts-Service *KundeES* aufgedeckt und werden für andere Services nicht gezeigt[20].

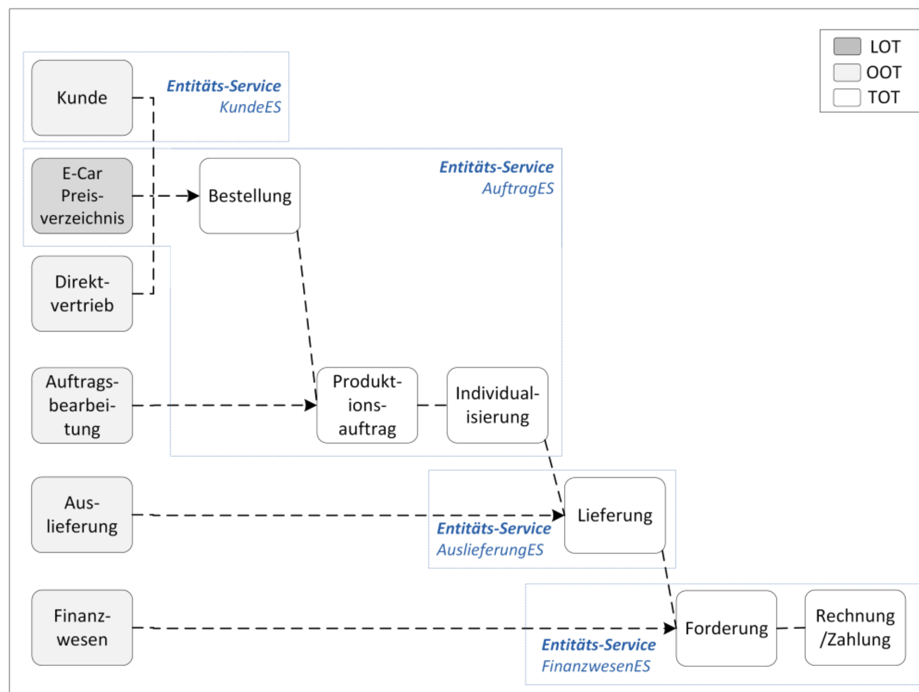

Abbildung 38: Identifikation von Entitäts-Services im konzeptuellen Objektschema

[19] Siehe Abschnitt 3.3.1

[20] Das vollständige Modell ist in textueller Notation auf der DVD enthalten.

```
EntitaetsService KundeES {
  Interface {…}
  // konzeptuelle Objekttypen:
  KOT Kunde;
}
EntitaetsService AuftragES {
  Interface {…}
  // konzeptuelle Objekttypen:
  KOT Bestellung;
  KOT Produktionsauftrag;
  KOT Individualisierung;
  KOT ECarPreisverzeichnis;
}
EntitaetsService AuslieferungES {
  Interface {…}
  // konzeptuelle Objekttypen:
  KOT Lieferung;
}
EntitaetsService FinanzwesenES {
  Interface {…}
  // konzeptuelle Objekttypen:
  KOT Forderung;
  KOT RechnungZahlung;
}
```

```
Interface {
  Kunde createKunde();
  Kunde readKunde(ID);
  updateKunde(Kunde);
  deleteKunde(Kunde);
}
```

Abbildung 39: Definition von Entitäts-Services in textueller Notation

Ein Entitäts-Service wird anhand eines Namens, eines Interface und den zu verwaltenden KOT definiert. Einzelne Entitäts-Services definieren Schnittstellen zum Zugriff durch elementare Vorgangs-Services oder für den Abgleich von Daten zwischen Entitäts-Services. In der Fallstudie ergeben sich keine Überschneidungen bezüglich der verwalteten Daten, so dass die Schnittstellen nur im Rahmen der Orchestrierung durch elementare Vorgangs-Services genutzt werden.

Beispielsweise definiert der Entitäts-Service *KundeES* Schnittstellen, die *Create*-*Read*- und *Update*-Operationen (CRUD) für den KOT *Kunde* anbieten.

Identifikation von Vorgangs-Services

Elementare Vorgangs-Services werden aus VOT einzelner betrieblicher Aufgaben abgeleitet. Die Orchestrierung dieser Services wird von nicht-elementaren Vorgangs-Services durchgeführt, die somit ein Vorgangsnetz bilden (Teusch und Sinz 2011, S. 299). Abbildung 40 zeigt die identifizierten Vorgangs-Services zusammen mit dem überarbeiteten Vorgangsobjektschema. Der nicht-elementare Vorgangs-Service *DirektvertriebVS* bildet ein Vorgangsnetz, das lediglich aus den elementaren Vorgangs-Services der VOT >*Web-Shop-Bestellung* und *Auftrag*> besteht. Aus dem VOS *Auftragsverwaltung* ergeben sich die beiden nicht-elementaren Vorgangs-Services *BestellungVS* sowie *AuftragsabwicklungVS*. Die beiden Vorgangsnetze sind bezüglich der Schnittstellen nicht voneinander abhängig, so dass sich die Bildung

von zwei Services zur Schaffung möglichst feingranularer Zugriffsmöglichkeiten anbietet.

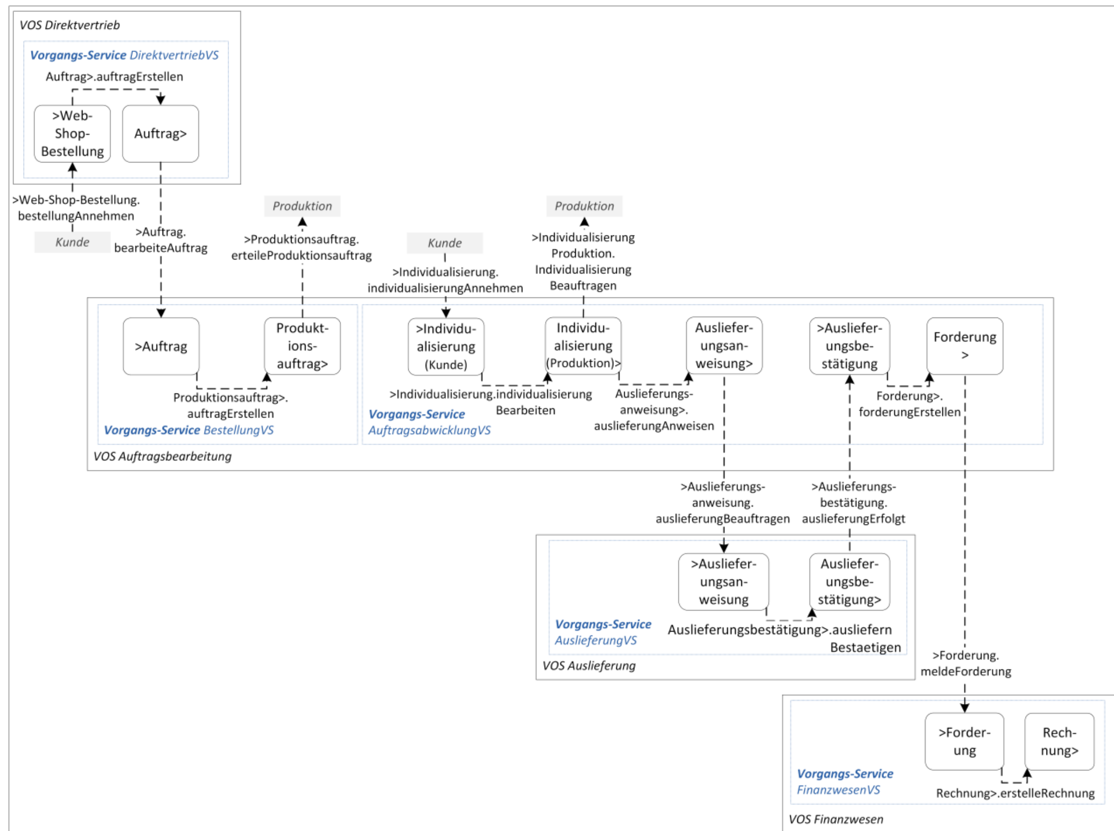

Abbildung 40: Identifikation von Vorgangs-Services im Vorgangsobjektschema

Die Identifikation von elementaren Vorgangs-Services basiert auf den zur Aufgabe gehörenden VOT, wobei entweder gemäß der Message Exchange Pattern (W3C 2007) *In-Only* oder *Out-Only* ein einzelner VOT beteiligt ist, oder zwei VOT, sofern das Pattern *In-Out* zum Einsatz kommt. Die elementaren Vorgangs-Services werden nun anhand der textuellen Beschreibung spezifiziert.

Abbildung 41 bildet die sämtliche Vorgangs-Services ab, wobei an dieser Stelle nur der Inhalt der externen Schnittstellen aufgedeckt ist. Ein Vorgangs-Service enthält neben Schnittstellen noch weitere elementare Vorgangs-Services und einen Workflow. Die Namen der elementaren Vorgangs-Services tragen das Suffix *VSe* und korrespondieren bis auf eine Ausnahme mit den Namen der entsprechenden VOT. So wurden die VOT zur Auslieferung (*Auslieferungsanweisung*> und >*Auslieferungsbestätigung*) im VOS *Auftragsabwicklung* gemäß dem *In-Out*-Pattern in einen elementaren Vorgangs-Service mit dem Namen *AuslieferungVSe* überführt.

Abbildung 41: Definition von Vorgangs-Services in textueller Notation

Der Vorgangs-Service *BestellungVS* ist in Abbildung 42 vollständig dargestellt[21]. Die Schnittstelle *bearbeiteAuftrag* des elementaren Vorgangs-Service *AuftragVSe* wird als externe Schnittstelle in das Interface des nicht-elementaren Vorgangs-Service exportiert, da sie dort ebenfalls definiert ist. Die Schnittstelle des Service *ProduktionsauftragVSe* ist dagegen nur intern verfügbar. Intern verfügbare Schnittstellen werden für die Orchestrierung derjenigen elementaren Vorgangs-Services benötigt, die diese anbieten. Elementare Vorgangs-Services orchestrieren selbst Entitäts-Services, die jeweils hinter dem Schlüsselwort *EntitaetsService* angegeben werden können. In diesem Fall wird der Service *AuftragES* zur Verwaltung der entsprechenden Entität benötigt.

Der angegebene Workflow spezifiziert Beziehungen, die die Orchestrierung elementarer Vorgangs-Services und einen Teil der Choreografie betreffen. In diesem Fall ist eine Beziehung zur Orchestrierung definiert, die nach der Ausführung von *bearbeiteAuftrag* des Service *BestellungVS* die Ausführung von *auftragErstellen* des

[21] Das vollständige Modell ist in textueller Notation auf der DVD enthalten.

Service *ProduktionsauftragVS* startet. Eine zweite Beziehung des gezeigten Workflows startet nach der Ausführung von *auftragErstellen* das Versenden einer Nachricht an den nicht-elementaren Service *ProduktionVS.* Der Versand der Nachricht von *BestellungVS* nach *ProduktionVS* definiert damit einen Teil der Choreografie zwischen nicht-elementaren Vorgangs-Services.

```
VorgangsService BestellungVS {
  Interface { bearbeiteAuftrag(Bestellung); }
  VorgangsService AuftragVSe {
    Interface { bearbeiteAuftrag(Bestellung);  }
    EntitaetsService AuftragES;
    VOT >Auftrag;
  }
  VorgangsService ProduktionsauftragVSe {
    Interface { auftragErstellen(Bestellung);  }
    EntitaetsService AuftragES;
    VOT Produktionsauftrag>;
  }
  Workflow {
    bearbeiteAuftrag -> auftragErstellen;                       // Orchestrierung
    auftragErstellen >> ProduktionVS.erteileProduktionsauftrag; // Choreografie
  }
}
```

Abbildung 42: Vorgangs-Service BestellungVS in textueller Notation

4.4 Ableitung des Softwaremodells: Design

Das im vorherigen Abschnitt gezeigte textuelle Modell definiert die fachliche Spezifikation des AwS und wird nun mit Hilfe der in Kapitel 3.4.1 definierten Zuordnungsfunktionen in ein Modell des Softwaredesign transformiert. Die Transformation ist mit QVT in der OML definiert[22] und kann beispielsweise mit Hilfe der Eclipse-Implementierung QVTo durchgeführt werden. QVTo ist unter anderem in den Eclipse Modeling Tools[23] enthalten, die hier exemplarisch für die Durchführung der Transformation genutzt werden.

Durchführung der Transformation

Ausgangspunkt ist das textuelle Modell des vorherigen Abschnitts. Zur Verarbeitung des Modells wird ein von Xtext und ANTLR generierter Parser benötigt[24], der das textuelle Modell einliest und dem EMF als Ecore-Modell zugänglich macht. Die benötigten Komponenten sind in einer vorkonfigurierten Eclipse-Version auf DVD enthalten.

[22] Siehe Abschnitt 2.4.1
[23] Siehe DVD oder https://www.eclipse.org/downloads/packages/eclipse-modeling-tools/keplersr1
[24] Siehe Abschnitt 2.3.2

Zunächst werden Quell- und Zielmodell in der *Run Configuration* des QVT-Eclipse-Projekts festgelegt.

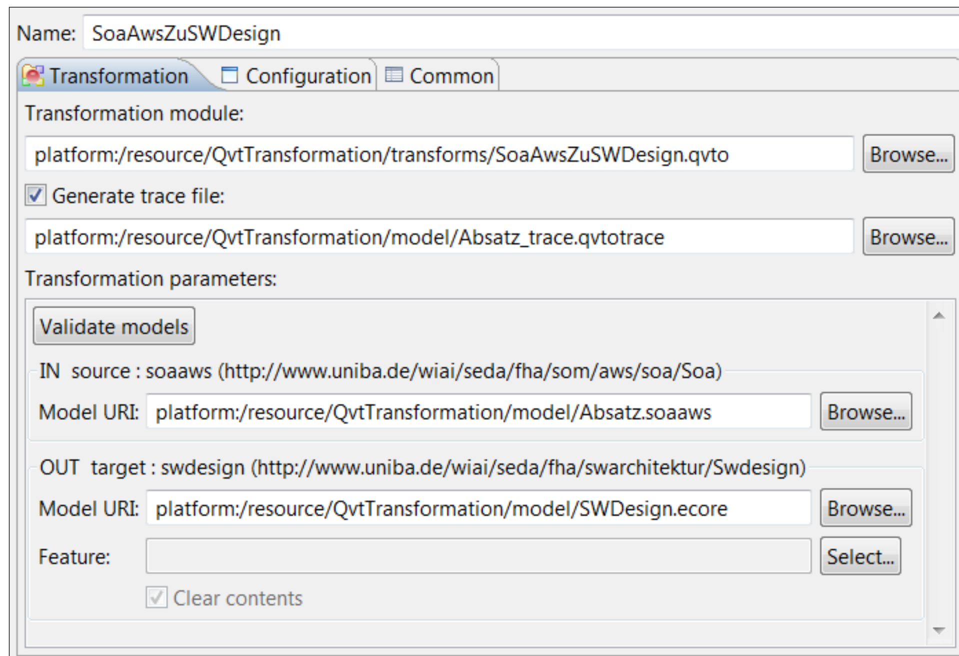

Abbildung 43: Konfiguration der QVT-Transformation

Abbildung 43 zeigt eine entsprechende Konfiguration, bei der außerdem die Generierung eines *Trace File* aktiviert ist. Ein *Trace* bietet die Möglichkeit, einzelne Modelelemente des Zielmodells nach der Transaktion auf Modellelemente des Quellmodells zurückführen. Der Trace stellt somit eine Verknüpfung zwischen den Modellen her (OMG 2011, S. 5). Der Dialog bietet außerdem die Möglichkeit, geladene Modelle syntaktisch zu validieren.

Wird die Transformation für das Modell des *Absatz* gestartet, ergeben sich die in Abbildung 44 gezeigten Statusmeldungen zur Transformation von Entitäts-Services. In der Abbildung werden die Services *KundeES* sowie *AuftragES* transformiert. Dabei wird zunächst ein EJB-Module für einen Service erstellt, anschließend werden das Interface, enthaltene Entitäten und der Service selbst transformiert. Die Transformation des Service folgt an letzter Stelle, da dieser von vorher erstellten Elementen abhängig ist. So beinhaltet die Stateless Session Bean etwa Referenzen auf einzelne JPA-Entities und ist außerdem abhängig vom Business Interface, dessen Methoden in der Session Bean implementiert werden.

```
Transformation von Entitaets-Services
  Entitaets-Service: KundeES
    Entitaets-Service -> EJB-Module: KundeES
    Interface -> Business Interface: KundeES
    Entität -> JPA-Entity: Kunde
    Entitäts-Service -> Stateless Session Bean: KundeES
…
```

Abbildung 44: Statusmeldungen zur Transformation von Entitäts-Services

Eine beispielhafte Transformation des Vorgangs-Services *DirektvertriebVS* ist in Abbildung 45 dargestellt. Zunächst wird auch hier ein EJB-Modul und ein Interface erzeugt. Anschließend werden die darin enthaltenen elementaren Vorgangs-Services transformiert, hier *WebShopBestellungVSe* und *AuftragVSe.* Für beide wird neben einem Interface eine Stateless Session Bean angelegt. Erst danach erfolgt die Transformation des nicht-elementaren Vorgangs-Services zu einer Message Driven Bean und einer Stateful Session Bean. Die Session Bean ist auch hier von vorher transformierten Elementen abhängig, wie etwa den elementaren Vorgangs-Services, da entsprechende Referenzen auf diese erstellt werden.

```
Transformation von Vorgangs-Services
  Vorgangs-Service: DirektvertriebVS
    Vorgangs-Service -> EJB-Module: DirektvertriebVS
    Interface -> Business Interface: DirektvertriebVS
    Transformation elementarer Vorgangs-Services
      Vorgangs-Service: WebShopBestellungVSe
        Interface -> Business Interface: WebShopBestellungVSe
        Vorgangs-Service -> Stateless Session Bean: WebShopBestellungVSe
      Vorgangs-Service: AuftragVSe
        Interface -> Business Interface: AuftragVSe
        Vorgangs-Service -> Stateless Session Bean: AuftragVSe
    Vorgangs-Service -> Message Driven Bean: DirektvertriebVS
    Vorgangs-Service -> Stateful Session Bean: DirektvertriebVS
…
```

Abbildung 45: Statusmeldungen zur Transformation von Vorgangs-Services

Resultierende Modelle

Nach der Transformation ergibt sich das in Abbildung 46 als Baumstruktur gezeigte Ecore-Modell. Dargestellt ist das Modul zum Entitäts-Service *KundeES*, welches neben der Stateless Session Bean zur Realisierung des Service noch ein entsprechendes Business Interface und eine JPA Entity mitbringt.

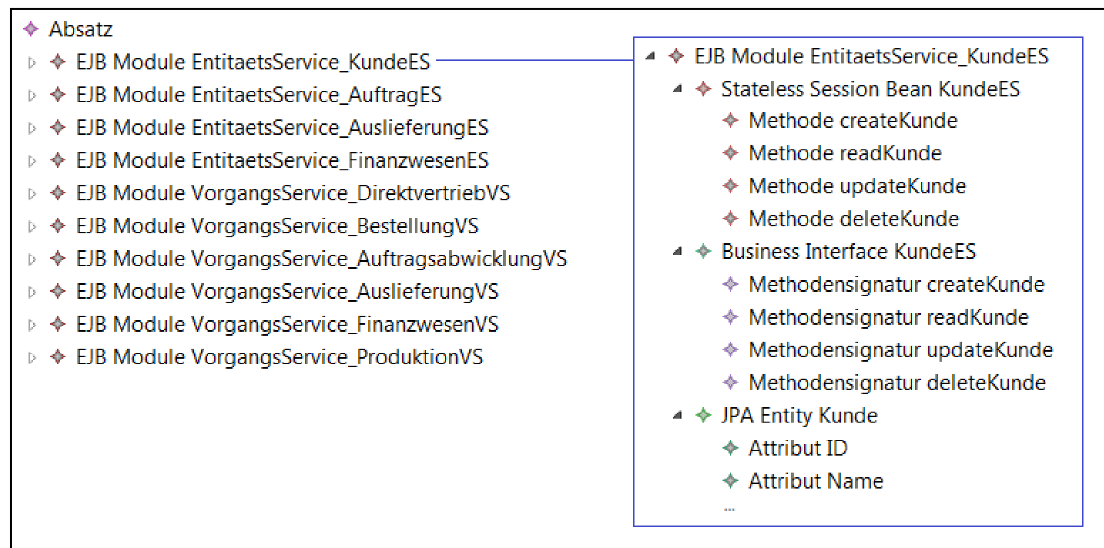

Abbildung 46: Ecore-Modell des Softwaredesigns (1)

Methoden und Methodensignaturen in Stateful Session Bean und Business Interface entsprechen den Definitionen der Service-Schnittstellen aus Abbildung 39. Die Attribute der JPA Entity ergeben sich aus den Attributen des KOT *Kunde*, die in Abbildung 34 dargestellt sind. Die JPA Entity enthält außerdem noch Operatoren des KOT, die hier aus Platzgründen nicht dargestellt sind.

Abbildung 47 zeigt den Ausschnitt für den Vorgangs-Service *DirektvertriebVS*. Die Stateful Session Bean realisiert den Service und besitzt Attribute, die nicht-elementare Vorgangs-Services in Form von Stateless Session Beans referenzieren, ebenso wie eine Message Driven Bean. Die Service-Methoden *bestellungAnnehmen* und *auftragErstellen* orchestrieren die elementaren Vorgangs-Services *WebShopBestellungVSe* bzw. *AuftragVSe.* Das Business Interface des *DirektvertriebVS* definiert die externe Schnittstelle zur Methode *bestellungAnnehmen,* die anderen beiden Business Interfaces der elementaren Vorgangs-Services spezifizieren interne Schnittstellen für die gleichnamigen Stateless Session Beans. Eine solche Schnittstelle implementiert beispielsweise der Service *AuftragVSe*, der zusätzlich zu der entsprechenden Methode noch ein Attribut enthält, das die Entität *Bestellung* referenziert. Attribut und Methode wurden aus dem VOT *Auftrag*> übernommen (siehe Abbildung 36).

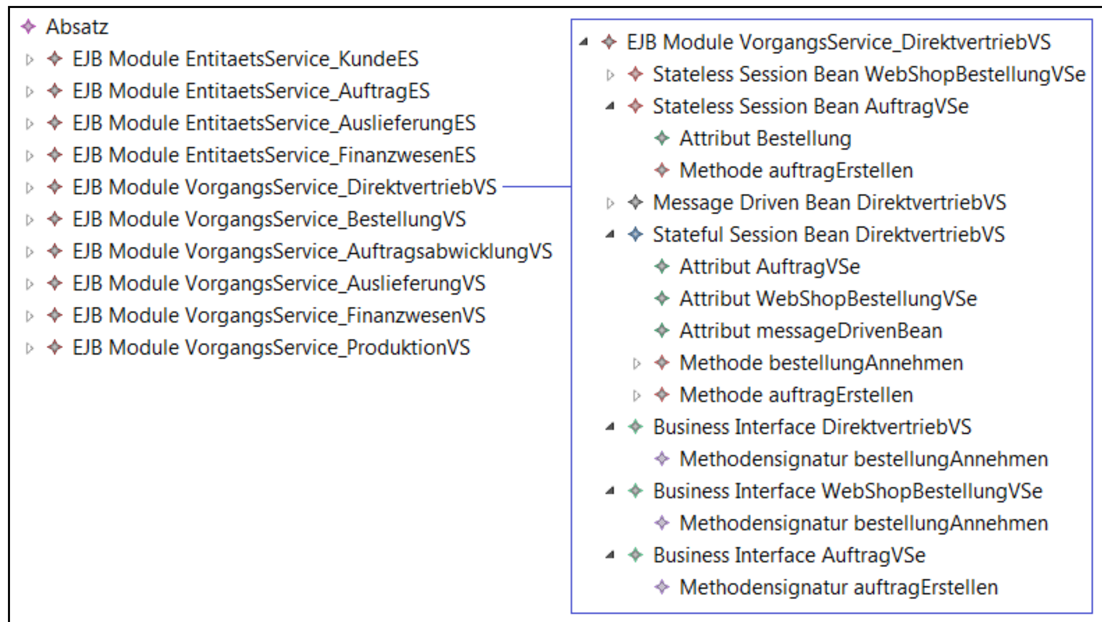

Abbildung 47: Ecore-Modell des Softwaredesigns (2)

Überarbeitung des Modells

Die gezeigte Transformation geht mit dem Übergang von der fachlichen Ebene zur softwaretechnischen Ebene einher. Das transformierte Modell bezieht sich somit teilweise auf fachliche Konzepte, die nun auf Konzepte der softwaretechnischen Ebene anzupassen sind. Dies erfolgt unter Beachtung der Spezifikation der Zielplattform *Java EE*.

Notwendig ist die Spezifikation von Datentypen für Attribute, Methoden und Methodensignaturen. Die Zuweisung erfolgt mit Hilfe des *Tree View Editors* von Eclipse und wird in Abbildung 48 exemplarisch für das Attribut *Name* gezeigt. Das Attribut besitzt in der Abbildung noch den fachlichen Typ *Name*, der sich aus der Bezeichnung des Attributs ergibt. Er wird nun in einen Java-Datentyp geändert; in diesem Fall *String*.

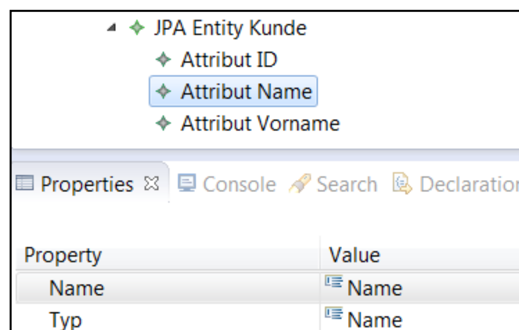

Abbildung 48: Zuweisung eines Datentyps für das Attribut Name

Neben der Zuweisung von Attributen können noch weitere optionale Überarbeitungsschritte erfolgen, wie etwa:

- das Hinzufügen neuer Methoden und Attribute zu Session Beans,
- das Hinzufügen zusätzlicher Module, EJBs oder JPA Entities und
- das Löschen und ggf. Ersetzen von Modulen, EJBs, JPA-Entities, Methoden oder Attributen.

Ein Hinzufügen von Modellelementen ist nötig, sofern entsprechende Elemente nicht aus fachlicher Sicht erfasst wurden, sondern nur aus softwaretechnischen Gründen benötigt werden. Beispielsweise können Hilfsmethoden zu Session Beans hinzugefügt werden, die häufig benötigte Funktionalität kapseln.

Die Entfernung eines Modellelements ist erforderlich, wenn ein fachliches Element softwaretechnisch nicht umgesetzt werden kann, z.B. weil die Zielplattform entsprechende Restriktionen aufweist. In diesem Fall kann das Element ggf. ersetzt werden.

Grundsätzlich sollten Überarbeitungen des Modells auf dieser Ebene jedoch minimiert werden, da eine erneute Transformation das Nachziehen sämtlicher überarbeiteter Modellelemente erfordert. Bei einer Überarbeitung ist zunächst zu analysieren, ob tatsächlich ein rein softwaretechnischer Grund vorliegt. Sofern die angestrebte Überarbeitung auch durch eine Änderung des fachlichen Modells mit anschließender Transformation erreicht werden kann, ist dies vorzuziehen.

4.5 Generierung von Softwareartefakten

Zur Generierung von Quellcode wird ein hierfür entwickeltes Werkzeug verwendet, welches das im vorherigen Abschnitt abgeleitete Modell mit Hilfe von Templates in Java-EE-Quellcode transformiert. Das Werkzeug liest ein Modell in Form einer Ecore-Datei ein, das mit Hilfe des EMF in Java-Objekte überführt wird, siehe Abbildung 49[25].

[25] Siehe Abschnitt 3.4.2

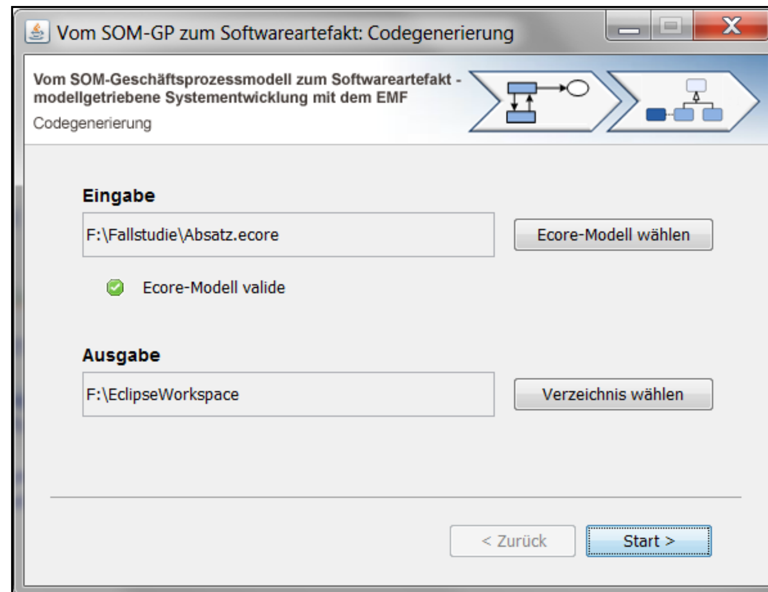

Abbildung 49: Werkzeug zur Codegenerierung

Das Werkzeug prüft nach Festlegung der Eingabe zunächst, ob das zu ladende Ecore-Modell syntaktisch valide ist. Da für alle generierten Module Eclipse-Projekte mit entsprechenden Konfigurationsdateien erzeugt werden, bietet sich als Ausgabeort das Arbeitsverzeichnis einer Eclipse-Entwicklungsumgebung für Java EE an.

```
Generiere Konfiguration für EJB-Module ...
  …
Generiere JPA-Entities ...
  EntitaetsService_KundeES
    Kunde
  …
Generiere EJBs ...
  EntitaetsService_KundeES
    Stateless Session Bean: KundeES
  VorgangsService_DirektvertriebVS
    Stateless Session Bean: WebShopBestellungVSe
    Stateless Session Bean: AuftragVSe
    Message Drive Bean: DirektvertriebVS
    Stateful Session Bean: DirektvertriebVS
Generiere Business Interfaces ...
  EntitaetsService_KundeES
  …
  VorgangsService_DirektvertriebVS
  …
```

Abbildung 50: Statusausgabe bei der Generierung von Quellcode

Die Transformation des Modells der Fallstudie ergibt die in Abbildung 50 dargestellte Ausgabe. Beispielsweise werden für das Modul *EntitaetsService_KundeES* eine JPA-Entity *Kunde*, eine Stateless Session Bean *KundeES* sowie ein Business Interface erstellt. Die Generierung des Quellcodes erfolgt zunächst im Speicher; der Code wird im Anschluss in eine Datei geschrieben.

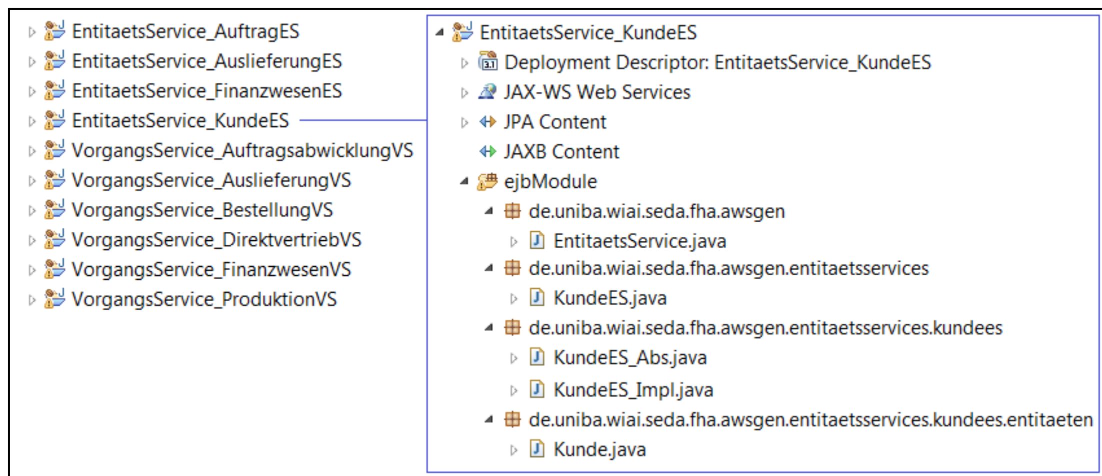

Abbildung 51: Generierte Eclipse-Projekte

Abbildung 51 listet links die einzelnen EJB-Module auf, für die jeweils ein Eclipse-Projekt angelegt ist. Das beispielhaft aufgeklappte Projekt *EntitaetsService_KundeES* beinhaltet neben Konfigurationsdateien für das Deployment (*Deployment Descriptor*) und die Konfiguration des DBVS für die JPA (*JPA Content*) verschiedene Java-Dateien mit Quellcode. Ein UML-Klassendiagramm ist in Abbildung 52 dargestellt, wobei einige Methoden der Klasse *Kunde* aus Platzgründen abgeschnitten sind. Der Service ist mit drei Klassen implementiert, die eine dreistufige Vererbungshierarchie bilden. *EntitaetsService* bietet oft benötigte Funktionen für alle Entitäts-Services an und wird in jedes Projekt kopiert. *KundeES_Abs* enthält generierten Quellcode, der durch einen Entwickler ggf. in der Klasse *KundeES_Impl* überschrieben oder modifiziert werden kann.

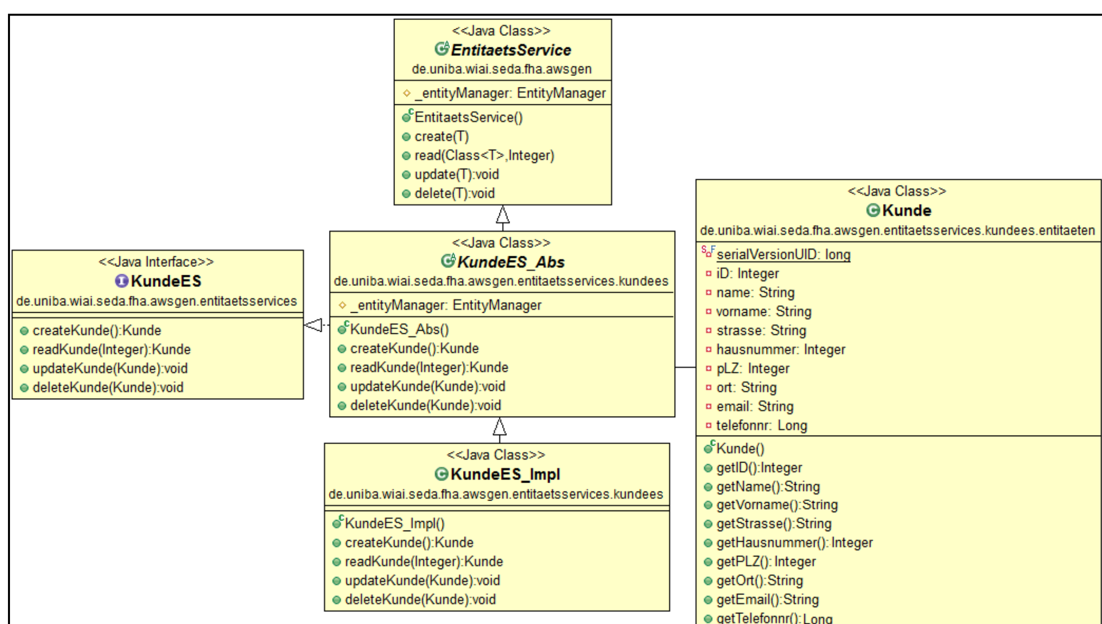

Abbildung 52: UML-Klassendiagramm eines Entitäts-Service

Die letztgenannte Klasse stellt damit einen definierten Punkt zur Erweiterung dar. Da sie durch den Entwickler auszufüllen ist, enthalten die Methoden keinerlei Funktionalität, sondern rufen lediglich die Methoden der Superklasse auf.

```
package de.uniba.wiai.seda.fha.awsgen.entitaetsservices.kundees;

import de.uniba.wiai.seda.fha.awsgen.entitaetsservices.kundees.entitaeten.*;

 * Abstrakte Klasse einer Stateless Session Bean.
public abstract class KundeES_Abs extends EntitaetsService implements KundeES {

     * @generated
    public Kunde createKunde() {
        Kunde _returnValue = null;
        _returnValue = super.create(new Kunde());
        return _returnValue;
    }

     * @generated
    public Kunde readKunde(Integer _iD) {
        Kunde _returnValue = null;
        _returnValue = super.read(Kunde.class, _iD);
        return _returnValue;
    }
```

Abbildung 53: Generierte Java-Klasse eines Entitäts-Service

Abbildung 53 zeigt die generierte Klasse *KundeES_Abs*. CRUD-Methoden werden anhand der Namenskonvention erkannt und vollständig generiert. Weitere Methoden müssen in der erbenden Klasse *KundeES_Impl* manuell hinzugefügt werden. Dies ist in Abbildung 54 beispielhaft anhand der Methode *getKunden* gezeigt, die eine SQL-Abfrage implementiert, mit der unter Nutzung eines objektrelationalen Mappers alle Kunden in der entsprechenden Tabelle als Liste zurückgegeben werden.

```
package de.uniba.wiai.seda.fha.awsgen.entitaetsservices.kundees;
import java.util.List;

 * Stateless Session Bean zur Realisierung eines Vorgangs-Service.
@Stateless
public class KundeES_Impl extends KundeES_Abs implements KundeES {

     * gibt alle Kunden zurück
    public List<Kunde> getKunden() {
        TypedQuery<Kunde> query;
        query = _entityManager.createQuery("SELECT k FROM Kunde k", Kunde.class);
        return query.getResultList();
    }

    @Override
    public Kunde createKunde() {
        return super.createKunde();
    }

    @Override
    public Kunde readKunde(Integer iD) {
        return super.readKunde(iD);
    }
```

Abbildung 54: Implementierungsklasse eines Entitäts-Service

Ein Beispiel einer generierten Entität stellt die Klasse *Kunde* (Abbildung 55) dar. Sie enthält das Attribut *ID*, das aufgrund der Annotation automatisch einen Primärschlüssel durch das DBVS erhält. Für weitere Attribute werden Getter- und Setter-Methoden selbsttätig generiert.

```
package de.uniba.wiai.seda.fha.awsgen.entitaetsservices.kundees.entitaeten;

import java.io.*;

 * Repraesentiert die Entitaet Kunde
@Entity
public class Kunde implements Serializable {

    private static final long serialVersionUID = 1L;

     * @generated
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer iD;

     * @generated
    private String name;

     * @generated
    public void setName(String name) {
        this.name = name;
    }

     * @generated
    public String getName() {
```

Abbildung 55: Generierte Klasse der Entität Kunde

Als Beispiel für einen Vorgangs-Service sind die generierten Artefakte des Moduls *VorgangsService_DirektvertriebVS* in Abbildung 56 aufgeführt.

Der Service *Direktvertrieb_VS* orchestriert die elementaren Vorgangs-Services *AuftragVSe* und *WebShopBestellungVSe.* Alle drei Services sind analog zu dem gerade gezeigten Entitäts-Service in einer dreistufigen Vererbungshierarchie mit jeweils drei Java-Klassen implementiert.

Weitere Pakete, deren Inhalt in der Abbildung verdeckt ist, enthalten die Schnittstellen anderer elementarer Vorgangs-Services, so dass insgesamt eine Choreografie realisiert werden kann. Außerdem sind die Schnittstellen von Entitäts-Services enthalten, anhand derer die elementaren Vorgangs-Services *AuftragVSe* und *WebShopBestellungVSe* orchestriert werden. Die für den Austausch von Nachrichten benötigten Entitäten sind ebenfalls enthalten.

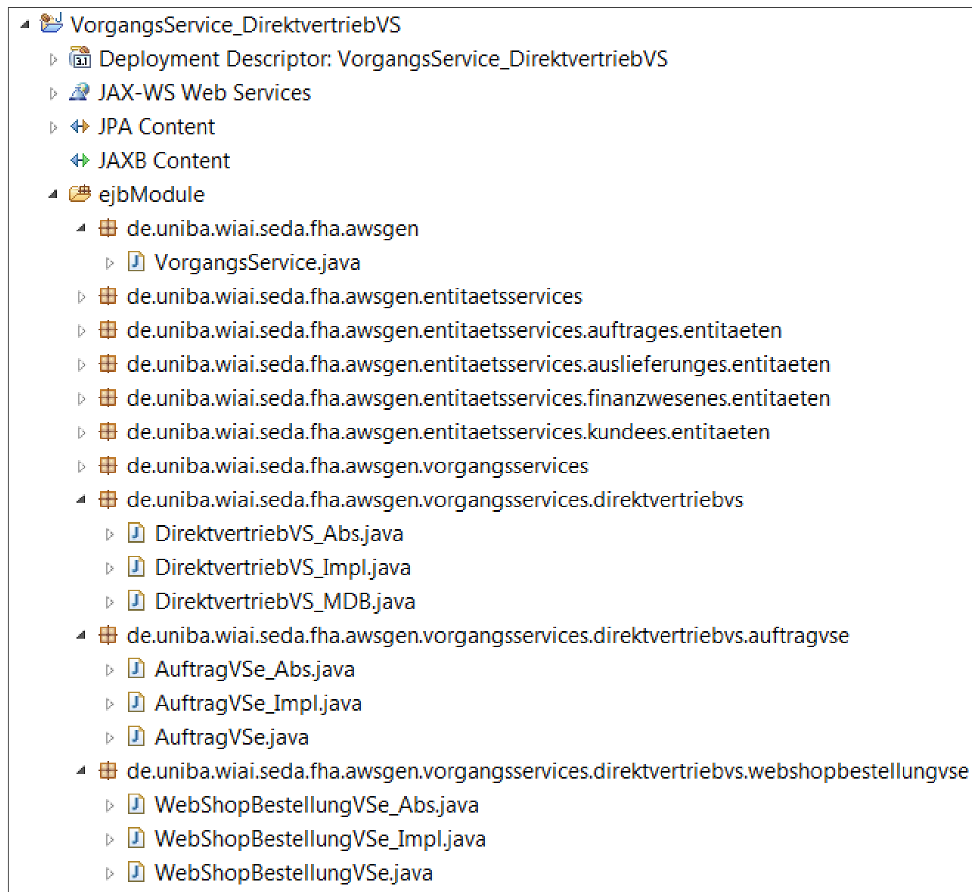

Abbildung 56: Generierte Softwareartefakte eines Vorgangs-Service

Das Klassendiagramm des Moduls *VorgangsService_DirektvertriebVS* ist in Abbildung 57 ausschnittsweise dargestellt. Es enthält nur den nicht-elementaren Vorgangs-Service *DirektvertriebVS;* die weiteren in Abbildung 56 gezeigten Klassen sind nicht enthalten. Die dreistufige Vererbungshierarchie zwischen den Klassen *VorgangsService, DirektvertriebVS_Abs* und *DirektvertriebVS_Impl* wird sichtbar.

Die Superklasse *VorgangsService* enthält vordefinierte Methoden und wird nicht generiert, sondern nur kopiert. *DirektvertriebVS_Abs* enthält die generierten Methoden zur Orchestrierung der elementaren Vorgangs-Services, die ggf. in *DirektvertriebVS_Impl* manuell überschrieben werden können. Die Message Driven Bean *DirektvertriebVS_MDB* realisiert die asynchrone Kommunikation im Rahmen der Choreografie.

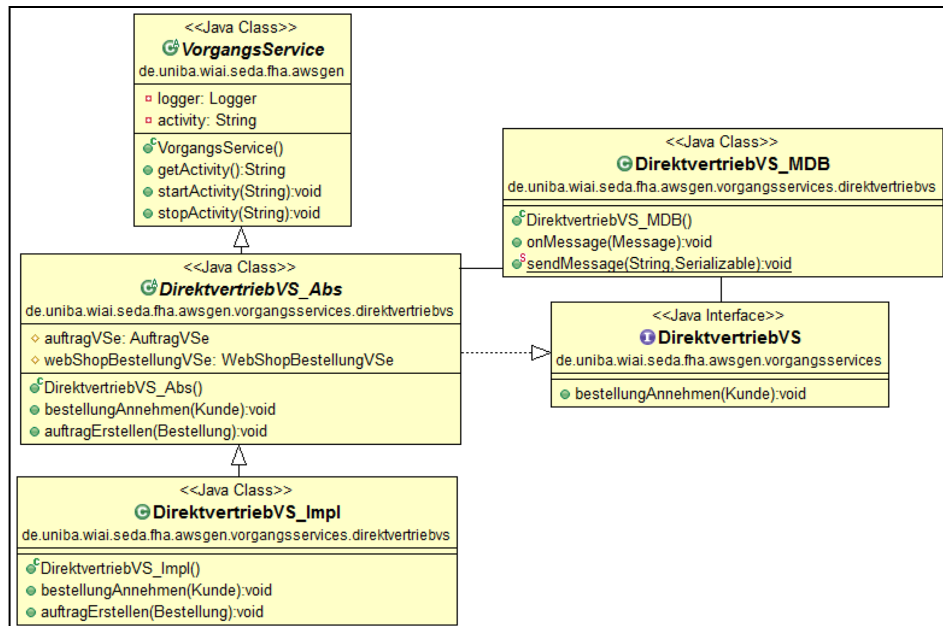

Abbildung 57: UML-Klassendiagramm eines Vorgangs-Service

Der generierte Vorgangs-Service wird in Abbildung 58 gezeigt. Der orchestrierte elementare Vorgangs-Service *webShopBestellungVSe* wird in der Methode *bestellungAnnehmen* aufgerufen. Die Methoden *startActivity* und *stopActivity* vermerken dies in einem Attribut der Superklasse und definieren den Zustand der Session Bean.

```
package de.uniba.wiai.seda.fha.awsgen.vorgangsservices.direktvertriebvs;

import javax.ejb.*;□

 * Abstrakte Superklasse einer Stateful Session Bean. □
public abstract class DirektvertriebVS_Abs extends VorgangsService implements DirektvertriebVS {

     * @generated□
    protected AuftragVSe auftragVSe;□

     * @generated□
    protected WebShopBestellungVSe webShopBestellungVSe;□

     * @generated□
    protected DirektvertriebVS_MDB messageDrivenBean;

     * @generated□
    public void bestellungAnnehmen(Kunde kunde) {
        startActivity("bestellungAnnehmen");
        webShopBestellungVSe.bestellungAnnehmen(kunde);
        stopActivity("bestellungAnnehmen");
        this.auftragErstellen(new Bestellung());
    }
```

Abbildung 58: Generierte Java-Klasse eines Vorgangs-Service

Die generierte Message Driven Bean, die Methoden des eben gezeigten Service bei eintreffenden Nachrichten von anderen nicht-elementaren Vorgangs-Services aufruft, ist in Abbildung 59 dargestellt. Sie setzt das Vorhandensein einer Message Queue *DirektvertriebVS_Queue* voraus.

```
package de.uniba.wiai.seda.fha.awsgen.vorgangsservices.direktvertriebvs;

import java.io.Serializable;

 * Message Driven Bean zur asynchronen Kommunikation 
@MessageDriven(activationConfig = { @ActivationConfigProperty(
    propertyName = "destinationType", propertyValue = "javax.jms.Queue")},
    mappedName = "DirektvertriebVS_Queue")
public class DirektvertriebVS_MDB implements MessageListener {

    @EJB
    private DirektvertriebVS _vorgangsservice;

    public void onMessage(Message queueMessage) {
        try {
            Object messageObject = ((ObjectMessage) queueMessage).getObject();
            if (messageObject instanceof Kunde) {
                Kunde message = (Kunde) messageObject;
                _vorgangsservice.bestellungAnnehmen(message);
            }
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
```

Abbildung 59: Message Driven Bean zur asynchronen Kommunikation

Die somit generierten Softwareartefakte dienen als Basis für Entwicklung des Systems. Sie stellen ein Rahmenwerk dar, das durch den Entwickler ausgefüllt und erweitert wird. Weitere Schritte hierfür sind die Definition nicht automatisch erstellter Datenbankabfragen in Entitäts-Services, die Erweiterung von Vorgangs-Services durch nicht generierte Lösungsverfahren sowie die Erstellung von Client-Anwendungen.

Als Beispiel für die manuelle Definition von Datenbankabfragen wird auf Abbildung 54 verwiesen, die den Entitäts-Service *KundeES* mit einer zusätzlichen SQL-Abfrage erweitert.

Die Erweiterung mit nicht generierten Lösungsverfahren wird am Beispiel des elementaren Vorgangs-Service *WebShopBestellungVSe* demonstriert. Die Klasse in Abbildung 60 stellt die Implementierungsklasse dar, bei der die generierte Methode *bestellungAnnehmen* durch einen Entwickler überschrieben wurde. In dieser Methode wird ein aus dem Web-Shop übergebenes Objekt des Typs *Kunde* mit Hilfe des Entitäts-Service *kundeES* in der Datenbank erstellt und nach dem Übernehmen einzelner Werte persistiert. Anschließend wird eine Bestellung erstellt, bei der die ID des Kunden als Fremdschlüssel sowie ein Zeitstempel gesetzt werden. Die Attribute *kundeES* und *auftragES*, mit denen die entsprechenden Entitäts-Services aufgerufen werden, sind in der Superklasse *WebShopBestellungVSe_Abs* definiert.

```java
package de.uniba.wiai.seda.fha.awsgen.vorgangsservices.direktvertriebvs.webshopbestellungvse;
import javax.ejb.*;

 * Stateless Session Bean für Web-Shop-Bestellungen
@Stateless
public class WebShopBestellungVSe_Impl extends WebShopBestellungVSe_Abs implements WebShopBestellungVSe {

    @Override
    public void bestellungAnnehmen(Kunde kunde) {
        Kunde k = kundeES.createKunde();
        k.setVorname(kunde.getVorname());
        k.setName(kunde.getName());
        k.setStrasse(kunde.getStrasse());
        k.setHausnummer(kunde.getHausnummer());
        k.setPlz(kunde.getPLZ());
        k.setOrt(kunde.getOrt());
        k.setTelefonnr(kunde.getTelefonnr());
        k.setEmail(kunde.getEmail());
        kundeES.updateKunde(k);

        Bestellung b = auftragES.createBestellung();
        b.setKundeID(k.getID());
        b.setZeitstempel(System.currentTimeMillis());
        auftragES.updateBestellung(b);
    }
}
```

Abbildung 60: Stateless Session Bean zur Annahme von Web-Shop-Bestellungen

Eine beispielhafte Client-Anwendung für den Vorgangs-Service *DirektvertriebVS* ist mittels Servlets und JSP implementiert (siehe Abbildung 62). Ein Kunde kann über den Client im Browser eine Bestellung aufgeben, die über das auf dem Web-Server ausgeführte Servlet an den Anwendungs-Server des genannten Vorgangs-Service weitergeleitet wird. Wie im letzten Codebeispiel gezeigt, werden Kunden- und Bestellungsdaten dort mit Hilfe von Entitäts-Services persistiert. In der Datenbank ergeben sich dadurch Einträge wie die in Abbildung 61. Im rechten Teil der Abbildung sind außerdem die durch den objektrelationalen Mapper (OR-Mapper) erstellten Tabellen zu sehen.

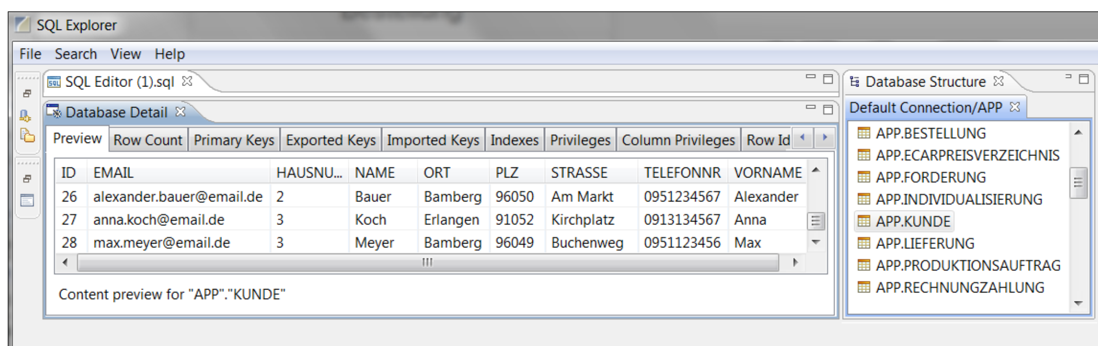

Abbildung 61: Datenbankeinträge nach dem Anlegen von Kunden

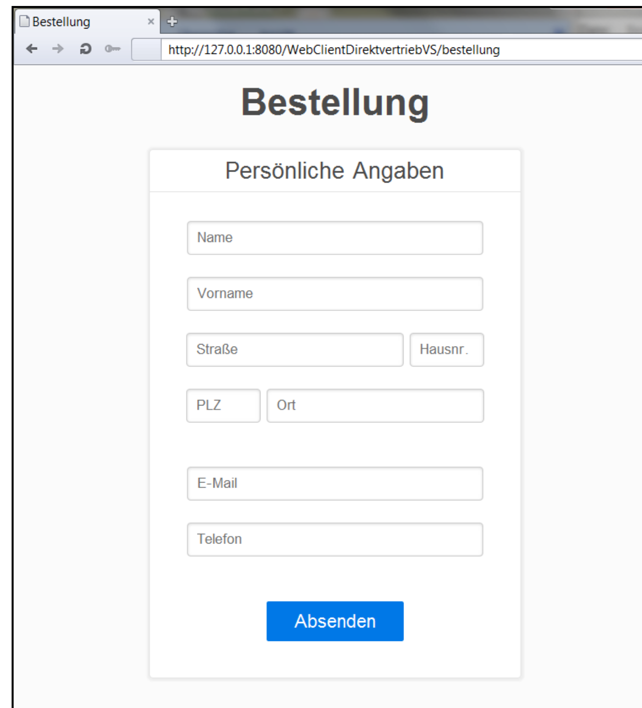

Abbildung 62: Web-Client zur Bestellung von e-Cars

Zur Inbetriebnahme ist noch die Konfiguration von Web-, Anwendungs- und Datenbankserver nötig. Das hier gezeigte System ist mit dem Web- und Anwendungs-Server Oracle GlassFish 4.0 getestet. Als DBVS kommt Apache Derby Network Server 10 zum Einsatz[26]. Die Message Queues werden in GlassFish angelegt, z.B. mit Hilfe des generierten Skripts *queueCreateSkript.bat*. Die Konfiguration der Datenbankschnittstelle und des OR-Mappers für JPA erfolgen in der ebenfalls generierten *persistence.xml* im Verzeichnis *META-INF* eines EJB-Moduls. Als OR-Mapper kommt beispielhaft *EclipseLink* zum Einsatz, für den die generierte *persistence.xml* nicht angepasst werden muss. Datenbanktabellen werden für alle Entitäten während des Deployment automatisch erstellt.

Erfolgt das Deployment auf dem Anwendungs-Server GlassFish mit Hilfe von Eclipse, so sind keine weiteren Konfigurationsschritte nötig. Ansonsten ist ggf. noch der Deployment Descriptor eines EJB-Moduls in der Datei *ejb-jar.xml* zu konfigurieren.

Für die Kommunikation zwischen Anwendungs-Server und DBVS wird ein *JDBC-Connection Pool* mit dem Ressourcentyp *javax.sql.DataSource* benötigt, der in der Weboberfläche der GlassFish-Konfiguration im Bereich *Ressourcen/JDBC* angelegt werden kann. Das Derby-DBVS selbst ist vorkonfiguriert auf DVD enthalten.

[26] Installationsdateien der Server-Software sind auf der DVD enthalten.

5 Zusammenfassung und Diskussion

Der vorgeschlagene Ansatz lässt sich im Rahmen der Systementwicklung dazu nutzen, die semantische Lücke zwischen fachlicher und softwaretechnischer Ebene zu überwinden. Hierfür wird eine durchgängig modellgetriebene Vorgehensweise verwendet, die vom SOM-Geschäftsprozessmodell über die fachliche Spezifikation bis zum Softwareartefakt konsequent angewendet wird. Die Ableitung von Modellen folgt dem Muster der metamodellbasierten Transformation der MDA. Das Vorgehen wird durch das Eclipse Modeling Framework unterstützt und mit zugehörigen Technologien implementiert.

Verwandte Ansätze, die eine modellbasierte Entwicklung ausgehend von der Geschäftsprozessebene durchführen, beinhalten in der Regel keine direkte Ableitung der AwS-Spezifikation, des Softwaredesigns und der Implementierung. So beschreiben González et al. (2011) lediglich die Ableitung von Klassendiagrammen aus Geschäftsprozessen, wobei der Schwerpunkt auf der Kommunikation zwischen einzelnen personellen Aufgabenträgern des Systems liegt. Bei den Ansätzen von Giner et al. (2007) und Montangero et al. (2011) wird von Workflows ausgegangen, die als Basis für eine Ableitung ausführbarer WS-BPEL-Prozessdefinitionen dienen. Im Gegensatz dazu verfolgen De Castro et al. (2011) einen umfassenderen Ansatz gemäß der MDA. Sie leiten ausgehend von einem Geschäftsprozess und einer Workflow-Definition (CIM) zunächst Use Cases und ein Service-Modell ab (PIM), das schließlich in Web-Services (PSM) transformiert wird. Hahn et al. (2010) verwenden ebenfalls den MDA-Ansatz, wobei auf Ebene des CIM eine Kombination aus BPMN und ARIS verwendet wird, die Ausgangspunkt für die Modellierung von Services auf Ebene des PIM ist. Bezüglich des PSM wird hier allerdings ein auf Agenten basierender Ansatz verwendet.

Zur SOM-Methodik existieren weitere Arbeiten, die eine modellbasierte Transformation in BPMN-Workflows (Pütz und Sinz 2011, S. 267-286) beschreiben oder auch die Ableitung einer SOA vorschlagen. Dabei beschreiben Teusch und Sinz (2012) u.a. die Identifikation von Services einer partiellen SOA, Wolf und Benker (2013) diskutieren die Ableitung einer RESTful SOA.

Im Unterschied zu den genannten Arbeiten führt der hier vorgestellte Ansatz die modellgetriebene Vorgehensweise bis zur softwaretechnischen Ebene konsequent weiter. Der Ansatz wird zunächst plattformneutral anhand von vier Ebenen

beschrieben. Das Geschäftsprozessmodell (1) wird in eine AwS-Spezifikation (2) überführt, die als Ausgangspunkt für das Softwaredesign (3) dient. Daran schließt die Generierung einzelner Softwareartefakte (4) an.

Zur Unterstützung durch das EMF folgt anschließend die Betrachung verschiedener Technologien. Dies führt zum Einsatz von Xtext und Ecore für die Modellierung sowie der Verwendung von QVTo und Xtend für die Transformation. Xtext ermöglicht die konzise Definition textueller Beschreibungssprachen anhand einer Grammatik. Für die Ebene zur Spezifikation von AwS wird eine entsprechende Grammatik entwickelt, mit der KOS und VOS der SOM-Methodik auf einfache Weise textuell modelliert werden können.

Der bisher abstrakt beschriebene Ansatz wird für die Entwicklung serviceorientierter Systeme konkretisiert. Eine Diskussion zur SOA zeigt, dass mit dem Architekturstil eine flexibel wählbare Granularität und die Kapselung einzelner Komponenten durch Services mit standardisierten Schnittstellen erreicht werden kann. Zur Modellierung serviceorientierter AwS wird ein Metamodell ausgehend von einer Xtext-Grammatik entwickelt. Ein vorliegendes Modell einer AwS-Spezifikation wird somit durch die Definition von Services zur SOA-AwS-Spezifikation erweitert. Dies schafft zusätzliche Flexibilität, da u.a. Freiheitsgrade bezüglich des Dienstschnitts bestehen.

Auf der Ebene des Softwaredesigns wird ein für Java EE entwickeltes Metamodell vorgestellt, mit dem eine komponentenorientierte Softwarearchitektur modelliert werden kann. Einzelne Services werden mit Session Beans modelliert, die mit orchestrierten Services synchron kommunizieren. Asynchrone Kommunikation wird durch Message Driven Beans realisiert.

Für die Transformation der SOA-AwS-Spezifikation in ein Modell für das Softwaredesign werden entsprechende Zuordnungsfunktionen in QVTo definiert. Die Implementierung der Zuordnungsfunktionen erweist sich als relativ komplex; sie ist aber für die Transformation mit Eclipse zweckmäßig.

Zur Generierung von Softwareartefakten wird ein hierfür entwickeltes Werkzeug vorgestellt, das ein Modell des Softwaredesigns in Quellcode überführt. Das Werkzeug ist in der Sprache Xtend implementiert, deren Template-Ausdrücke sich für die Definition von Vorlagen als hilfreich erweisen.

Die abschließende Fallstudie zeigt die Anwendung des entwickelten Ansatzes für ein fiktives Szenario zum Absatz von e-Cars. KOS und VOS werden in der entwickelten Beschreibungssprache notiert und anschließend um ein Service-Modell ergänzt. Die hierfür verwendeten textuellen Modelle sind präzise und verständlich, allerdings im Vergleich zur semiformalen Spezifikation mit Diagrammen weniger gut lesbar. Die Ableitung mit Hilfe von QVTo und Eclipse gestaltet sich problemlos.

Bei der Generierung von Softwareartefakten wird ein für die weitere Entwicklung als Basis dienender Rahmen erzeugt. Generierbar sind Schnittstellen, Entitäten und EJBs zur Realisierung der Services, Aufrufe zur Ablaufsteuerung, die asynchrone Kommunikation über Message Queues sowie Implementierungen für Getter-, Setter- und CRUD-Methoden. Die Entwicklung anderer Methoden kann durch Erweiterung dafür vorgesehener Klassen erfolgen.

Ein sinnvoller Einsatz der vorgeschlagenen modellgetriebenen Vorgehensweise ist nur möglich, wenn Änderungen, die eine Anpassung des Systems erfordern, von der Geschäftsprozessebene ausgehen. Sie werden durch erneute Ableitung einer AwS-Spezifikation, eines Softwaredesign-Modells und neuem Quellcode von oben nach unten propagiert. Softwaretechnische Erweiterungen, die beispielsweise einzelne Lösungsverfahren implementieren, müssen an definierten Punkten vorgenommen werden. Das Programm wird somit explizit für die Wiederverwendung entwickelt und sieht eine mehrstufige Erweiterung der generierten Klassen vor. Sofern keine größeren Strukturänderungen erfolgen, bleiben Anpassungen somit auch bei erneuter Durchführung der Ableitung erhalten.

Die Transformation eines Geschäftsprozessmodells bis hin zum Softwareartefakt zeigt das Potenzial der modellgetriebenen Systementwicklung. So ist etwa auch die Abbildung eines Modells auf unterschiedliche Plattformen möglich. Entscheidend ist die Entkoppelung von fachlichen und softwaretechnischen Belangen auf verschiedenen Modellebenen, die über Transformationen ineinander überführt werden. Die modellgetriebene Systementwicklung leistet damit einen Beitrag zur Steigerung der Flexibilität und zur Reduktion der Komplexität bei der Entwicklung von Systemen.

Anhang

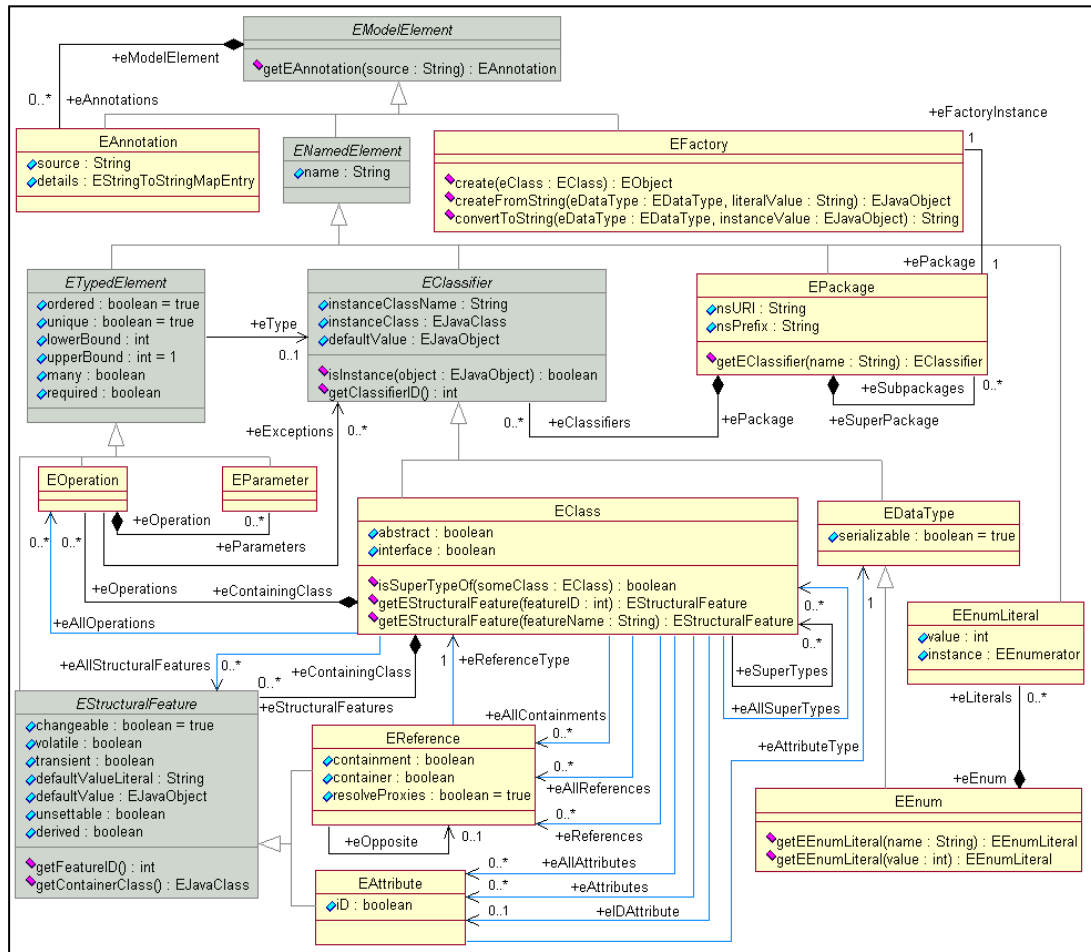

Abbildung 63: Ecore Metamodell (Eclipse 2013)

```
grammar de.uniba.wiai.seda.fha.som.aws.soa.Soa with
de.uniba.wiai.seda.fha.som.aws.kosvos.Kosvos

import "http://www.uniba.de/wiai/seda/fha/som/aws/Aws" as aws
import "http://www.uniba.de/wiai/seda/fha/som/aws/kosvos/Kosvos" as kosvos

generate soa "http://www.uniba.de/wiai/seda/fha/som/aws/soa/Soa"

// SoaAws has KosVos, Vorgangs-Services, Entitaets-Services
SoaAws: aws=KosVos (vorgangsServices+=VorgangsServiceNichtElementar |
                    entitaetsServices+=EntitaetsService)+;

// Generalisierung der Service-Klassen
Service: EntitaetsService | VorgangsService;
VorgangsService: VorgangsServiceNichtElementar | VorgangsServiceElementar;

// EntitaetsService has Name, Interface, Entitäten
EntitaetsService: "EntitaetsService" name=ID "{"
   interface=Interface entitaeten+=Entitaet+ "}";

// Entität has KOT
Entitaet: "KOT" kot=[aws::KOT]";";

// VorgangsServiceNichtElementar has Name, Interface, Komposition, Workflow
// (Choreo./Orchestr.)
VorgangsServiceNichtElementar: "VorgangsService" name=ID     "{"
  interface=Interface komposition+=VorgangsServiceElementar*
  ("Workflow" "{" (vsChoreografie+=Choreografie |
                   vsOrchestrierung+=Orchestrierung)* "}")*  "}";

// VorgangsServiceElementar has Name, Interface, Orchestr., VOT
VorgangsServiceElementar: "VorgangsService" name=ID                  "{"
  interface=Interface
  (esOrchestrierung+=EntitaetsServiceOrchestrierung)*
  "VOT" votRequest=[aws::VOT]";" ("VOT" votResponse=[aws::VOT]";")?  "}";

// Interface has Name, Methoden
Interface: name="Interface" "{" methoden+=ServiceMethode+ "}";

// Service_Methode has Name, Input/Output-Parameter
ServiceMethode: outParam=ID? name=ID "("inParam+=ID*")"";";

// Generalisierung der Service-Beziehungen zur Choreografie und Orchestrierung
ServiceBeziehung: Choreografie | Orchestrierung;
Orchestrierung: VorgangsServiceOrchestrierung | EntitaetsServiceOrchestrierung;

// Service-Beziehungen mit aufrufender Methode (serviceMethode1) auf
// aufgerufener Methode (serviceMethode2)
Choreografie:
  serviceMethode1=[ServiceMethode] ">>"
  service=[Service]"."serviceMethode2=[ServiceMethode]";";

VorgangsServiceOrchestrierung:
  serviceMethode1=[ServiceMethode] "->" serviceMethode2=[ServiceMethode]";";

EntitaetsServiceOrchestrierung:
  "EntitaetsService" (serviceMethode1=[ServiceMethode] "->")?
  service=[Service]"."serviceMethode2=[ServiceMethode]";";
```

Abbildung 64: Grammatik des SOA-AwS-Metamodells

```
grammar de.uniba.wiai.seda.fha.swarchitektur.Swdesign with
        org.eclipse.xtext.common.Terminals

generate swdesign
         "http://www.uniba.de/wiai/seda/fha/swarchitektur/Swdesign"

// Design has Modules
Design: name=ID ("v"version=ID)? modules+=EJBModule*;

// EJBModule has Name, EJBs, Interfaces, Entities
EJBModule: "EJBModule" name=ID   "{" (ejb+=EJB | interfaces+=BusinessInterface |
           entities+=JPAEntity)+ "}";

// Generalisierungsbeziehungen zu EJBs
EJB: SessionBean | MessageDrivenBean;
SessionBean: StatelessSessionBean | StatefulSessionBean;

// BusinessInterface has Name, Package, Methodensignaturen
BusinessInterface: "BusinessInterface" package=PACKAGE_ID name=ID "{"
        methoden+=Methodensignatur* "}";

// Methodensignatur has Name, Input/Output-Typen
Methodensignatur: outputTyp=ID name=ID "(" inputTypen+=ID?
        ("," inputTypen+=ID)* ")" ";";

// Methode has Name, Input/Output-Typen, Invokes
Methode: outputTyp=ID name=ID "(" inputTypen+=ID?
        ("," inputTypen+=ID)* ")" "{" invoke+=Invoke* "}";

// Generalisierungsbeziehungen zu Invokes
Invoke: InvokeEJBSync | InvokeQueueAsync;

// InvokeEJBSync has EJB, Methodensignatur, Parameter-Typen
InvokeEJBSync: ejb=[SessionBean]"."methodensignatur=[Methodensignatur] "("
       parameterTypen+=ID? ("," parameterTypen+=ID)* ")" ";";

// InvokeQueueAsync has Queue-Name, Nachrichten-Typen
InvokeQueueAsync: "queueMessage" "(" queueName=[MessageQueue]
       ("," nachrichtenTypen+=ID)+ ")" ";";

// StatefulSessionBean hase Name, Package, Attribute, Methoden
StatefulSessionBean: "StatefulSessionBean" package=PACKAGE_ID name=ID "{"
       attribute+=Attribut* methode+=Methode* "}";

// StatelessSessionBean has Name, Package, Attribute, Methoden
StatelessSessionBean: "StatelessSessionBean" package=PACKAGE_ID name=ID "{"
       attribute+=Attribut* methode+=Methode* "}";

// MessageDrivenBean has Name, Package, Message Queue, Attribute, Methoden
MessageDrivenBean: "MessageDrivenBean" package=PACKAGE_ID name=ID
       queue=MessageQueue "{" attribute+=Attribut* methode+=Methode* "}";

// MessageQueue has Name, Nachrichten Typen
MessageQueue: "JMSQueue" name=ID nachrichtenTypen+=ID;

// JPAEntity has Name, Attribute, Methoden
JPAEntity: "JPAEntity" package=PACKAGE_ID name=ID "{"
       attribute+=Attribut* methode+=Methode* "}";

// Attribut has Typ, Name
Attribut: typ=ID name=ID ";";

terminal PACKAGE_ID: '^'? ('a'..'z'|'_') ('a'..'z'|'_'|'0'..'9'|'.')*;
```

Abbildung 65: Grammatik des Softwaredesign-Metamodells

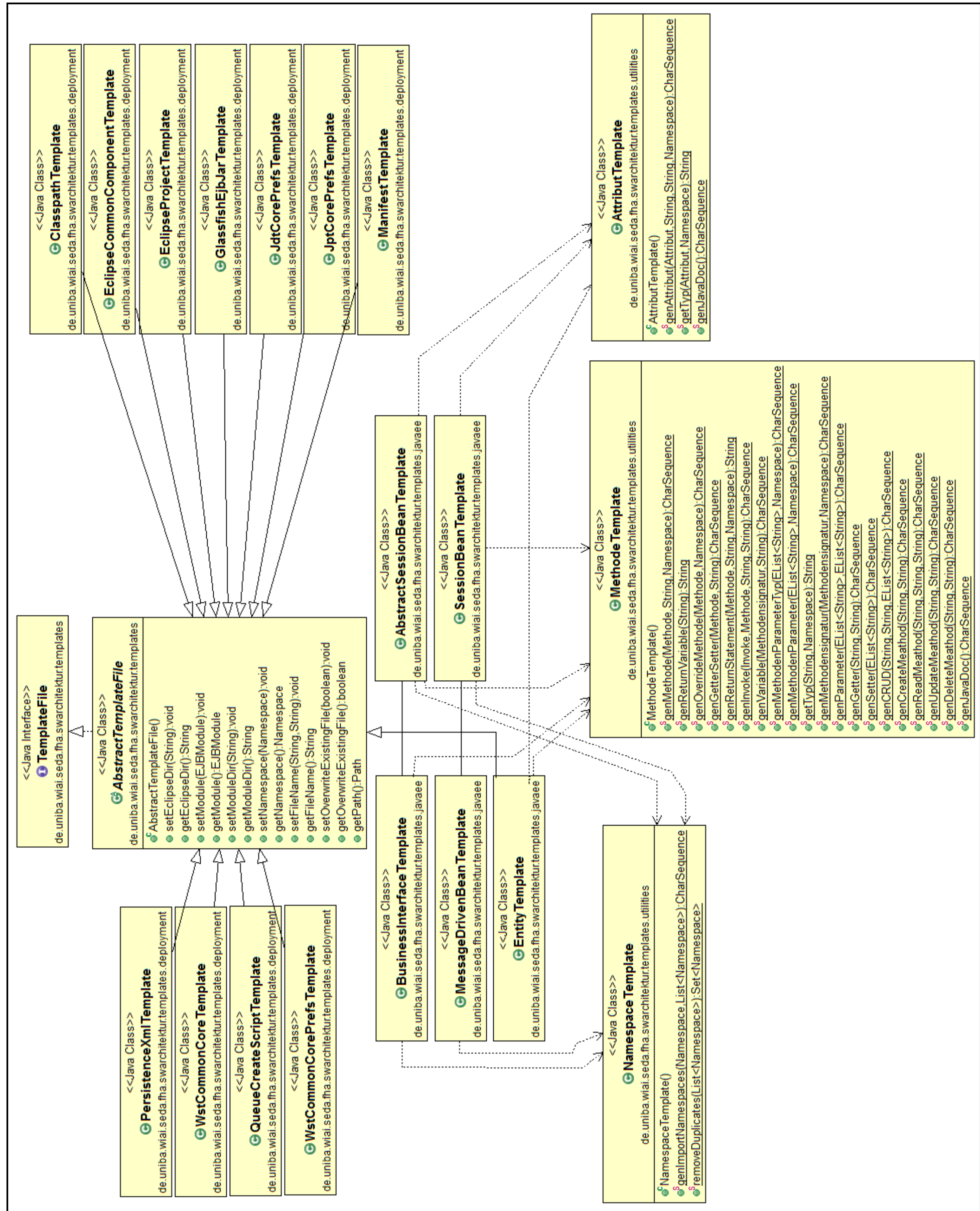

Abbildung 66: UML-Klassendiagramm der Templates zur Codegenerierung

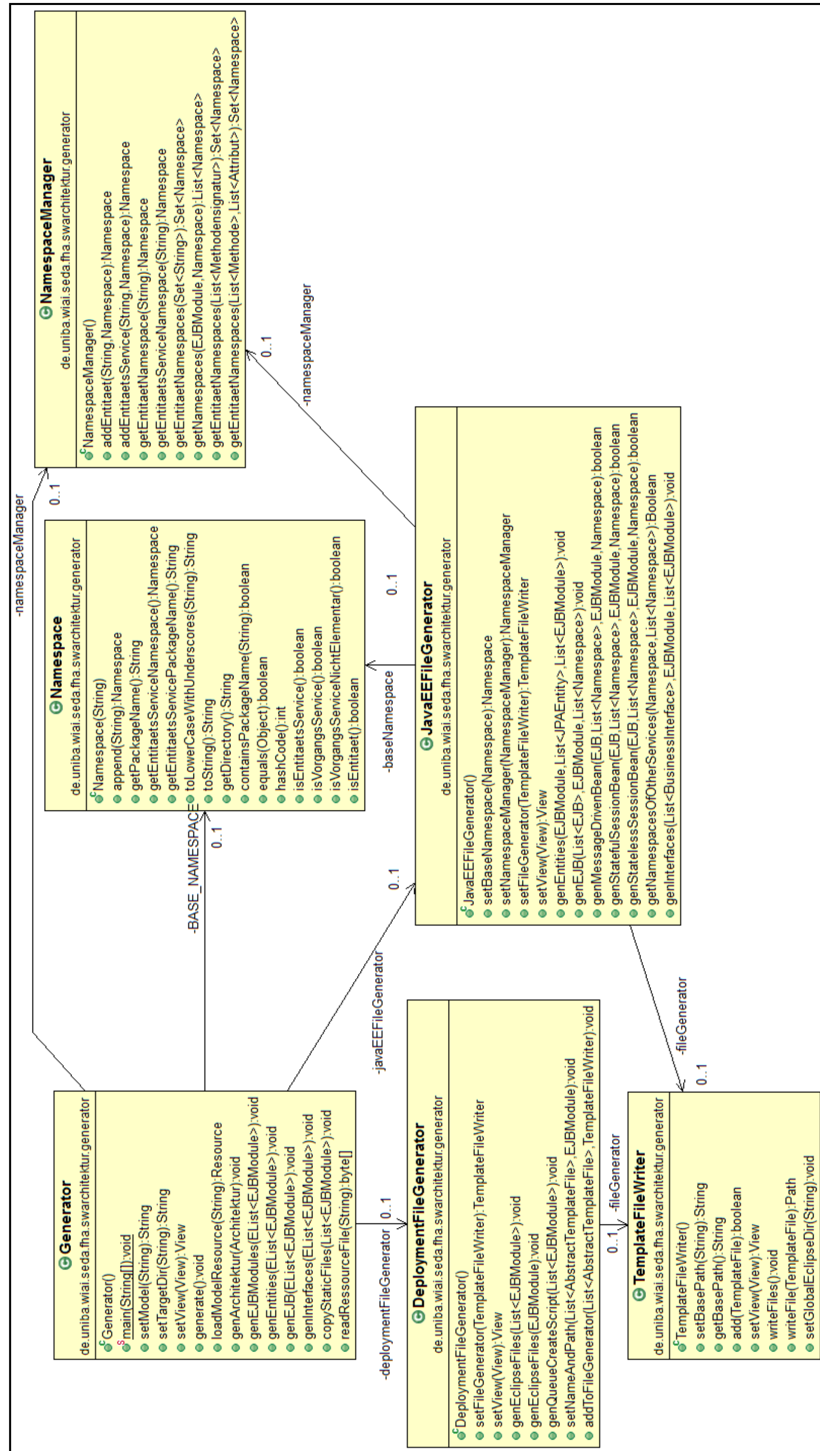

Abbildung 67: UML-Klassendiagramm des Werkzeugs zur Codegenerierung

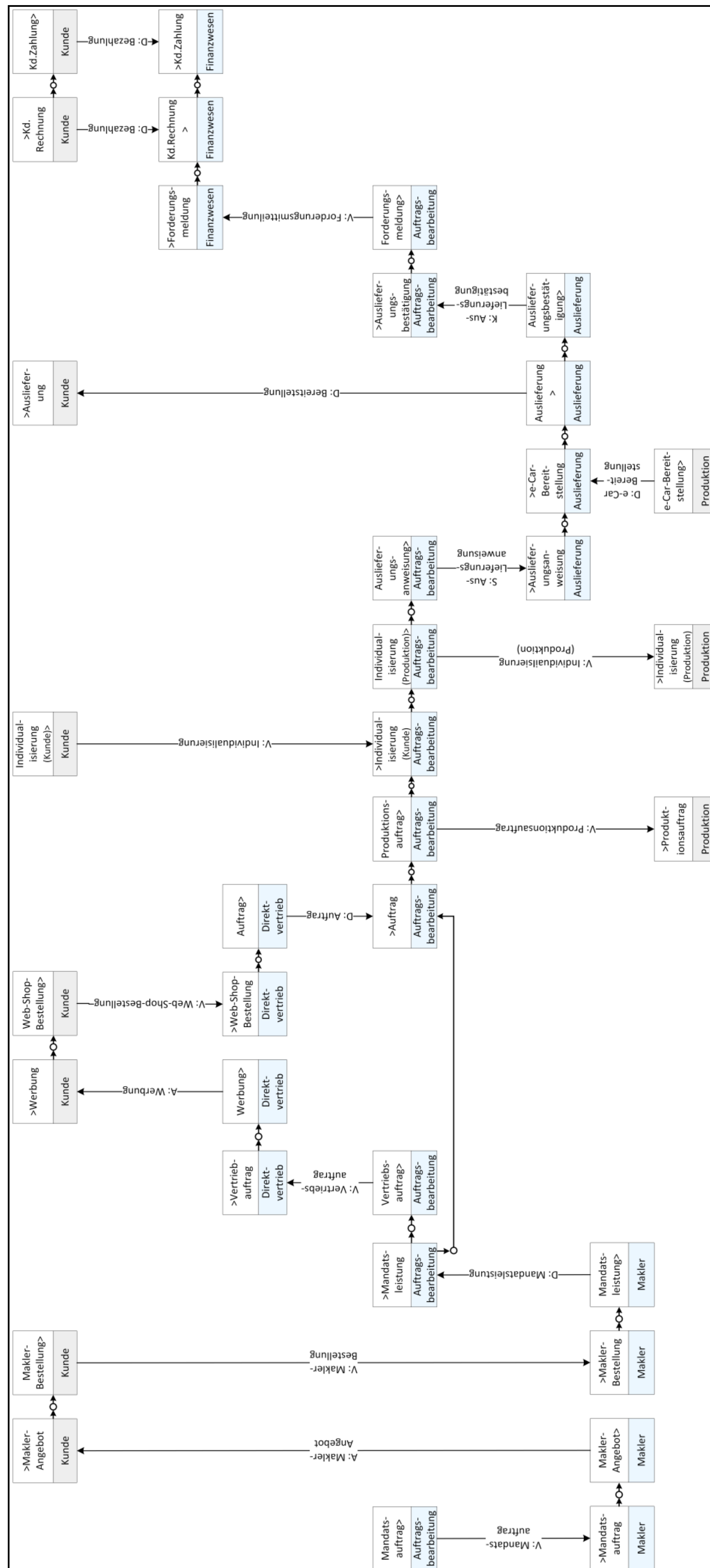

Abbildung 68: Vorgangs-Ereignis-Schema zum Absatz von e-Cars

Literaturverzeichnis

Bézivin, Jean; Hammoudi, Slimane; Lopes, Denivaldo; Jouault, Frédéric (2004): Applying MDA Approach for Web Service Platform. In: Bryant, Barrett R.; Akehurst, David H.; Van Sinderen, Marten; Mukerji, Jishnu; Auguston, Mikhail (Hrsg.): Proceedings of the 8th Enterprise Distributed Object Computing Conference, IEEE International (EDOC 2004). Monterey, California, USA, S. 58 - 70.

Brambilla, Marco; Cabot, Jordi; Wimmer, Manuel (2012): Model-Driven Software Engineering in Practice. 1. Aufl., Morgan & Claypool Publishers, San Rafael, California, USA.

Bork, Domenik; Sinz, Elmar J. (2011): Ein Multi-View-Modellierungswerkzeug für SOM-Geschäftsprozessmodelle auf Basis der Meta-Modellierungsplattform ADOxx. In: Sinz, Elmar J.; Bartmann, Dieter; Bodendorf, Freimut; Ferstl, Otto K. (Hrsg.): Dienstorientierte IT-Systeme für hochflexible Geschäftsprozesse. University of Bamberg Press, Bamberg, S. 367 - 384.

Clarke, Edmund M.; Grumberg, Orna; Peled, Doron (1999): Model Checking. 1. Aufl., MIT Press, Cambridge, Massachusetts, USA.

De Castro, V.; Marcos, E.; Vara, J.M. (2011): Applying CIM-to-PIM model transformations for the service-oriented development of information systems. In: Information and Software Technology 53 (1), S. 87 - 105.

Eclipse (2011): Xtext 2.1 Documentation. http://www.eclipse.org/Xtext/documentation/2_1_0/Xtext%202.1%20Documentation.pdf, Abruf am 2014-02-02.

Eclipse (2013): Ecore Components. http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html, Abruf am 2014-02-02.

Ferstl, Otto K.; Sinz, Elmar J. (2013): Grundlagen der Wirtschaftsinformatik. 7. Aufl., Oldenbourg, München.

Ferstl, Otto K.; Sinz Elmar J.; Hammel, Ch.; Schlitt, M.; Wolf, St.; Popp, K.; Kehlenbeck, R.; Pfister, A.; Kniep, H.; Nielsen, N.; Seitz, A. (1998): WEGA - Wiederverwendbare und erweiterbare Geschäftsprozeß- und Anwendungssystemarchitekturen. Abschlussbericht, Walldorf.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994): Design Patterns. Elements of Reusable Object-Oriented Software. 1. Aufl., Pearson Education, Boston, Massachusetts, USA.

Giner, Pau; Torres, Victoria; Pelechano, Vicente (2007): Bridging the Gap between BPMN and WS-BPEL. M2M Transformations in Practice. In: Koch, Nora; Vallecillo, Antonio; Houben, Geert-Jan (Hrsg.): Proceedings of the 3rd International Workshop on Model-Driven Web Engineering (MDWE 2007). Como, Italien.

González, Arturo; España, Sergio; Ruiz, Marcela; Pastor, Òscar (2011): Systematic Derivation of Class Diagrams from Communication-Oriented Business Process Models. In: Halpin, T.; Nurcan, S.; Krogstie, J.; Soffer, P.; Proper, E.; Schmidt, R.; Bider, I. (Hrsg.): Proceedings 12th International Conference, BPMDS 2011, and 16th International Conference, EMMSAD 2011, held at CAiSE 2011, London, UK. Springer, Heidelberg, S. 246 - 260.

Gronback, Richard C. (2009): Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. 1. Aufl., Pearson Education, Boston, Massachusetts, USA.

Hahn, Christian; Panfilenko, Dmytro; Fischer, Klaus (2010): A model-driven approach to close the gap between business requirements and agent-based execution. In: Van Der Hoek, Wiebe; Kaminka, Gal A.; Lespérance, Yves; Luck, Michael; Sen, Sandip (Hrsg.): Proceedings of the 4th Workshop on Agent-based Technologies and applications for enterprise interoperability: International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-10). Toronto, Canada, S. 13 - 24.

Heineman, George T.; Councill, William T. (2001): Component-Based Software Engineering: Putting the Pieces Together. 1. Aufl., Pearson Education, Boston, Massachusetts, USA.

Jouault, Frédéric; Kurtev, Ivan (2005): Transforming models with ATL. In: Briand, Lionel; Williams, Clay (Hrsg.): Model Driven Engineering Languages and Systems, 8th Intenational Conference, MoDELS 2005. Montego Bay, Jamaica, S. 128 - 138.

Jouault, Frédéric; Allilaire, Freddy; Bézivin, Jean; Kurtev, Ivan; Valduriez, Patrick (2006): ATL: a QVT-like Transformation Language. In: Peri, L. Tarr; Cook, William R. (Hrsg.): 21st ACM SIGPLAN symposium on Object-oriented programming systems (OOPSLA 2006). Portland, Oregon, USA, S. 719 - 720.

Kent, Stuart (2002): Model Driven Engineering. In: Butler, Michael; Petre, Luigia; Sere, Kaisa (Hrsg.): Third International Conference, IFM 2002. Turku, Finland, S. 286 - 298.

Leunig, Benjamin; Wagner, Daniel; Ferstl, Otto K. (2011): E-Car-Szenario – Hochflexible Geschäftsprozesse in der Logistik als Integrationsszenario für den Forschungsverbund forFLEX. In: Sinz, Elmar J.; Bartmann, Dieter; Bodendorf, Freimut; Ferstl, Otto K. (Hrsg.): Dienstorientierte IT-Systeme für hochflexible Geschäftsprozesse. University of Bamberg Press, Bamberg, S. 15 - 38.

Montangero, Carlo; Reiff-Marganiec, Stephan; Semini, Laura (2011): Model-Driven Development of Adaptable Service-Oriented Business Processes. In: Wirsing, Martin; Hölzl, Matthias (Hrsg.): Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing, Lecture Notes in Computer Science, Vol. 6582. Springer Verlag, Berlin, S. 115 - 132.

OASIS (2007): Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, Abruf am 2014-02-02.

OMG (2001): Model Driven Architecture (MDA). Document Number: ormsc/01-07-01. http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, Abruf am 2014-02-02.

OMG (2003a): MDA Guide Version 1.0.1. Document Number: omg/2003-06-01. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf, Abruf am 2014-02-02.

OMG (2003b): Common Warehouse Metamodel, v1.1. Document Number: formal/2003-03-02. http://www.omg.org/spec/CWM/1.1/, Abruf am 2014-02-02.

OMG (2005a): Meta Object Facility (MOF), v1.4.1, ISO PAS 19502. Document Number: formal/05-05-05. http://www.omg.org/cgi-bin/doc?formal/05-05-05, Abruf am 2014-02-02.

OMG (2005b): XML Metadata Interchange (XMI), v2.0.1, ISO PAS 19503. Document Number: formal/05-05-06. http://www.omg.org/cgi-bin/doc?formal/05-05-06, Abruf am 2014-02-02.

OMG (2010): Object Constraint Language, v2.2. Document Number: formal/10-02-01. http://www.omg.org/cgi-bin/doc?formal/10-02-01, Abruf am 2014-02-02.

OMG (2011): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Document Number: formal/2011-01-01. http://www.omg.org/spec/QVT/1.1, Abruf am 2014-02-02.

Oracle (2013a): JSR 345: Enterprise JavaBeans, Version 3.2. EJB Core Contracts and Requirements. https://jcp.org/en/jsr/detail?id=345, Abruf am 2014-02-02.

Oracle (2013b): JSR 338: Java Persistence API, Version 2.1. https://jcp.org/en/jsr/detail?id=338, Abruf am 2014-02-02.

Peltz, Chris (2003): Web Services Orchestration and Choreography. In: IEEE Computer 36 (10), S. 46 - 52.

Pütz, Corinna; Sinz, Elmar J. (2011): Vom SOM-Geschäftsprozessmodell zum BPMN-Workflowschema. In: Sinz, Elmar J.; Bartmann, Dieter; Bodendorf, Freimut; Ferstl, Otto K. (Hrsg.): Dienstorientierte IT-Systeme für hochflexible Geschäftsprozesse. University of Bamberg Press, Bamberg, S. 267 - 286.

Schmidt, Douglas C. (2006): Model-Driven Engineering. In: IEEE Computer 39 (2), S. 25 - 31.

Shaw, Mary; Garlan, David (1996): Software Architecture: Perspectives on an Emerging Discipline. 1. Aufl., Prentice Hall, Upper Saddle River, New Jersey, USA.

Siedersleben, Johannes (2007): SOA Revisited: Komponentenorientierung bei Systemlandschaften. In: Wirtschaftsinformatik 49, S. 110 - 117.

Sinz, Elmar J. (2002): Architektur von Informationssystemen. In: Rechenberg, Peter; Pomberger, Gustav (Hrsg.): Informatik-Handbuch. 3. Aufl., Hanser-Verlag, München, S. 1055 - 1068.

Sommerville, Ian (2011): Software Engineering. 9. Aufl., Pearson Education, Boston, Massachusetts, USA.

Stahl, Thomas; Völter, Markus; Efftinge, Sven (2007): Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management. 2. Aufl., DPunkt Verlag, Heidelberg.

Steinberg, David; Budinsky, Frank; Paternostro, Marcelo; Merks, Ed (2008): EMF: Eclipse Modeling Framework. Second Edition, Pearson Education, Boston, Massachusetts, USA.

Teusch, Andree; Sinz, Elmar J. (2011): Entwurf partieller SOA auf der Grundlage von Geschäftsprozessmodellen. In: Sinz, Elmar J.; Bartmann, Dieter; Bodendorf, Freimut; Ferstl, Otto K. (Hrsg.): Dienstorientierte IT-Systeme für hochflexible Geschäftsprozesse. University of Bamberg Press, Bamberg, S. 287 - 312.

Teusch, Andree; Sinz, Elmar J. (2012): Konzeptuelle Modellierung partieller SOA. In: Mattfeld, D.J.; Robra-Bissantz, S. (Hrsg.): Multikonferenz Wirtschaftsinformatik 2012. Tagungsband der Multikonferenz Wirtschaftsinformatik 2012, GITO mbH Verlag Berlin, S. 1637 - 1648.

Voelter, Markus (2013): DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. http://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf, Abruf am 2014-02-02.

W3C (2004): Web Services Architecture. http://www.w3.org/TR/ws-arch/, Abruf am 2014-02-03.

W3C (2007): Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts. http://www.w3.org/TR/wsdl20-adjuncts/, Abruf am 2014-02-03.

Wolf, Matthias; Benker, Thomas (2013): Vom SOM-Geschäftsprozessmodell zur vollständig dokumentenorientierten RESTful SOA - Ein modellbasierter Ansatz. In: Alt, Rainer; Franczyk, Bogdan (Hrsg.): Proceedings der 11. Internationalen Tagung Wirtschaftsinformatik (WI2013). Leipzig, S. 1229 - 1243.

Erklärung

Ich erkläre hiermit gemäß § 17 Abs. 2 APO, dass ich die vorstehende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum	Unterschrift